

Documentation

Version 0.8.5

November 11, 2026

This edition

Doriax Engine Documentation, version 0.8.5. Generated from `v0.8.5` (commit `a0d7853`, November 11, 2026) of `github.com/doriaxengine/doriax-docs`. The current documentation is online at `docs.doriax.org`.

Reading on paper

- Code shown in tabs on the website is printed in full, one language after the other, with a **C++** or **Lua** label on each block.
- References to other chapters carry the page they point to, as in "Scripting (p. 195)". In a PDF viewer they also work as links.
- Web addresses appear as footnotes.
- The Reference part holds every class page, grouped as in the API Index.

License

Copyright © 2026 Eduardo Doria. Documentation under CC-BY 4.0.

Contents

Welcome 6

Part 1 · Getting Started

Introduction 10
Installation 12
Your First Project 14
The Editor 19

Part 2 · Tutorials

First 2D Scene 26
First 3D Scene 31
First UI Scene 35
Exporting a Project 40

Part 3 · Editor

Editor Overview 44
Project Workflow 48
Project Settings 55
Version Control 65
Editor Settings 68
Structure Panel 73
Scene View 81
Properties & Components 90
Resources Browser 106
Bundles 111
Sprite Slicer 116
Tileset Slicer 119
Terrain Editor 123
Animation Timeline 129
Code Editor 136
Custom Shaders 143
Export Window 155
Command-Line Tools 167
MCP Server 172

Part 4 · Manual

Engine Architecture 180
Scenes & Entities 182
Entity Component System 190
Scripting 195
Creating Scripts 199
C++ Build Setup 209
Events 215
Script Properties 223
Input 229
Resources & Assets 238
2D Graphics 243
3D Graphics 248
Rendering Pipeline 258
Shader Variants 278
Shader Effects 282
Multiple Resolutions 288
User Interface 292
Animation 298
Particles 306
Terrain 309
Physics 315
Audio 326
Math & Utilities 329
Threading & Async Loading 333
Monetization 337

Part 5 · Reference

API Index 352
Enumerations 355
Build Options 357

Core

Engine 361
EntityManager 378
EntityRegistry 379
Input 380
Log 387
Scene 390
System 407

Objects

Bone 413
Camera 414
Fog 422
Light 424
Light2D 430
Lines 433
Mesh 436
Mirror 448
Model 450
Object 455
Occluder2D 462
Points 464
ReflectionProbe 468
Shape 471
SkyBox 472
Sound 475
Sprite 480
Terrain 486
Tilemap 492
Water 496
WaterExclusion 503

UI

Button 506
Container 510
Image 513
Panel 518
Polygon 521
Progressbar 523
Scrollbar 525
Text 528
TextEdit 533
UILayout 537

Physics

Body2D 541
Body3D 551
CollideShapeResult3D 561
Contact2D 562
Contact3D 562
Joint2D 563
Joint3D 564
Manifold2D 566
PhysicsSystem 567

Actions
Action ... 569
AlphaAction ... 572
Animation ... 573
ColorAction ... 577
KeyframeTracks ... 578
MorphTracks ... 579
PositionAction ... 580
RotateTracks ... 581
RotationAction ... 582
ScaleAction ... 584
ScaleTracks ... 585
SpriteAnimation ... 586
TimedAction ... 588
TranslateTracks ... 590

Systems
ActionSystem ... 590
AudioSystem ... 591
MeshSystem ... 592
RenderSystem ... 594
SubSystem ... 594
UISystem ... 595

Managers
BundleManager ... 597
SceneManager ... 601

Services
AdMob ... 610
InAppPurchase ... 618
PricingPhase ... *623*
ProductDetails ... *622*
ProductOffer ... *622*
PurchaseDetails ... *623*
WebPortal ... 626

Scripting
ScriptBase ... 630

Math
AABB ... 632
Angle ... 636
Matrix3 ... 637
Matrix4 ... 638
OBB ... 639
Plane ... 642
Quaternion ... 644
Ray ... 649
Rect ... 652
Sphere ... 654
Vector2 ... 656
Vector3 ... 660
Vector4 ... 664

Components
ActionComponent ... 666
AlphaActionComponent ... 666
AnimationComponent ... 667
Body2DComponent ... 667
Body3DComponent ... 668
BoneComponent ... 668
ButtonComponent ... 669
CameraComponent ... 670
ColorActionComponent ... 671
FogComponent ... 671
ImageComponent ... 672
InstancedMeshComponent ... 672
Joint2DComponent ... 673
Joint3DComponent ... 674
KeyframeTracksComponent ... 675
Light2DComponent ... 676
LightComponent ... 677
LinesComponent ... 678
MeshComponent ... 679
MeshPolygonComponent ... 680
MirrorComponent ... 680
ModelComponent ... 681
MorphTracksComponent ... 681
Occluder2DComponent ... 682
PanelComponent ... 682
ParticlesComponent ... 683
PointsComponent ... 684
PolygonComponent ... 684
PositionActionComponent ... 685
ProgressbarComponent ... 685
ReflectionProbeComponent ... 686
RotateTracksComponent ... 686
RotationActionComponent ... 687
ScaleActionComponent ... 687
ScaleTracksComponent ... 687
ScrollbarComponent ... 688
SkyComponent ... 688
SoundComponent ... 689
SpriteAnimationComponent ... 690
SpriteComponent ... 691
TerrainComponent ... 692
TextComponent ... 693
TextEditComponent ... 694
TilemapComponent ... 695
TimedActionComponent ... 695
Transform ... 696
TranslateTracksComponent ... 697
UIComponent ... 698
UIContainerComponent ... 699
UILayoutComponent ... 700
WaterComponent ... 701
WaterExclusionComponent ... 702

I/O
Data ... 703
File ... 705
FileData ... 706
UserSettings ... 707

Other
ActionFrame ... 708
Base64 ... 708
Color ... 709
EntityHandle ... 711
FramebufferRender ... 712
InstanceData ... 713
LineData ... 713
MeshPolygon ... 713
OverallBuildProgress ... 714
PointData ... 714
PolygonPoint ... 714
PostProcessPass ... 715
RayReturn ... 716
ResourceBuildInfo ... 717
ResourceProgress ... 717
TerrainFoliageLayer ... 718
TextureLoadResult ... 718

TextureRender ... 718
ThreadPoolManager ... 719
TileData ... 719
TileRectData ... 719

Particles

ParticleAccelerationInitializer ... 720
ParticleAccelerationModifier ... 720
ParticleAlphaInitializer ... 720
ParticleAlphaModifier ... 720
ParticleBurst ... 721
ParticleColorGradient ... 721
ParticleColorGradientStop ... 721
ParticleColorInitializer ... 721
ParticleColorModifier ... 722
ParticleLifeInitializer ... 722
ParticlePositionInitializer ... 722
ParticlePositionModifier ... 723
ParticleRotationInitializer ... 723
ParticleRotationModifier ... 723
Particles ... 724
ParticleScaleInitializer ... 731
ParticleScaleModifier ... 731
ParticleSizeInitializer ... 731
ParticleSizeModifier ... 732
ParticleSpriteInitializer ... 732
ParticleSpriteModifier ... 732
ParticleVelocityInitializer ... 732
ParticleVelocityModifier ... 733

Textures

Framebuffer ... 733
Material ... 734
Texture ... 738
TextureData ... 744

Part 6 · Building from Source

Overview ... 746
Build Script ... 748
Windows ... 750
Linux ... 752
macOS ... 754
Android ... 756
iOS ... 758
HTML5 ... 760

Part 7 · Contributing

Documentation Guide ... 764

Part 8 · About

FAQ ... 768
License & Credits ... 771

Open Source & Free

Lightweight engine for 2D & 3D games

Welcome to the official documentation for **Doriax Engine** — a free, lightweight, and efficient open-source game engine with an integrated editor for creating 2D and 3D games. Script in **Lua** or **C++**, design scenes visually, and deploy to six platforms.

Get Started (p. 10) **Download the Editor**[1]

> **Documentation status**
> This documentation covers the editor workflow, runtime architecture, core subsystems, build targets, and the current public API surface exposed by the Doriax source tree. Doriax is moving quickly, so the reference pages are written to be practical guides rather than frozen ABI guarantees.

Start here

Introduction (p. 10)
What Doriax is, its architecture, and what you can build with it.

Entity Component System (p. 190)
Understand the data-oriented ECS at the core of the engine.

Installation (p. 12)
Download a prebuilt editor or build Doriax from source.

API Reference (p. 352)
Complete class reference with C++ and Lua on every page.

Your First Project (p. 14)
Create a scene, add an entity, and run your first game.

Why Doriax

2D & 3D Engine

Full support for both 2D and 3D development — sprites, tilemaps, 3D models, skeletal animation, and more in one lean workflow.

Built-in Editor

Scene hierarchy, properties inspector, animation timeline, integrated code editor, and resource management — all in one place.

Lua & C++

Prototype quickly in Lua or compile C++ at build time for maximum native performance. Combine both seamlessly.

Entity Component System

A modern, data-driven ECS that maximizes CPU cache performance while keeping the engine lightweight and efficient.

Cross-Platform

Deploy to Windows, Linux, macOS, Android, iOS, and HTML5 with OpenGL, Vulkan, Metal, and DirectX backends.

PBR Rendering

Physically-based rendering with dynamic shadows, fog, sky system, water, and advanced materials for photorealistic visuals.

1. https://doriax.org/#download

Editor Export Pipeline

Scene serialization, script generation, shader compilation, and platform-specific project generation live in the editor.

The editor



Doriax ships with a complete visual editor: design scenes, edit 2D tilemaps, animate characters, write code, and test your game in play mode — all from one unified environment. Learn more in The Editor (p. 44).

> **Coming from Supernova?**
>
> Doriax Engine is the continuation of **Supernova Engine** under a new identity. Version `0.5.5` was the last legacy Supernova release. The core ECS and data-oriented runtime remain, now wrapped in a full editor and export pipeline.

PART 1

Getting Started

Introduction 10
Installation 12
Your First Project 14
The Editor 19

Introduction

Doriax Engine is a free, open-source 2D/3D game engine with an integrated editor for building cross-platform games and interactive projects. It combines a lightweight, data-oriented runtime with a modern desktop editor, so you can author scenes visually, script in Lua or C++, and export to multiple platforms from a single environment.

Doriax is the continuation of **Supernova Engine** under a new identity and a broader workflow focused on visual editing, shader generation, and project export.

Design philosophy

Doriax is built around a few core principles:

- **Lightweight and efficient** — no heavy GUI overhead or unnecessary processing. The engine stays lean and produces small output.
- **Data-oriented design** — an Entity Component System (ECS) arranges data to take advantage of CPU caching, avoiding the performance pitfalls of traditional object-oriented designs.
- **Two languages, one workflow** — use **Lua** for rapid iteration and flexible scripting, or **C++** for full native performance. C++ scripts are compiled at build time, not runtime.
- **Independent runtime** — export turns scenes and bundles into C++ that builds and runs without the editor. See how export works (p. 155).
- **Write once, deploy everywhere** — a single project can target six platforms with native graphics backends.

Documentation structure

This documentation follows a broad structure similar to large engine manuals:

Section	Use it for
Getting Started	Install Doriax, understand the editor, and create a first project
Tutorials	Step-by-step 2D, 3D, and export walkthroughs
Editor	Panels, project workflow, assets, code editing, animation, and export tools
Manual	Engine concepts and subsystem guides
Reference	API maps, components, Lua bindings, and build options
Building	Platform-specific toolchain setup

What you can build

Capability	Description
2D games	Sprites, tilemaps, sprite slicing, and 2D physics
3D games	GLTF/OBJ/FBX models, PBR materials, dynamic shadows, fog, sky, and water
Animation	Skeletal animation, morph targets, and a timeline editor
UI	A built-in UI system for menus, HUDs, and overlays
Physics	Integrated 2D and 3D physics via Box2D and Jolt Physics
Audio	3D positional audio
Monetization	AdMob ads and in-app purchases (Google Play and the App Store) on mobile, and the SDKs of five web game portals — see Monetization (p. 337)

Engine features

- 2D and 3D scenes with a shared ECS foundation
- GLTF and OBJ model loading, and FBX import
- Skeletal animation and morph targets
- PBR materials, dynamic shadows, fog, sky, and water
- Particle systems, UI, terrain LOD, mesh LOD, and instancing
- Scene serialization, texture and shader pools, and multithreading
- 3D audio
- Google AdMob ads, Google Play and App Store in-app purchases, and web game portal SDKs

> **Custom shaders**
> Fork, edit, and customize the built-in shaders for Mesh, UI, Points, Lines, Sky, and Water components directly in the editor — see Custom Shaders (p. 143).

Platforms

Area	Support
Editor downloads	Windows, Linux, macOS
Project targets	Windows, Linux, macOS, Android, iOS, HTML5
Graphics APIs	OpenGL, Vulkan, Metal, DirectX
Scripting	Lua, C++

Next steps

Installation → (p. 12)
Download the editor or build Doriax from source.

Architecture → (p. 180)
Learn how the runtime is organized internally.

Your First Project → (p. 14)
Build and run your first Doriax scene.

Installation

There are two ways to get Doriax Engine: download a prebuilt editor, or build it from source.

Download a prebuilt editor

The fastest way to start is to grab a prebuilt build of the editor from the official website:

Download Doriax Engine[2]

Builds are available for:

Platform	Requirements
Windows	Windows 10+ · x64
Linux	Ubuntu 22.04+ · x64
macOS	macOS 12+ · Universal

Each platform offers the latest release, and beside it a build of the current `main` branch.

> **Main branch builds**
> The `main` builds are made from the newest commit and are **not stable releases**. Expect bugs, incomplete features, and breaking changes. Use them at your own risk.

The Windows download contains two executables: run `doriax-editor.exe` to open the editor, and use `doriax-editor-cmd.exe` for scripting and CI. See Command-Line Tools (p. 167).

The Linux download is an **AppImage**: a single file to run directly, with nothing to install. From the release page it arrives without its executable bit, since no permission survives an HTTP download, so mark it executable once:

```
chmod +x Doriax-*-linux-x86_64.AppImage
```

The main-branch build is a zip instead, which does carry the bit — unzip it and run. The release also offers the same build as a plain zip of loose binaries.

> **Adding Doriax to your application menu**
> An AppImage is a single self-contained file, so nothing registers it with your desktop on its own — which is why it has no menu entry or icon until you integrate it. Tools such as Gear Lever[3] or `appimaged` read the desktop entry and icons bundled inside the AppImage and register them for you.

The macOS download is a disk image holding `Doriax.app`. Open the `.dmg` and drag the app onto the **Applications** shortcut inside it.

Beside the disk image, macOS is also offered as a plain zip of loose binaries — `doriax-editor` next to `libdoriax.dylib` and the `engine-sdk/` folder, with no `.app` bundle. It is the same build, for cases where a bundle is inconvenient (a CI checkout, a tool that expects the binary directly). A zip that came through GitHub Actions drops the executable bit, so mark it once if the binary refuses to run:

```
chmod +x doriax-editor
```

2. https://doriax.org/#download
3. https://github.com/mijorus/gearlever

macOS blocks the first launch

Doriax is signed, but not notarized by Apple — notarization requires a paid Apple Developer membership. macOS therefore reports that it cannot verify the app the first time you open it.

To allow it, open **System Settings → Privacy & Security**, scroll to the security section near the bottom, and click **Open Anyway**. You only do this once per version.

Before macOS Sequoia you could Control-click the app and choose **Open** instead. That shortcut was removed.

Downloading from a terminal skips the prompt altogether, because the quarantine flag is set by the browser rather than by macOS. This downloads the latest release:

```
curl -LO "$(curl -s https://api.github.com/repos/doriaxengine/doriax/releases/latest | grep -o 'https://[^"]*macos-universal\.dmg')"
```

Build from source

Doriax is built with **CMake**. The root project builds the desktop editor target `doriax-editor`.

Quick build

```
git clone https://github.com/doriaxengine/doriax.git
cd doriax
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

On single-config generators (Ninja, Makefiles) the executable is created under `build/`. On multi-config generators such as Visual Studio, look under the configuration subdirectory (for example `build/Release/`).

Tip

Each platform has additional dependencies and tooling. See Building from Source (p. 746) for detailed, per-platform instructions.

Repository layout

When working from source, it helps to know how the repository is organized:

Directory	Contents
`editor/`	Desktop editor, windows, tools, project generation, and export flow
`engine/`	Runtime engine, platform layers, rendering, ECS, and project templates
`shadercompiler/`	Shader compilation and translation tools
`libs/`	Bundled third-party dependencies

Next steps

Once you have the editor running, continue to Your First Project (p. 14).

Your First Project

This walkthrough uses the current Doriax editor workflow. Older Doriax/Supernova examples often start by writing a full scene in code; that still works for low-level runtime use, but day-to-day projects should start in the editor: create a project, author a scene, add entities and components, attach scripts, then run in play mode.

Create a project

Open the Doriax editor. If there is no saved project to reopen, the editor starts with a temporary project and a new 3D scene; there is no startup dialog. Choose **File → Save All**, enter a name, and select an empty directory before doing work you want to keep. See Project Workflow (p. 48) for the complete startup and save behavior.

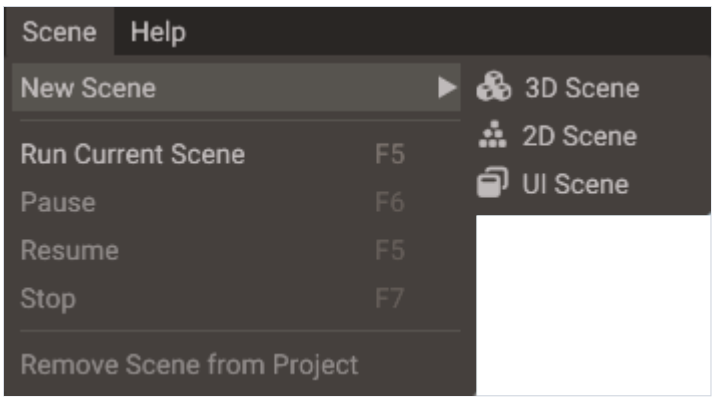
The saved project stores the editable source for your game: scenes, assets, Lua scripts, C++ scripts, bundles, and export settings.

A typical project contains:

- **Assets** - textures, models, audio, fonts, and other resources
- **Scripts** - Lua and/or C++ game code
- **Scene files** - YAML scene data edited by the editor and exported into runtime code
- **Project settings** - build targets, scene references, and export configuration

Create a scene

Use the initial 3D scene, or create a 2D, 3D, or UI scene with **Scene → New Scene** or from the Structure panel. Save the scene separately with a stable name such as `main`, `level_01`, or `menu`.



A new 3D scene starts with a **Sun** (a directional light that casts shadows) and a **Sky**. 2D and UI scenes start empty and draw through a built-in camera until you add one.

Scene type controls editor defaults such as camera setup and viewport behavior. It does not create a different runtime type; every scene is still a `Scene` that owns entities, components, and systems.

Add entities

Open the Structure panel's create menu: click **+** at the top of the panel and point at **Create entity**, or right-click empty space in the panel. The most important first distinction is:

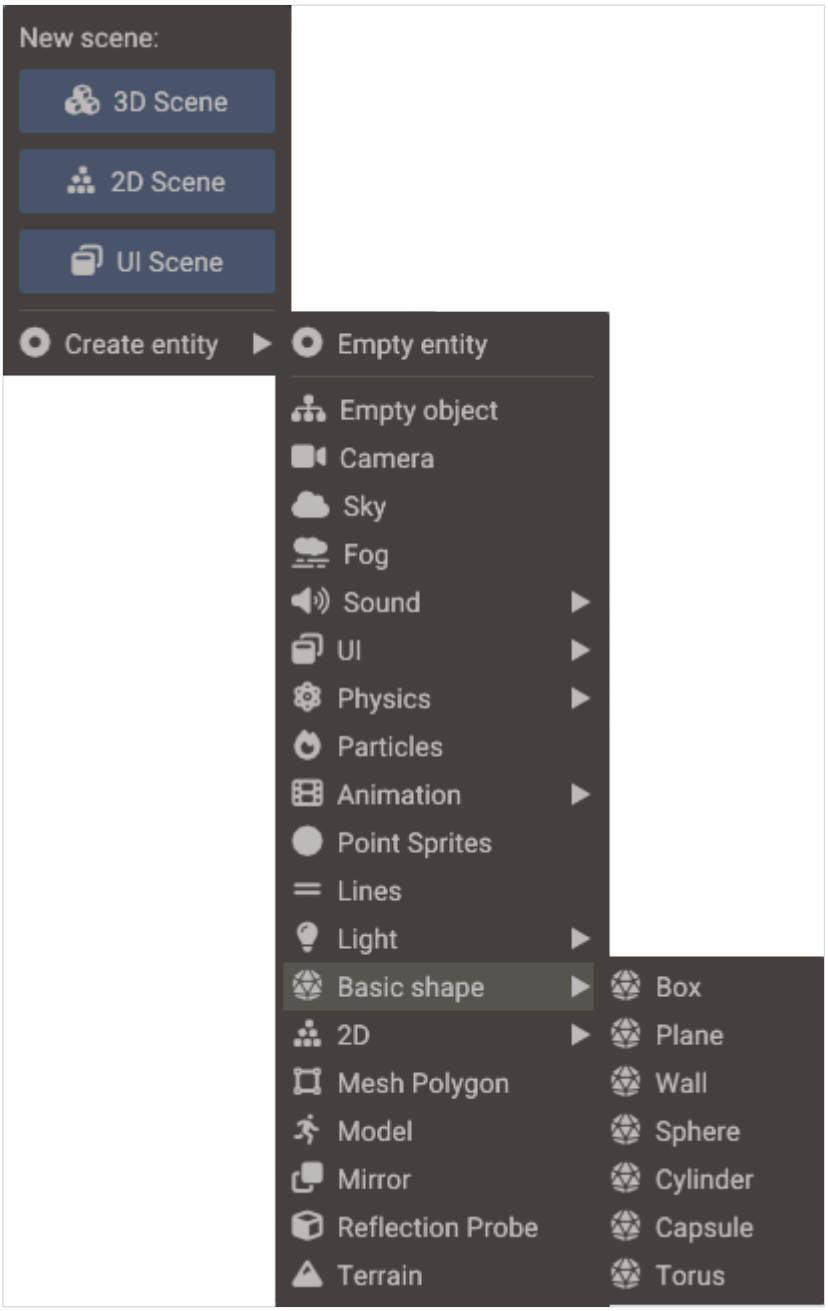
Menu item	What it creates	Where it appears
Empty entity	An entity ID with no `Transform` component	The non-hierarchical area of Structure

Menu item	What it creates	Where it appears
Empty object	An entity with a `Transform` component	The hierarchy area of Structure

Everything in the scene is an entity. An entity is only an ID; it owns no position, mesh, physics body, script, or audio by itself. Components give the entity data and behavior. `Transform` is the component that makes an entity spatial and allows it to participate in the hierarchy.

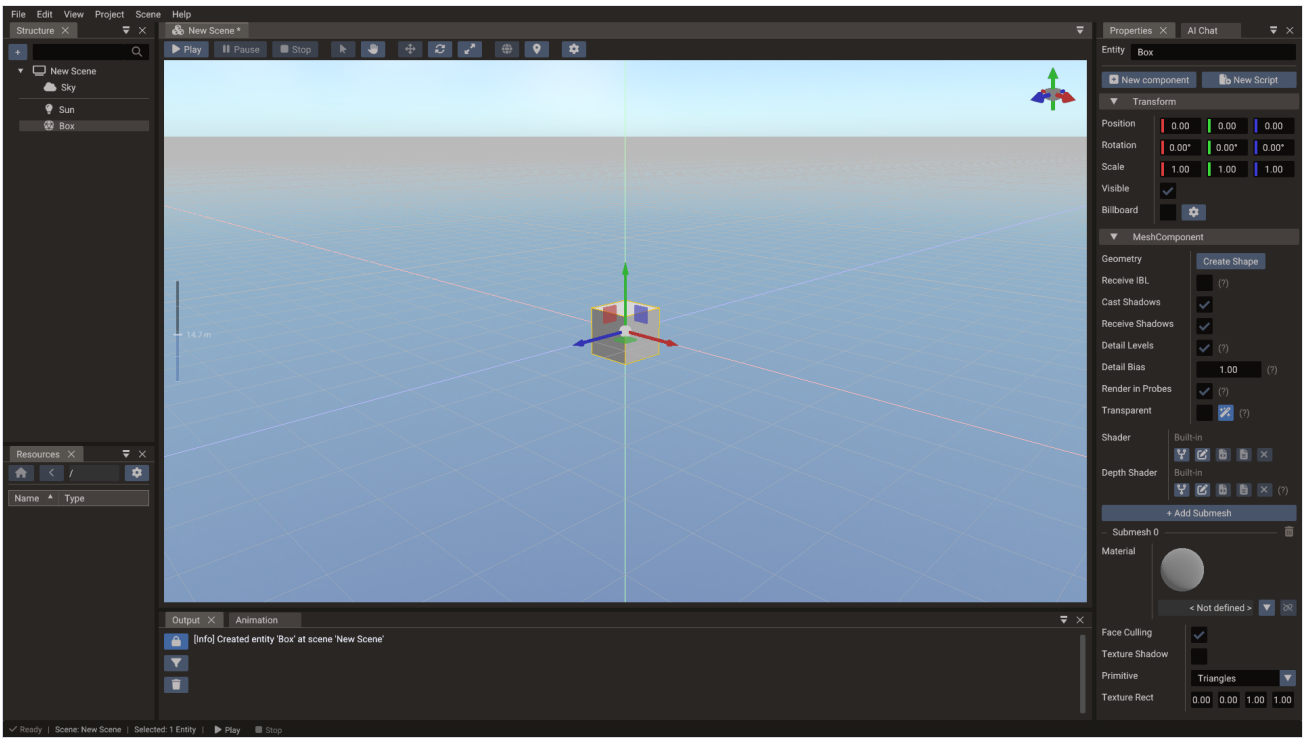
For a first visible object, choose one of these:

- In a 2D scene, create **2D > Sprite**.
- In a 3D scene, create **Basic shape > Box** or **Model**.
- In a UI scene, create **UI > Button**, **Text**, or **Image**.



Edit components

Select the entity and use the Properties window. You will see the components attached to that entity.



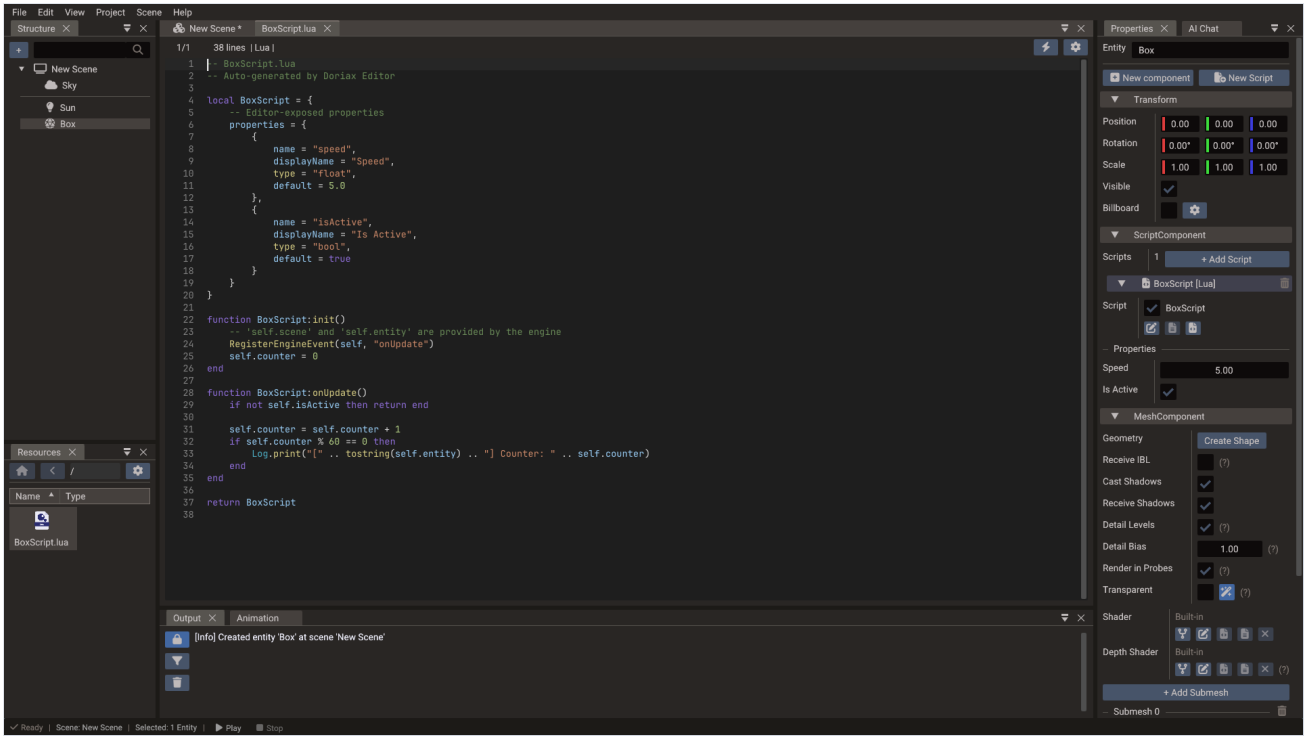
A sprite, for example, is not a special kind of entity; it is an entity with components such as `Transform`, `MeshComponent`, and `SpriteComponent`.

Common first edits:

Component	Edit
`Transform`	Position, rotation, scale, visibility, parent hierarchy
`SpriteComponent`	Width, height, frame, pivot, texture region
`MeshComponent`	Material, color, shadows, transparency, primitive data
`CameraComponent`	Projection, clipping, target, render-to-texture
`ScriptComponent`	Lua or C++ scripts attached to the entity

Add a script

Select the entity and click **New Script** at the top of the Properties window. Choose **Lua Script** — the fastest to iterate on — enter a name, and click **Create**. The editor writes the `.lua` file, adds a `ScriptComponent` to the entity, and links the script to it. Open the file with the source button on the script entry, or double-click it in the Resources Browser.



The generated Lua script returns a table. The engine creates an instance, injects `self.scene` and `self.entity`, then calls `init()` when the scene starts.

```lua
local PlayerController = {
    properties = {
        { name = "speed", displayName = "Speed", type = "float", default = 5.0 },
        { name = "isActive", displayName = "Is Active", type = "bool", default = true }
    }
}

function PlayerController:init()
    RegisterEngineEvent(self, "onUpdate")
end

function PlayerController:onUpdate()
    if not self.isActive then return end
    if Input.isKeyPressed(Input.KEY_RIGHT) then
        local object = Object(self.scene, self.entity)
        object.position = object.position + Vector3(self.speed * Engine.deltatime, 0, 0)
    end
end

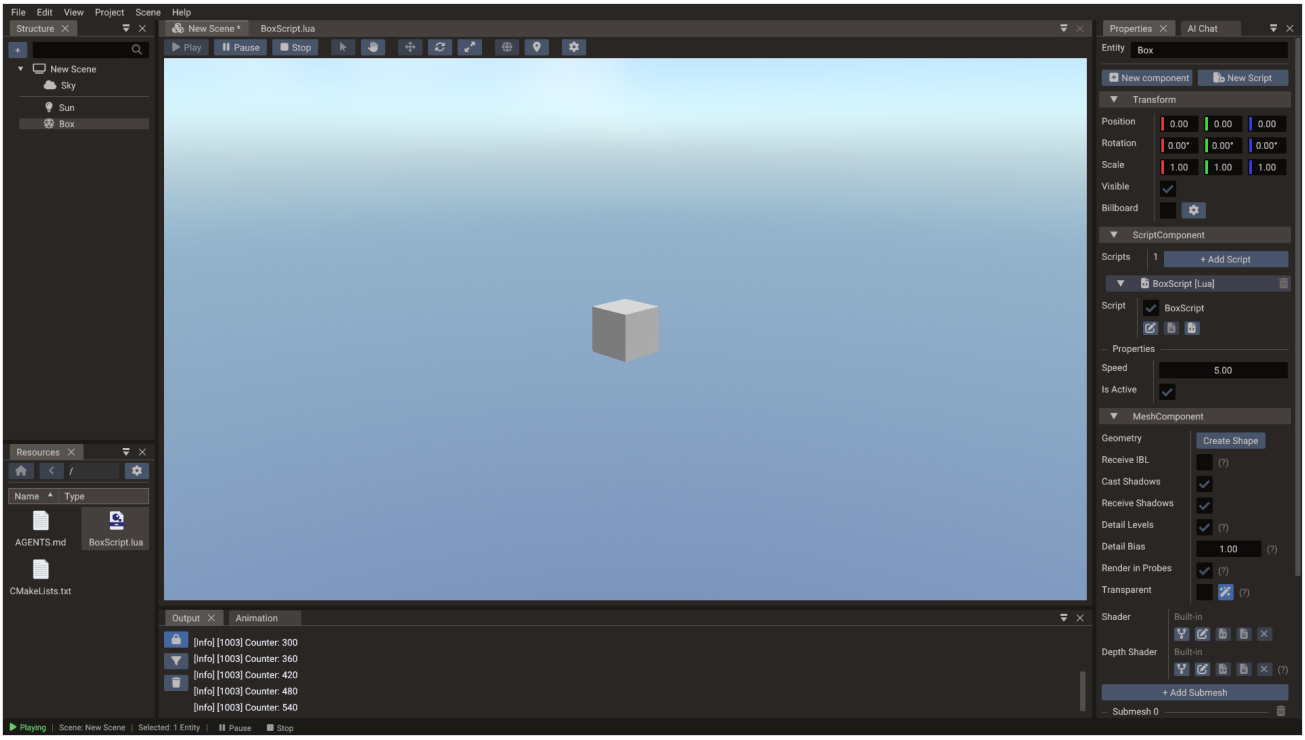
return PlayerController
```

For C++ scripts, the editor can generate either a subclass script or a `ScriptBase` script. C++ scripts use `DPROPERTY` for values that should appear in the Properties window and `REGISTER_ENGINE_EVENT` for callbacks.

> **Mixing Lua and C++ entry points**
>
> Editor-authored projects should not rely on both `main.lua` and C++ `DORIAX_INIT` code to set the startup scene. If both paths call `Engine.setScene()` / `Engine::setScene()`, one startup path can replace the other. Use `NO_CPP_INIT` or `NO_LUA_INIT` when you intentionally want only one entry point.

Run the project

Press Play in the editor. The editor saves a play-mode snapshot, starts the runtime scene, initializes scripts, and restores editable scene state when you stop.



If something does not appear, check these in order:

- The intended scene is selected or configured as the start scene.
- The entity has the required components.
- Visible/spatial entities have a `Transform`.
- The active camera can see the entity.
- The script is enabled in `ScriptComponent`.
- The output panel has no Lua or C++ errors.

Next steps

The Editor (p. 19)
Tour the editor's panels and tools.

Scenes & Entities (p. 182)
Learn how scenes and entities are structured.

First 2D Scene (p. 26)
Follow a more complete 2D tutorial.

Creating Scripts (p. 199)
Create Lua and C++ behavior scripts for entities.

The Editor

Doriax ships with a complete visual editor built for creators. Design scenes, write code, animate characters, manage resources, and test your game from one unified environment.

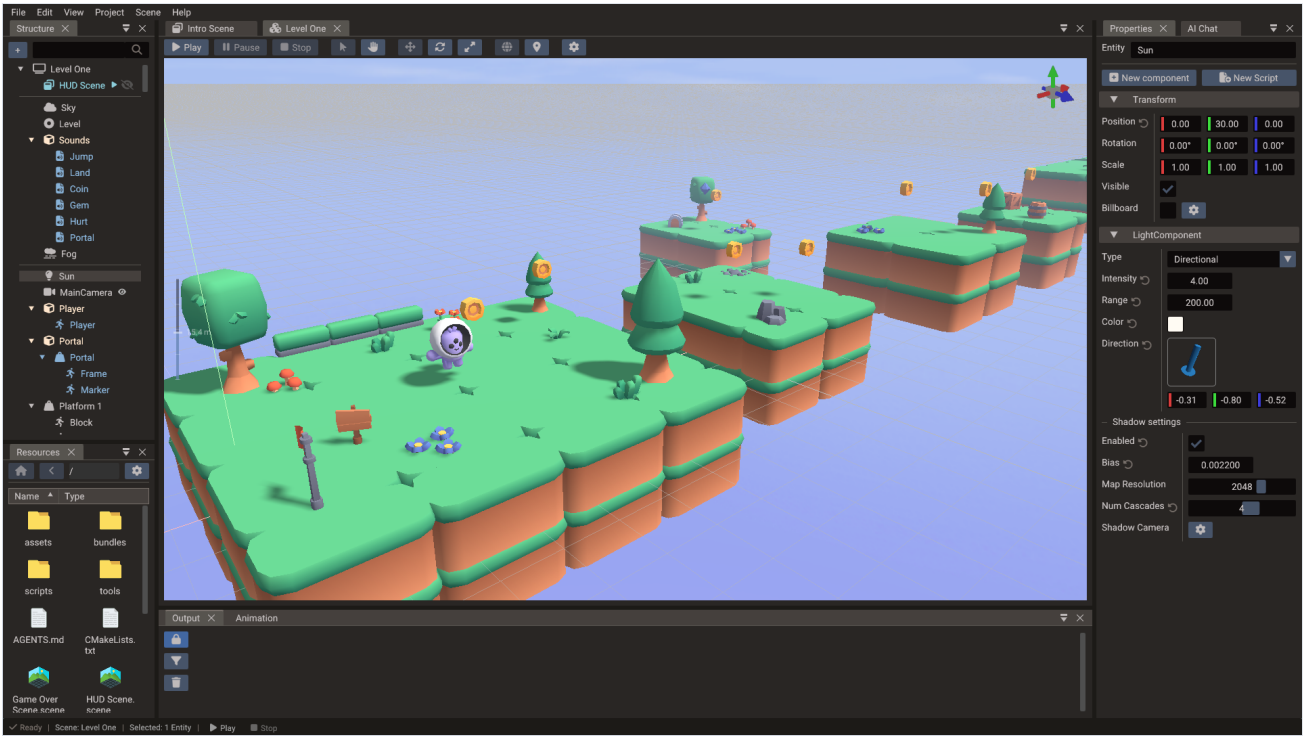


Core panels

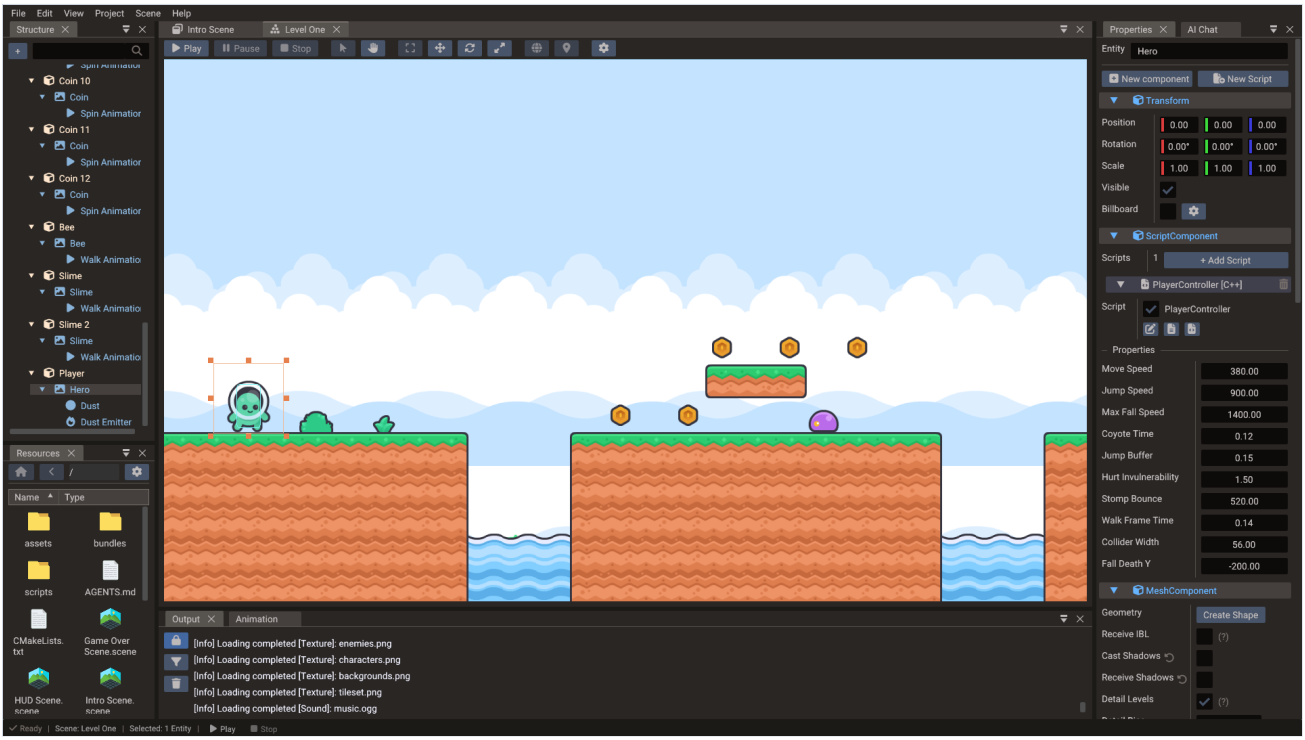
Panel	Purpose
Scene view	Viewport for editing 2D and 3D scenes with gizmos and cameras
Structure	The tree of scenes and entities in the current scene
Properties	View and edit the components and properties of the selected entity
Resources	Manage textures, models, audio, and other project assets
Code editor	Integrated editor for Lua and C++ scripts
Animation	Timeline, keyframes, sprite animation, skeletal animation, and morph tracks
Output	Build logs, export messages, and diagnostics

3D scenes

Edit 3D scenes with cameras, lighting, models, and play mode. Position entities with gizmos, set up PBR materials, and preview shadows, fog, and the sky system directly in the viewport.

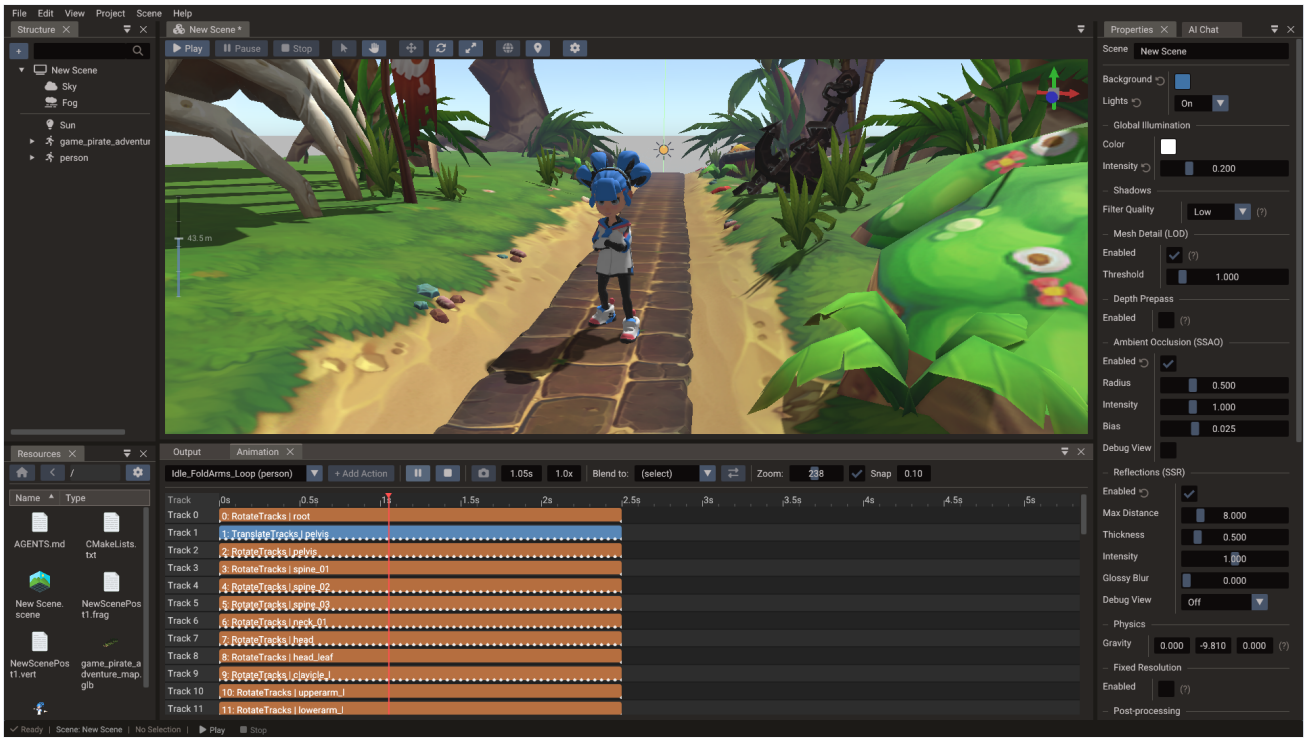


2D & tilemaps



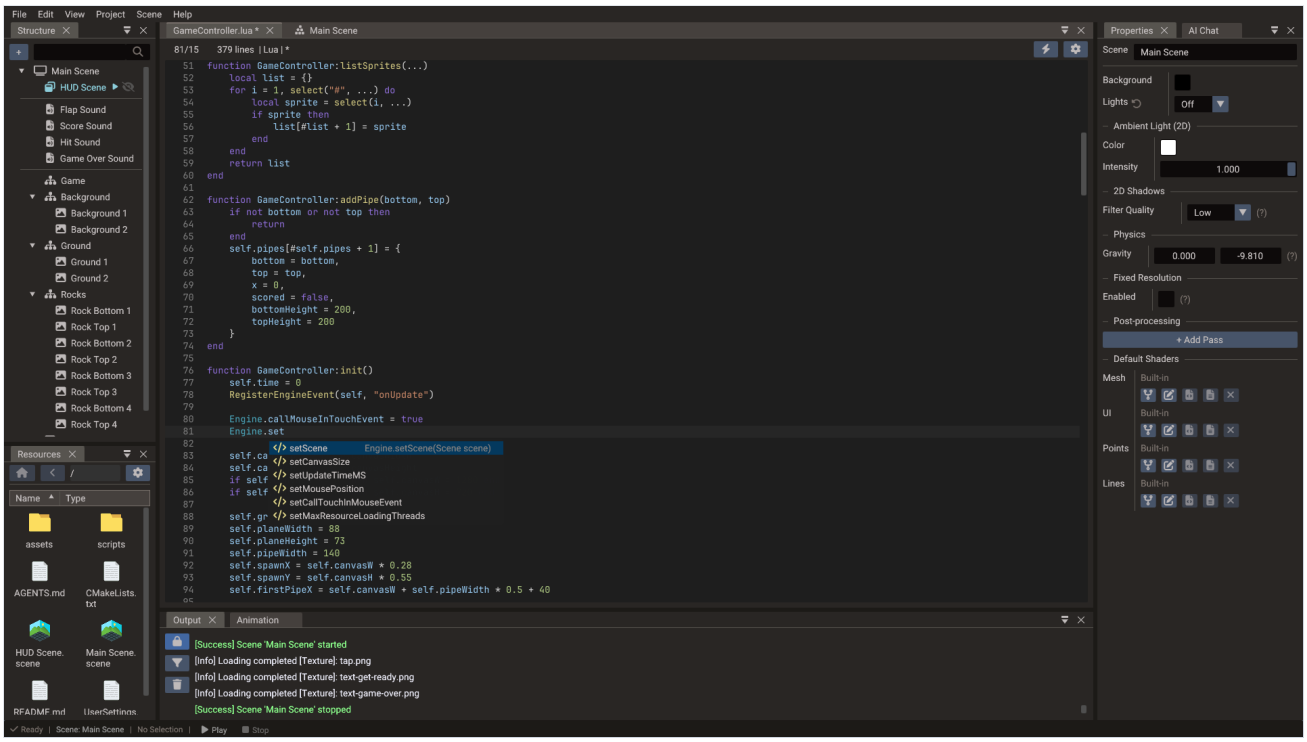
Build 2D scenes with sprites and tilemaps. The **sprite slicer** and **tileset slicer** tools let you cut sprite sheets and tilesets into usable frames and tiles.

Animation



Animate characters and objects with the timeline editor, and work with skeletons using the bone tools for skeletal animation.

Code editor



Write Lua and C++ scripts in the integrated code editor without leaving the environment. Lua iterates quickly at runtime, while C++ is compiled at build time for native performance. Its event menu (p. 137) adds engine, input, component, and physics event handlers to a script for you.

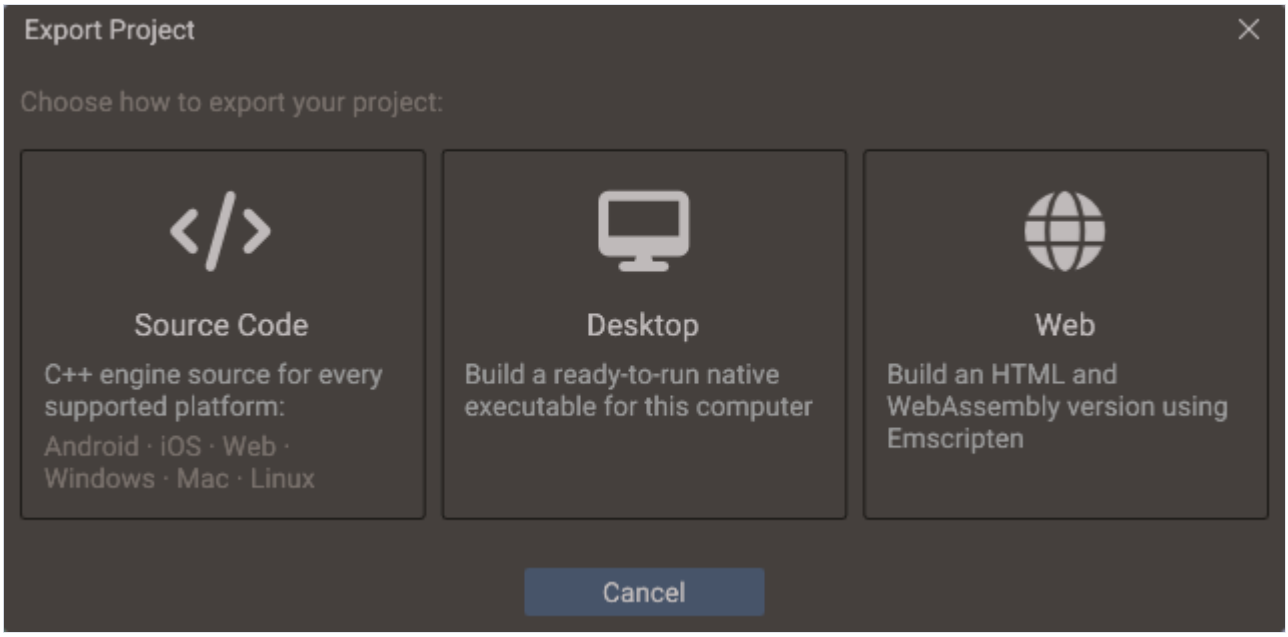
Play mode

Press **Play** to run your game inside the editor. Test gameplay, UI overlays, and physics interactively, then return to editing.



Export pipeline

The editor includes a shader-aware export pipeline that prepares scenes, assets, scripts, engine files, and compiled shaders for your target platform — and can compile ready-to-run desktop and web builds directly. See the Export Window (p. 155) for the export modes and output options.



> **Editor coverage**
> Terrain sculpting, painting, foliage and prop placement have a dedicated **Terrain Editor** window, particle entities can be created from the Structure panel, and built-in shaders can be forked and edited in place (see Custom Shaders (p. 143)). Audio tooling is not fully integrated into the editor yet — it is available at the engine/runtime level.

More editor documentation

The detailed editor manual is split into focused pages:

- Project Workflow (p. 48)
- Project Settings (p. 55)
- Editor Settings (p. 68)
- Scene View (p. 81)
- Properties & Components (p. 90)
- Resources Browser (p. 106)
- Terrain Editor (p. 123)
- Animation Timeline (p. 129)
- Code Editor (p. 136)
- Export Window (p. 155)

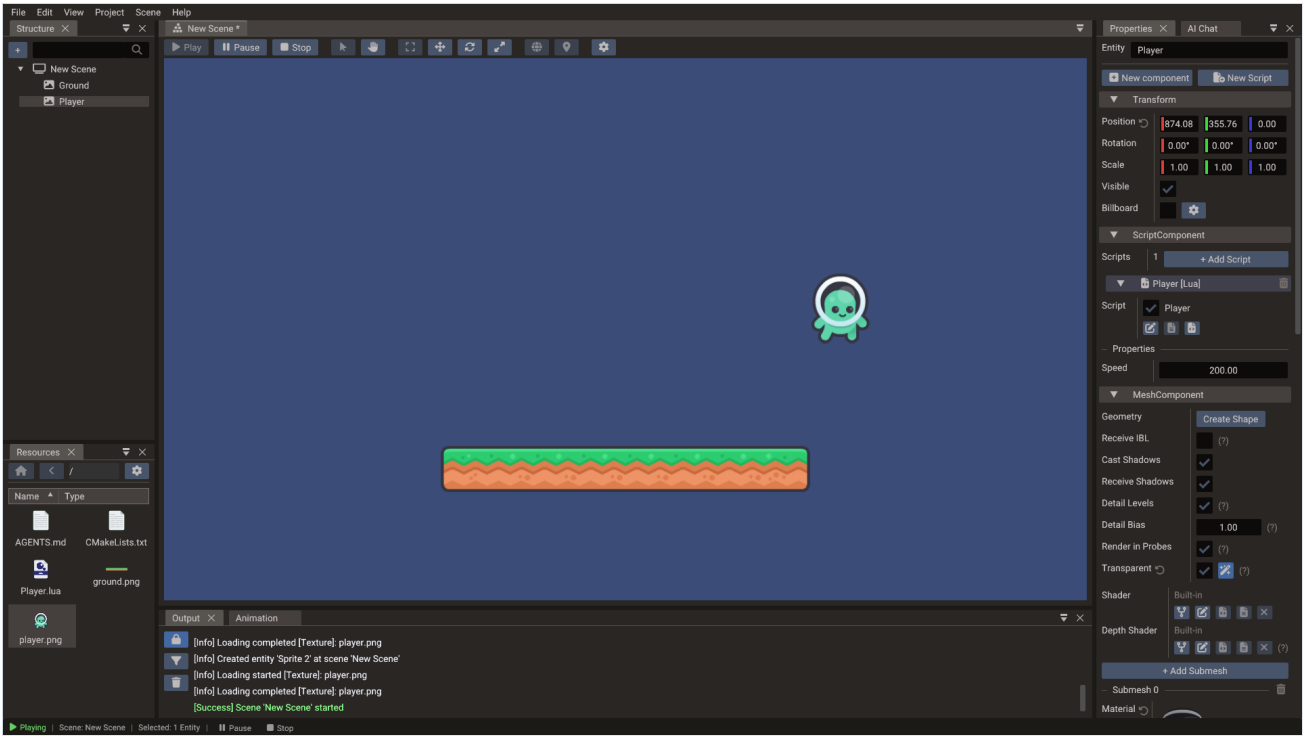
PART 2

Tutorials

First 2D Scene 26
First 3D Scene 31
First UI Scene 35
Exporting a Project 40

First 2D Scene

This tutorial builds a complete small 2D scene from scratch and introduces the core workflow you will reuse in larger projects: create a scene, add entities, attach components, write a script, and test in play mode.



What you will build

By the end of this tutorial you will have:

- A 2D scene with a visible sprite
- A canvas set up for your design resolution
- A Lua script that moves the sprite with keyboard input
- A `Body2D` physics body ready for collision

1. Create the project and scene

1. Open the editor and choose **File → New Project** to start with a fresh temporary project.
2. Choose **Scene → New Scene → 2D Scene**.
3. Choose **File → Save All**. The editor asks where each unsaved scene goes — name the 2D one `main` — and then where the project itself lives: enter a project name and select an empty directory.
4. The initial 3D scene was never saved, so the new 2D scene replaces it — there is nothing to remove.

2. Add a sprite entity

1. Copy a PNG into the project folder, or drag one from your file manager into the **Resources Browser**.
2. Drag the PNG from the Resources Browser into the viewport. The editor creates a **Sprite** entity at the drop point, sized to the image.
3. Press **F2** and rename it `Player`.

You can also create the sprite first with **+ → Create entity → 2D → Sprite** and then drop the PNG onto it in the viewport to give it a texture.

3. Configure the camera

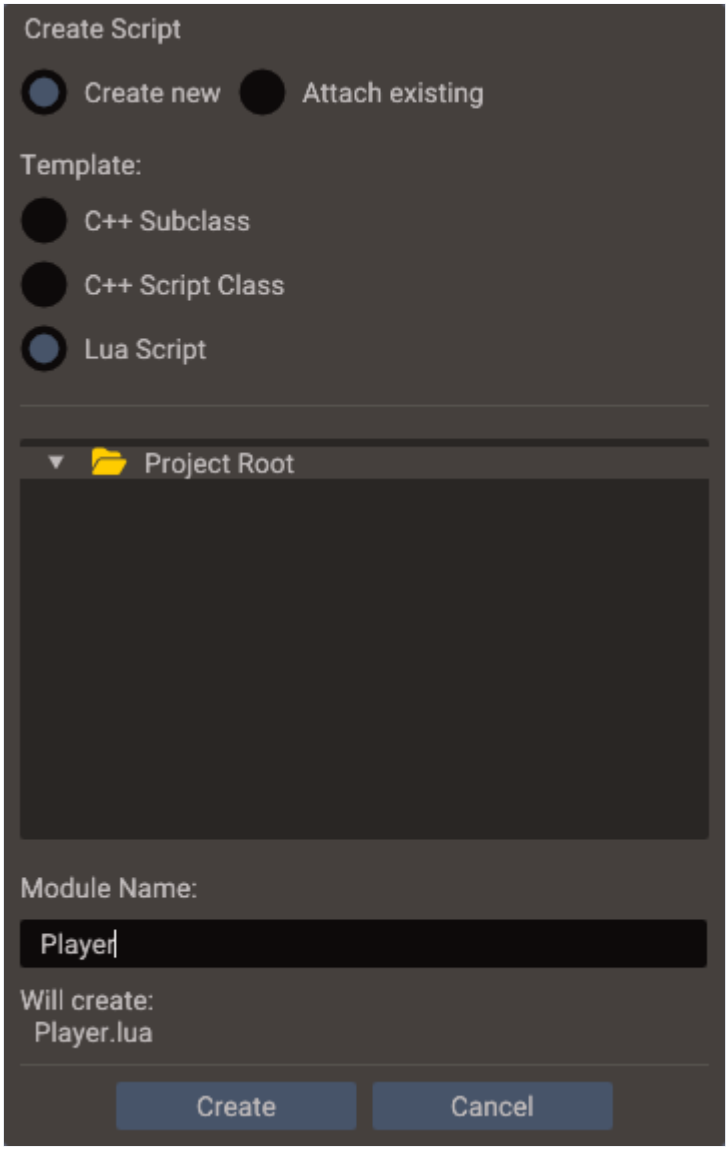
A 2D scene does not need a camera entity: it draws through a built-in orthographic camera that frames the canvas. The canvas belongs to the project — open **Project → Project Settings → Canvas** and check that:

- **Canvas Width / Height** match your design resolution, for example `1280 × 720`.
- **Scaling Mode** is **Letterbox** or **Fit Width**.

Add a **Camera** entity (**+ → Create entity → Camera**) only when the view has to move, for example to follow the player. The first camera you add becomes the scene's main camera.

4. Add a script

1. Select **Player** and click **New Script** at the top of the Properties window.
2. Choose **Lua Script**, enter `Player` as the module name, and click **Create**. The editor writes `Player.lua` into the folder selected in the dialog, adds a `ScriptComponent` to the entity, and links the script to it.



3. Open the file with the source button on the script entry, or double-click it in the Resources Browser, and replace its contents with the following movement code:

```
local Player = {
    properties = {
        { name = "speed", displayName = "Speed", type = "float", default = 200.0 }
```

```
    }
}

function Player:init()
    RegisterEngineEvent(self, "onUpdate")
end

function Player:onUpdate()
    local obj = Object(self.scene, self.entity)
    local dt = Engine.deltatime
    local dx, dy = 0, 0

    if Input.isKeyPressed(Input.KEY_LEFT)  then dx = dx - self.speed * dt end
    if Input.isKeyPressed(Input.KEY_RIGHT) then dx = dx + self.speed * dt end
    if Input.isKeyPressed(Input.KEY_UP)    then dy = dy + self.speed * dt end
    if Input.isKeyPressed(Input.KEY_DOWN)  then dy = dy - self.speed * dt end

    -- Object.position is returned by value; assign the updated vector back.
    obj.position = obj.position + Vector3(dx, dy, 0)
end

return Player
```

4. Save the file (**Ctrl+S**). The **Speed** property appears on the script entry in the Properties window.

5. Add physics (optional)

To make the sprite collide with walls:

1. Select the **Player** entity.
2. Click **New component** and choose **2D Physics Body** (type `body` in the search field to find it).
3. Set the **Body Type** to **Kinematic** (for a player controlled by script) or **Dynamic** (for full physics simulation).
4. Leave the shape type on **Polygon** and click **Add Shape**. The polygon starts as a box the size of the sprite. If the sprite needs a tighter outline, edit its vertices in the shape's properties or drag their handles in the scene view.



Give walls a 2D Physics Body with **Body Type** set to **Static** and a polygon shape to block the player's movement.

6. Run and test

Press **Play** in the toolbar. The sprite should be visible and respond to arrow key input.

If the sprite is not visible, check in order:

☐ The scene is set as active in project settings.

☐ The entity has a Sprite component with a valid texture path.

☐ If you added a Camera entity, it is orthographic and frames the sprite.

☐ The canvas size and scaling mode are sensible for the editor window size.

☐ The script entry is enabled in the ScriptComponent.

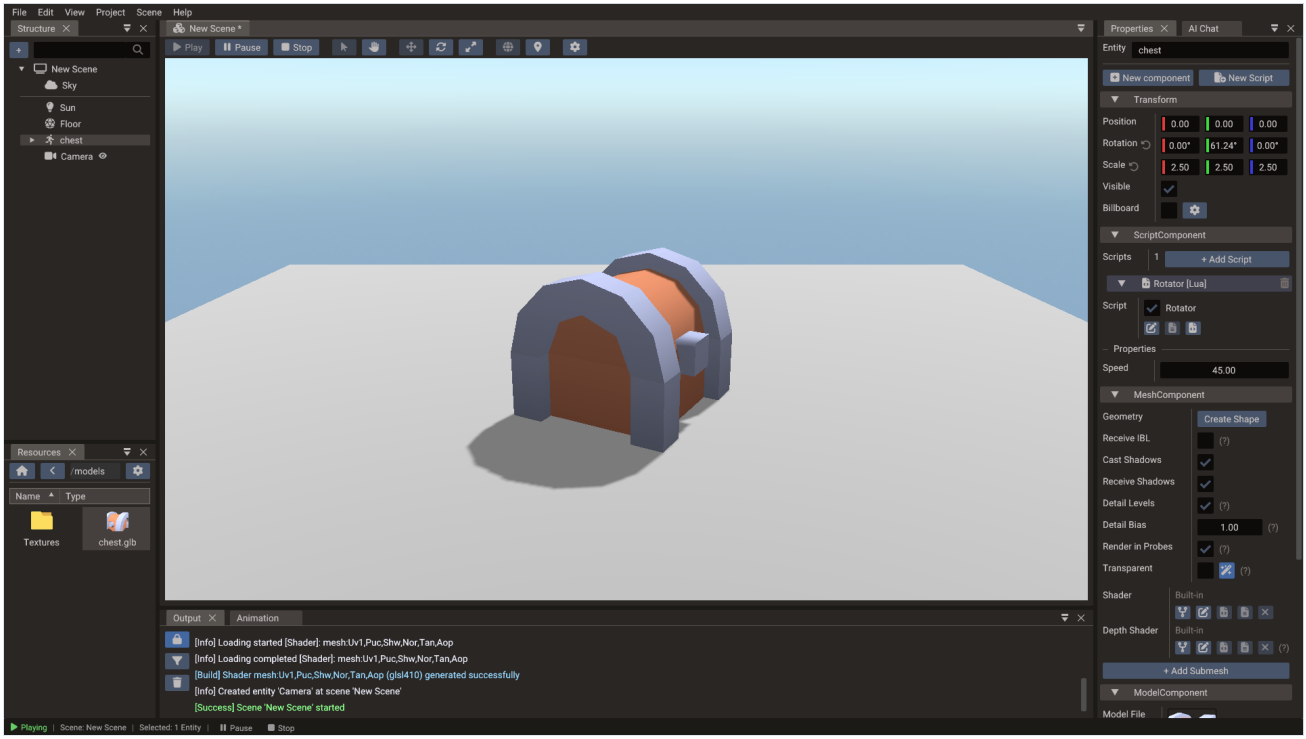
Press **Stop** when done — the editor restores the scene to its pre-play state.

7. Next steps

- Add more sprites and create a tilemap background with the Tileset Slicer (p. 119).
- Animate the player with the Sprite Slicer (p. 116) and `SpriteAnimation`.
- Add a HUD overlay using a UI scene (p. 35).
- Continue with 2D Graphics (p. 243), Input (p. 229), and Physics (p. 315).

First 3D Scene

This tutorial creates a complete small 3D scene and introduces the essential pieces of a Doriax 3D level: scene setup, camera, directional light, a 3D model, PBR materials, and optional collision.



What you will build

By the end of this tutorial you will have:

- A 3D scene with a model visible in the game camera
- Directional lighting with optional shadows
- A physics floor and optional collision on the model
- A script that rotates the object each frame

1. Create the project and scene

1. Open the editor and choose **File → New Project** to start with a fresh temporary project and its initial 3D scene.
2. Choose **File → Save All**. The editor asks where the new scene goes — name it `main` — and then where the project itself lives: enter a project name and select an empty directory.

2. Set up the camera

A new 3D scene has no camera entity; it renders through a built-in camera until you add one. In the **Structure panel**, choose **+ → Create entity → Camera**. The first camera you add becomes the scene's main camera. Then:

1. Keep **Type** on **Perspective**.
2. Position it at approximately `(0, 3, 8)`. To keep it facing the origin, enable **Use Target** and set **Target** to `(0, 0, 0)`.
3. Set **Near** to `0.1` and **Far** to `200` for a typical scene scale.

Click **View** on the **Preview** row of the CameraComponent to look through the camera, and **Exit** to return to the editor view. Press **F** with the camera selected to frame it in the editor view.

3. Adjust the directional light

A new 3D scene already has a **Sun**: a directional light with shadows enabled. Other lights come from **+ → Create entity → Light → Directional**, **Point**, or **Spot**.

1. Select **Sun**.
2. Change its **Direction** so shadows fall at a diagonal; it starts at `(-0.2, -0.5, 0.3)`.
3. Set **Intensity** to `2.0` and choose a warm white **Color** (`1.0, 0.95, 0.8`).
4. Keep **Shadows → Enabled** on so the model casts a shadow onto the floor.

The scene's **Global Illumination** (select nothing to see the scene settings) starts at an intensity of `0.2`, which keeps shadowed areas from going completely black.

4. Import and place a model

1. Copy a GLTF file into your project's `assets/` folder (or drag one from your OS file manager into the Resources Browser).
2. In the **Resources Browser**, drag the GLTF file into the scene view — a Model entity is created automatically.
3. Use the **Translate gizmo** (W) to move the model to the origin.
4. If the model appears very large or very small, adjust its **Scale** in the Properties window. GLTF files exported from Blender with default settings use meters — scale down if your scene uses smaller units.

5. Tune the materials

If the model carries its own GLTF materials, they appear automatically. To adjust them:

1. Select the model — or, for a model with several meshes, one of its child mesh entities in the Structure panel.
2. In the Properties window, expand the **Submesh** material section.
3. Adjust **Roughness** (lower = shinier), **Metallic** (0 = plastic, 1 = metal), and check that the **Base Texture** path is correct.

To reuse the same material on several objects later, drag the **material preview** from Properties into the Resources Browser to create a `.material` file, then drag that file onto other meshes. See Material files (p. 108).

Tips:

- Roughness controls how wide the specular highlight is — 0.1 looks mirror-smooth, 0.9 looks matte.
- Metallic should only be 1.0 for actual metals; leave it at 0.0 for wood, stone, fabric, and plastic.
- Preview under both bright and dim lighting to catch texture issues early.

6. Add a script — rotating object

1. Select the model and click **New Script** at the top of the Properties window.
2. Choose **Lua Script**, enter `Rotator` as the module name, and click **Create**. The editor adds a `ScriptComponent` to the model and links the new script to it.
3. Open the file with the source button on the script entry and replace its contents with:

```
local Rotator = {
    properties = {
        { name = "speed", displayName = "Speed", type = "float", default = 45.0 }
```

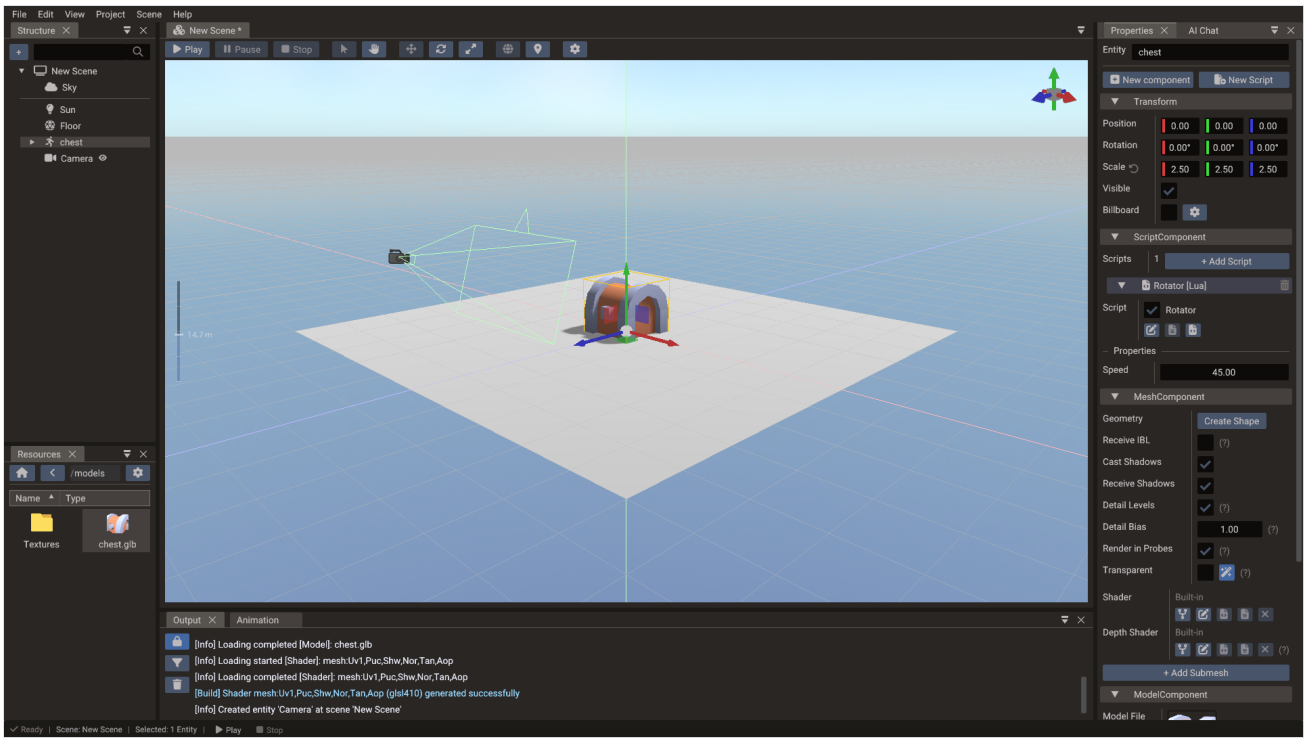
```
    }
}

function Rotator:init()
    RegisterEngineEvent(self, "onUpdate")
    self.angle = 0
end

function Rotator:onUpdate()
    local obj = Object(self.scene, self.entity)
    self.angle = self.angle + self.speed * Engine.deltatime
    -- Quaternion(x, y, z) builds a rotation from Euler angles in ZYX order
    -- (in degrees, since Engine.useDegrees is true by default).
    obj.rotation = Quaternion(0, self.angle, 0)
end

return Rotator
```

4. Save the file (**Ctrl+S**). The **Speed** property appears on the script entry.



7. Add a floor with collision

1. Choose **+ → Create entity → Basic shape → Plane** for the floor.
2. Position it at `(0, -0.5, 0)` and scale it wide.
3. Click **New component** and choose **3D Physics Body**.
4. Choose **Box** in the shape list and click **Add Shape**, then set the shape's **Width**, **Height**, and **Depth** to match the floor.
5. Keep **Body Type** on **Static**.

Optionally give the model a 3D Physics Body with a **Convex Hull** or **Box** shape and set it to **Dynamic** to see it react to gravity.

8. Run the scene

Press **Play**. The model should rotate and be illuminated by the directional light.

If the scene appears completely black:

☐ The light is enabled and has positive intensity.

☐ The scene's Light State is `ON` or `AUTO`.

☐ The model has materials with valid albedo textures or colors.

☐ The camera's Far plane is greater than the distance to the model.

☐ The camera is the active camera in the scene's Camera field.

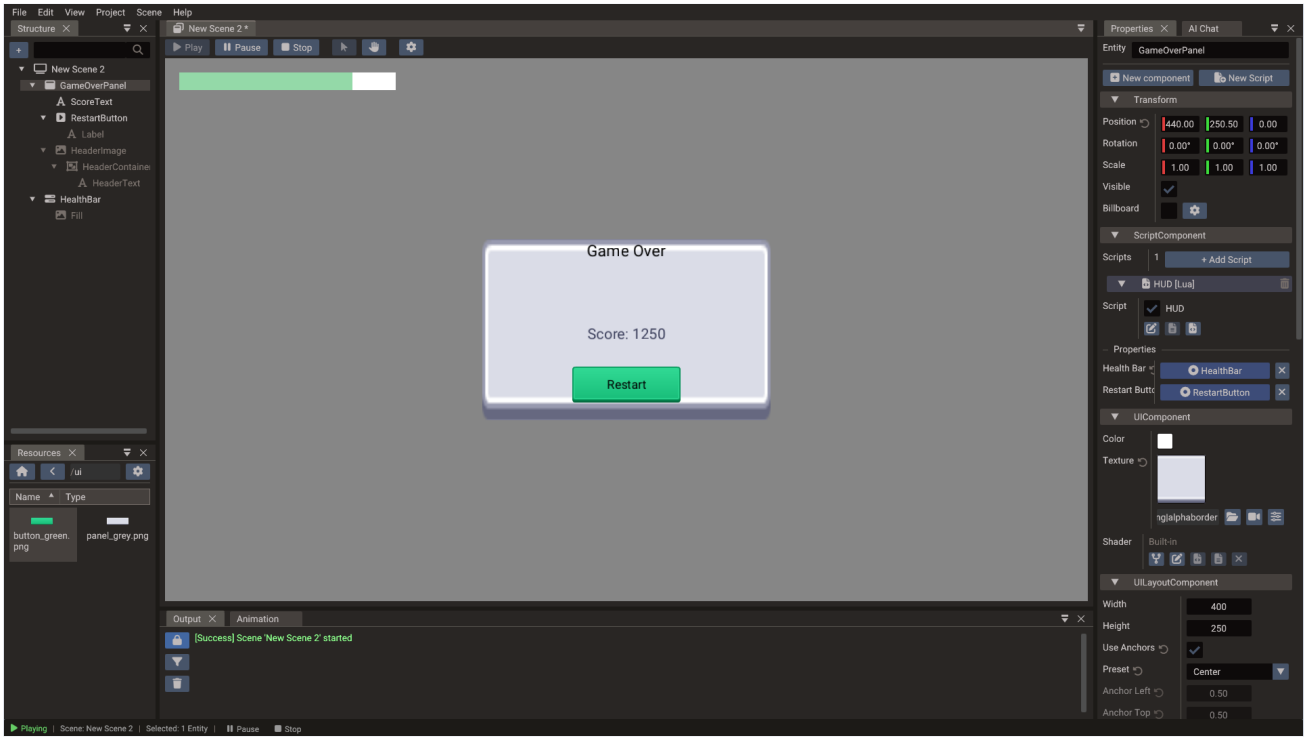
Press **Stop** when done.

9. Next steps

- Add skeletal animation using the Animation Timeline (p. 129).
- Add a **Sky** entity, enable **Receive IBL** on reflective meshes, and optionally hide the sky with **Visible** while keeping reflections: see Rendering Pipeline — IBL (p. 263).
- Add fog for depth: see Rendering Pipeline (p. 258).
- Add a UI overlay HUD: see First UI Scene (p. 35).
- Continue with 3D Graphics (p. 248) and Physics (p. 315).

First UI Scene

This tutorial builds a **UI scene** — a separate scene dedicated to screen-space widgets — and shows how to layer it on top of a gameplay scene. UI scenes are the recommended approach for HUDs, menus, and overlays in Doriax.



What you will build

By the end of this tutorial you will have:

- A UI scene with a centered panel, a text label, and a Button
- A `Progressbar` acting as a health bar in the top-left corner
- The UI scene loaded on top of a gameplay scene
- A script that updates the health bar value from gameplay logic

1. Create a UI scene

1. Open the editor and open or create a project that already has a gameplay scene (`main` or `level_01`).
2. Choose **Scene → New Scene → UI Scene**.
3. Save it as `scenes/hud.scene`.

A new UI scene starts empty and draws through a built-in UI camera that frames the project's canvas.

2. Set the canvas size

The canvas is shared by every scene in the project. Open **Project → Project Settings → Canvas**, set **Canvas Width / Height** to your design resolution — for example `1920 × 1080` — and set **Scaling Mode** to **Letterbox** to keep the aspect ratio across different screen sizes.

3. Add a health bar

1. In the **Structure panel**, choose **+ → Create entity → UI → Progressbar**. It comes with a **Fill** child that draws the filled part of the bar.
2. In the **Properties window**:
3. Set **Width** to `300` and **Height** to `24`.
4. Enable **Use Anchors** and set **Preset** to **Top Left**.
5. Set **Position Offset** to `20, 20` to inset the bar from the corner.
6. Set the initial **Value** to `0.8`. Progressbar values are normalized from `0.0` (empty) to `1.0` (full).
7. Select the **Fill** child to give the bar a texture or a **Color** (e.g. green for health).
8. The health bar appears in the top-left corner of the canvas.

4. Add a center panel with a button

For a "Game Over" overlay:

1. Choose **+ → Create entity → UI → Panel**.
2. Set **Width** to `400` and **Height** to `250`, enable **Use Anchors**, and set **Preset** to **Center**.
3. Drop an image from the Resources Browser onto the panel in the viewport to give it a background texture, or leave it as a solid color.

The panel comes with a title bar. Set its text:

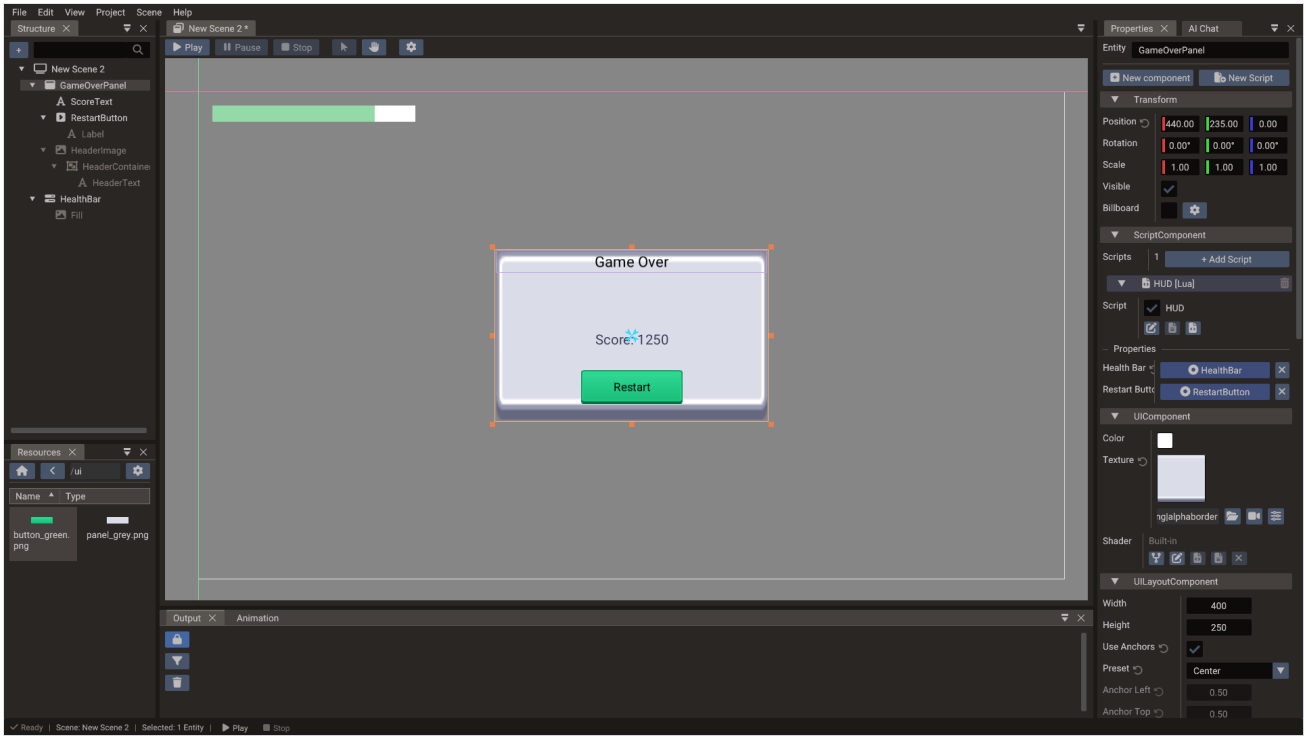
1. Expand the panel in the Structure panel and select **HeaderText** (under **HeaderImage → HeaderContainer**).
2. Set **Text** to `Game Over` and **FontSize** to `26`.

Add a label inside the panel:

1. Right-click the panel and choose **Create child → UI → Text**.
2. Enable **Use Anchors** and set **Preset** to **Center**.
3. Set **Text** to `Score: 1250`.

Add a restart button:

1. Right-click the panel in the Structure panel and choose **Create child → UI → Button**.
2. Enable **Use Anchors**, set **Preset** to **Center Bottom**, and set **Position Offset** to `0, -24` to lift it off the edge.
3. Select the button's **Label** child and set its **Text** to `Restart`.
4. The HUD script in the next step exposes a `restartButton` property for this button.



5. Write a HUD script

Create a Lua script `scripts/HUD.lua`:

```lua
-- In Lua, editor-exposed fields are declared in a `properties` table
-- (DPROPERTY is a C++-only macro).
local HUD = {
    properties = {
        { name = "healthBar", displayName = "Health Bar", type = "Progressbar" },
        { name = "restartButton", displayName = "Restart Button", type = "Button" }
    }
}

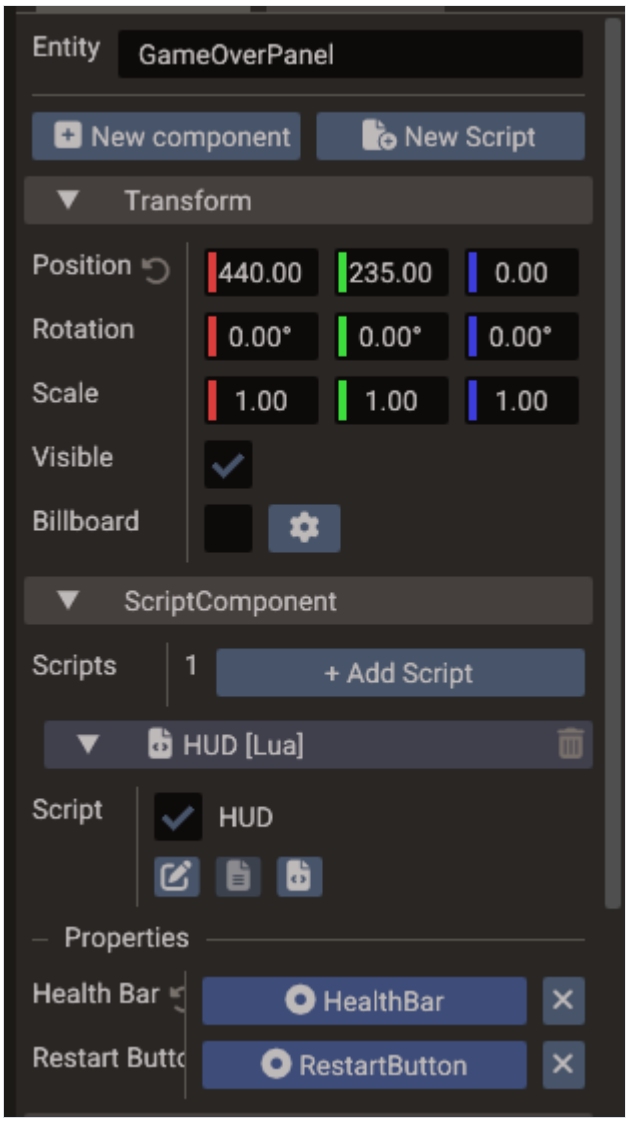
function HUD:init()
    RegisterEngineEvent(self, "onUpdate")
    if self.restartButton then
        local button = self.restartButton:getButtonComponent()
        RegisterEvent(self, button.onPress, "onRestart")
    end
end

function HUD:onUpdate()
    -- Read health from a shared value or an event
    -- For demonstration, GameState.health is a percentage from 0 to 100
    if GameState and GameState.health and self.healthBar then
        self.healthBar.value = GameState.health / 100
    end
end

function HUD:onRestart()
    SceneManager.loadScene("Main")
end

return HUD
```

Select the panel, click **New Script**, and create a Lua script named `HUD` — or, if you wrote the file already, choose **Attach existing** in the same dialog. Then drag the health bar and the button from the Structure panel onto the **Health Bar** and **Restart Button** fields of the script entry.



6. Load the UI scene as a layer

In the editor, the simplest way is to make the HUD a **child scene** of the gameplay scene: select the gameplay scene in the Structure panel (p. 73), right-click the scene root, and choose **Add child scene → HUD**. Leave **Start active** on so the HUD loads with the level. The exporter then loads both together when you call `SceneManager.loadScene("Main")`.

If you are wiring scenes up by hand instead, register the gameplay scene and its HUD as a **single stack** — one factory that sets the main scene *and* adds the HUD layer — so a single `loadScene` brings up both:

Lua

```lua
SceneManager.registerScene(1, "Main", function()
    Engine.setScene(gameplayScene)
    Engine.addSceneLayer(hudScene)   -- HUD layer on top
end)

SceneManager.loadScene("Main")
```

C++

```
SceneManager::registerScene(1, "Main", []() {
    Engine::setScene(&gameplayScene);
    Engine::addSceneLayer(&hudScene);   // HUD layer on top
});

SceneManager::loadScene("Main");
```

> **Note**
> Don't call `loadScene` once per layer — each call to `loadScene` clears every scene first. Add the HUD inside the same stack (as above, or as a start-active child scene) so it survives the load. The UI scene then renders on top of the gameplay scene automatically.

7. Run and test

Press **Play**. You should see:

- The gameplay scene rendering as normal.
- The health bar in the top-left corner.
- The "Game Over" panel centered on screen.
- The "Restart" button responding to clicks.

If the UI is not visible:

☐ The HUD scene is registered and loaded as an **additive** layer with `addSceneLayer`, not `setScene`.

☐ The canvas size and scaling mode are set correctly.

☐ The UI scene camera is orthographic.

☐ UI events are enabled on the scene (the scene's `enableUIEvents` setting).

8. Next steps

- Add a pause menu to a separate UI scene and toggle it with **Escape**.
- Use `Container` with `VERTICAL` layout for a dynamic item list.
- Use `TextEdit` for a name-entry field in a high-score screen.
- Continue with User Interface (p. 292) for the full UI system reference.

Exporting a Project

Exporting turns editor-authored scenes, resources, scripts, and shaders into a shippable build — either a buildable source project or a compiled binary, depending on the mode you choose.

Choose an export mode

File → Export Project... opens with three modes:

- **Source Code** — generates a self-contained C++ project (engine included) you build yourself with CMake or the platform toolchain. It supports Android, iOS, Web, Windows, macOS, and Linux. Pick this for mobile targets, CI pipelines, or when you want to customize the build.
- **Desktop** — compiles a ready-to-run executable for the OS the editor runs on and copies it with its resources to your destination folder. Resources ship as `assets/` and `lua/` by default, or as one `resources.pak` when the experimental project option is enabled. Needs CMake and a C++ compiler installed. Choose OpenGL or Vulkan on Linux; Direct3D 11, Vulkan, or OpenGL on Windows; and Metal or OpenGL on macOS.
- **Web** — compiles an HTML + JavaScript + WebAssembly build with Emscripten. Needs the Emscripten SDK[4] installed (auto-detected, or point the window at your `emsdk` folder once) and uses WebGL 2.

Desktop and Web reuse the Source Code pipeline internally, building inside `.doriax/export/` in your project — the first export compiles the engine and is slow, later ones are incremental and fast.

Export checklist

Before exporting, verify:

- All scenes are saved.
- The intended startup scene is selected.
- Resource paths are inside the project or otherwise reachable by the exporter.
- **Project Settings → Directories → Native Resource Pack** is configured for the target: leave it off for loose files, or enable it for Desktop/Android after checking its runtime limitations (p. 160).
- Lua and C++ entry points are configured so only the intended startup path creates the main scene.
- Target platform dependencies are installed.
- The output folder is outside the source asset folders you edit by hand.

What the exporter prepares

The editor export pipeline can prepare:

Output	Purpose
Scene code/ data	Serialized scene hierarchy, components, bundles, and child scene references
Resources	Textures, models, fonts, audio, and other project assets, as loose files or an optional native `resources.pak`
Scripts	Lua files and generated C++ glue for script entry points
Shaders	Shader data compiled for each selected graphic backend
Build files	CMake, Gradle, Xcode, Emscripten, or platform-specific project files

4. https://emscripten.org/

Shader output

The exporter supports shader output formats configured by the editor's shader export settings. Use the default unless you are debugging shader packaging or integrating a custom build step.

Build after export

Desktop and **Web** mode exports are already built — run the executable from the destination folder, or serve the web files with any local web server (`python3 -m http.server`).

Source Code exports are built with CMake. The export includes a build script (p. 748) that runs both steps for you:

```
python3 doriax.py
```

or run CMake yourself:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

Android, iOS, macOS, and HTML5 have additional platform requirements. See the Export Window (p. 155) section and the target-specific guide.

Troubleshooting

Symptom	Check
Scene does not load	Startup scene, scene ID/name, exported scene list
Blank window	Active camera, canvas size, render backend, asset paths
Missing textures	Resource folder, texture path case, unsupported source format
Assets disappear only when packing is enabled	Code that constructs `File` directly; packed read-only resources must use `Data` or an engine loader
Export rejects `resources.pak`	Rename the top-level asset; `resources.pak` and `resources.pak.tmp` are reserved export names
Script starts twice	`NO_CPP_INIT` / `NO_LUA_INIT` configuration
Shader errors	Target backend, shader compiler output, generated shader files
Slow or unusually large C++ build	Large inline mesh data; prefer a GLTF or OBJ asset

Exporting without the GUI

You can run the same export from a terminal with the `doriax-editor` CLI — handy for build servers and automated releases:

```
doriax-editor export --project ./MyGame --out ./build/MyGame --backend "vulkan,opengles"
```

See Command-Line Tools (p. 167) for all options.

Continue with Export Window (p. 155) and Build Options (p. 357).

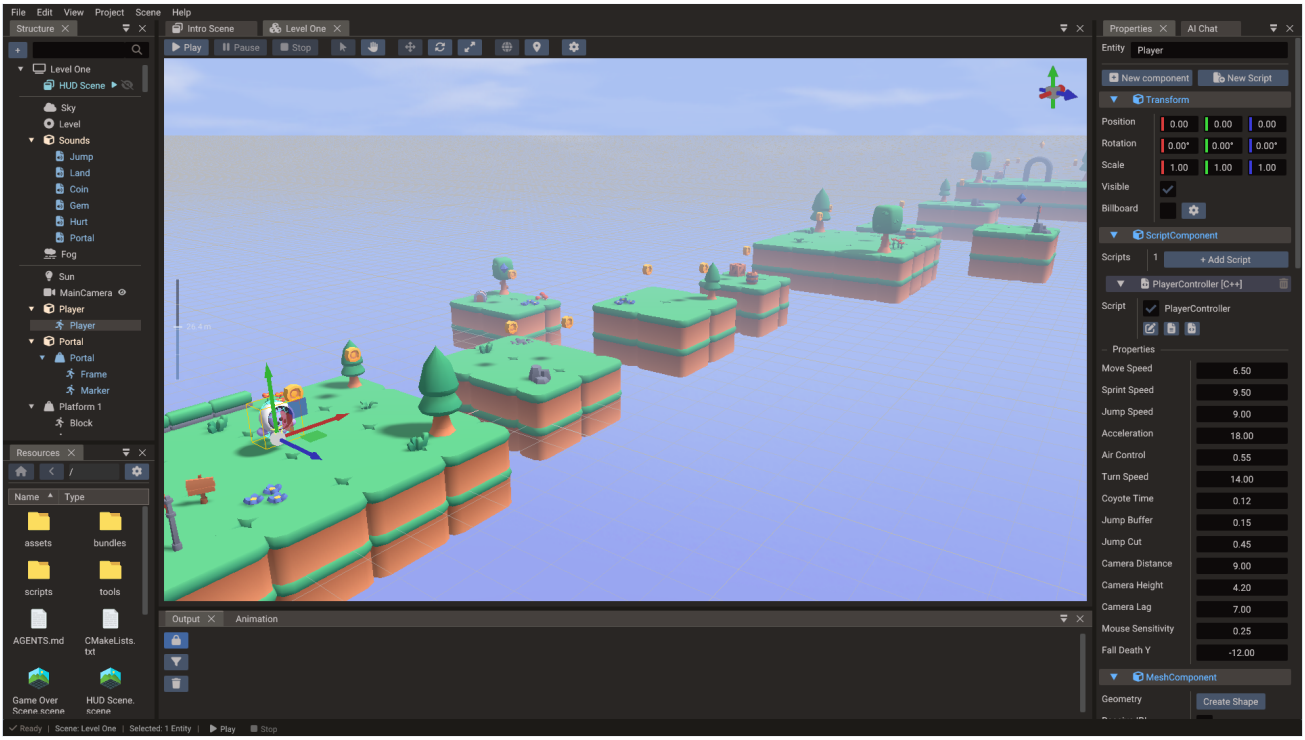
PART 3

Editor

Editor Overview 44
Project Workflow 48
Project Settings 55
Version Control 65
Editor Settings 68
Structure Panel 73
Scene View 81
Properties & Components 90
Resources Browser 106
Bundles 111
Sprite Slicer 116
Tileset Slicer 119
Terrain Editor 123
Animation Timeline 129
Code Editor 136
Custom Shaders 143
Export Window 155
Command-Line Tools 167
MCP Server 172

Editor Overview

The Doriax editor is the primary authoring environment for your projects. It brings together scene editing, resource management, scripting, animation, play mode, and export tooling in one unified window so you can build, test, and ship games without leaving the editor.



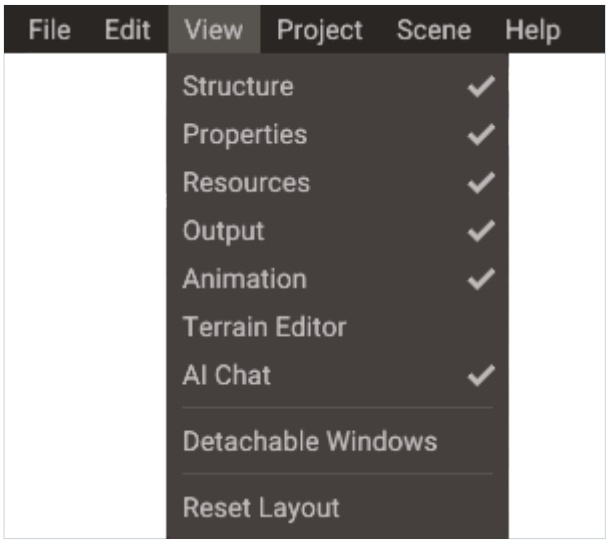
Editor layout

The editor window is divided into a set of resizable panels:

Panel	Purpose
Menu bar	File, project, scene, and play-mode controls
Scene view	Main viewport for 2D, 3D, and UI scene editing with gizmos
Structure panel	Hierarchical tree of scenes and entities
Properties window	Component list and property fields for the selected entity
Resources Browser	Project asset manager: import, preview, and organize files
Animation Timeline	Keyframe editor for object animation, sprite frames, and bones
Code Editor	Integrated Lua and C++ editor with API completion and an event menu
AI Chat	The built-in AI assistant, docked as a tab beside Properties. Its keys are in Editor Settings → AI (p. 70), and outside agents reach its tools through the MCP server (p. 172)
Output panel	Build logs, play-mode diagnostics, and export messages, persisted to `editor.log` (p. 769) for crash reports
Footer	The Git branch (p. 66) and Play / Pause / Stop on the left; FPS, frame time, draw calls, and triangles on the right

Menu bar

Menu	Contents
File	New Project, Open Project (**Ctrl+O**), Recent Projects, Save (**Ctrl+S**), Save All (**Ctrl+Shift+S**), Save Project As, Export Project, Exit
Edit	Undo, Redo, Duplicate, Delete, and Editor Settings (p. 68)
View	Show or hide each panel, Detachable Windows, Reset Layout
Project	Project Settings (p. 55), Manage Scenes, Bundles, Empty Project Trash
Scene	New Scene (3D / 2D / UI), Run, Pause, Resume, Stop, Detach Camera (p. 89), Full Screen (p. 89), Remove Scene from Project
Help	Documentation (**F1**), Keyboard Shortcuts, Report an Issue, About Doriax



Save writes the one thing you were last working on — the focused script, or the selected scene. **Save All** writes every changed scene and script *and* `project.yaml`, which is how settings changes are committed to disk. On a project that has never been saved, both ask where the project should live first.

Empty Project Trash permanently deletes the project's trashed files and asks for confirmation first. **Remove Scene from Project** drops a scene from the project, after offering to save it if it has unsaved changes.

Main workflows

Task	Where it happens
Create and configure scenes	Project Workflow, Structure panel, scene save dialog
Place and transform entities	Scene view gizmos
Edit component data	Properties window
Import and preview assets	Resources Browser
Share PBR materials across meshes	Create `.material` files (Properties → Resources drag), link from Scene view
Slice sprite sheets and tilesets	Sprite Slicer, Tileset Slicer
Sculpt, paint and populate terrain	Terrain Editor
Write Lua or C++ scripts	Code Editor and script creation dialog
Animate objects and characters	Animation Timeline
Test gameplay	Play / Pause / Stop controls
Prepare builds	Export Window

Scene types

The editor supports three scene workflows:

Type	Purpose
3D scene	Games with perspective cameras, 3D models, PBR materials, sky/IBL, and lighting
2D scene	Top-down or side-scrolling games with sprites and tilemaps
UI scene	Menus, HUDs, overlays, and screen-space controls using the UI system

All three types share the same ECS foundation, camera, and scripting system. A project can load multiple scene types simultaneously — for example, a 3D gameplay scene with a UI scene overlay.

Play mode



Press **Play** to run the game inside the editor. The editor takes a snapshot of the current scene state before entering play mode and restores it when you stop, so runtime mutations do not permanently corrupt your authored data. Entities the game creates while playing are listed in the Structure panel, read-only, until **Stop** destroys them (see Structure panel — While playing (p. 80)).

Use play mode for quick iteration. For platform-specific validation (mobile input, web memory limits, native graphics), test with a proper exported build. The footer shows FPS, frame time, draw calls, and triangles while the viewport is drawing; the `benchmark` (p. 169) command records the same counters with VSync forced off.

Undo and redo

All edits made in the scene view, Properties window, and hierarchy are recorded in the command history. Use **Ctrl+Z** / **Ctrl+Y** to step back and forward through changes. Undo is available across transforms, component edits, hierarchy reparenting, resource assignments, and cascade deletes (a parent and its children restore together).

Keyboard shortcuts

Help → Keyboard Shortcuts shows this list inside the editor. On macOS, menu shortcuts use **Command** instead of Ctrl.

Shortcut	Action
Ctrl+O	Open project
Ctrl+S	Save the current scene or script
Ctrl+Shift+S	Save all scenes, scripts, and the project
Ctrl+Z	Undo
Ctrl+Shift+Z / **Ctrl+Y**	Redo
Ctrl+D	Duplicate selection
Delete	Delete selection
F2	Rename selection
F5	Run current scene, or resume a paused one
F6	Pause
F7	Stop
F8	Detach or re-attach the camera while running (Detached camera (p. 89))
F11	Show the running scene over the whole editor window, or go back (Full screen (p. 89))
W / **E** / **R**	Move / Rotate / Scale gizmo
T	Toggle local / global transforms
F / **Home**	Frame selection / all objects (3D)
F1	Open the documentation

Duplicate and Delete follow the selection: with a tile or a tilemap instance selected they act on that, otherwise on the selected entities, and Delete also removes a selected child scene reference. F2 jumps to the name field in the Properties window (p. 90) for the selected entity, or for the scene when nothing is selected. In the Resources window (p. 109), F2 renames the selected file and Ctrl+D duplicates the selected files. They are ignored while typing in a text field, so a shortcut never steals a keystroke from an input.

Detailed pages

- Project Workflow (p. 48)
- Project Settings (p. 55)
- Version Control (p. 65)
- Editor Settings (p. 68)
- Structure Panel (p. 73)
- Scene View (p. 81)
- Properties & Components (p. 90)
- Resources Browser (p. 106)
- Sprite Slicer (p. 116)
- Tileset Slicer (p. 119)
- Terrain Editor (p. 123)
- Animation Timeline (p. 129)
- Code Editor (p. 136)
- Export Window (p. 155)
- Command-Line Tools (p. 167)
- MCP Server (p. 172)

Project Workflow

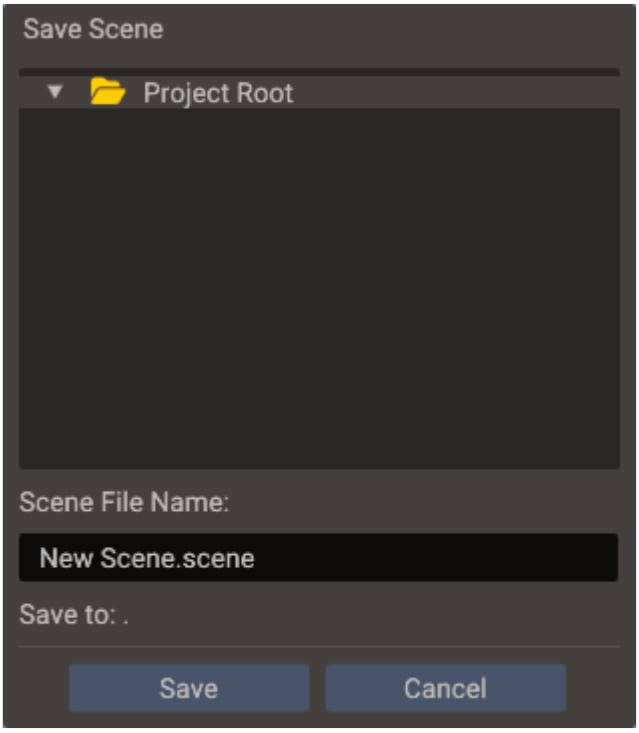
A Doriax project is the editable source for a game. It organizes scenes, resources, scripts, bundles, and export configuration into a single directory that the editor understands. The editor keeps visual authoring, code, and build settings clearly separated so teams can iterate and collaborate effectively.

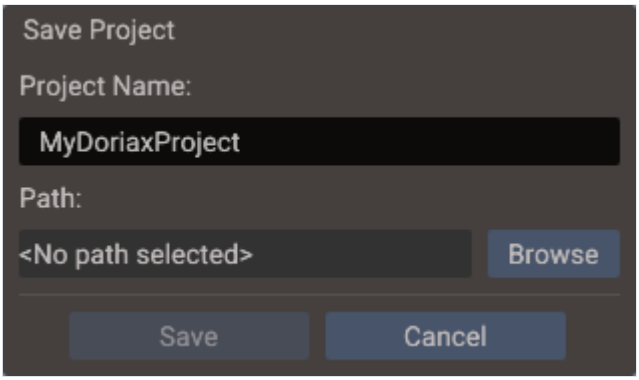
Creating a project

Doriax does not display a startup dialog. When the editor launches, it reopens the last saved project if that project is still available. Otherwise, it opens a temporary project named `MyDoriaxProject` with a new 3D scene so you can begin editing immediately. **File → New Project** also replaces the current project with a fresh temporary project.

To keep a temporary project:

1. Choose **File → Save All** (**Ctrl+Shift+S**). On a temporary project, **Save** (**Ctrl+S**) does the same thing, because there is nowhere to save a single file to yet.
2. The editor saves the open scenes and scripts, asking where in the project any never-saved scene should go, and then asks for the project itself.
3. Enter the project name, click **Browse**, and select an empty directory.
4. Click **Save**. The editor moves the temporary project into that directory and writes its `project.yaml` file.





Temporary projects are stored in the operating system's temporary directory. Save the project to another directory before relying on it as permanent work.

A project can be open in only one editor at a time, so two editors never overwrite each other's saves. If the last project is already open in another Doriax editor, a newly started editor says so and opens a temporary project instead. **Open Project** and **Recent Projects** refuse it with the same message. Each editor keeps its temporary project in a folder of its own.

New projects begin with a 3D scene. To add a different scene type, choose **Scene → New Scene → 2D Scene** or **Scene → New Scene → UI Scene**. Doriax does not use project templates.

Project anatomy

Area	Purpose
`assets/` (or `resources/`)	Textures, models, fonts, sounds, materials, and other imported files. The folder chosen as the assets directory is the root every asset reference is stored against
`scenes/`	YAML scene files edited by the visual editor
`scripts/`	Lua scripts and C++ source files
`shaders/`	Default location for forked shader sources (`.vert` / `.frag` / `.glsl`), and where compiled `.sdat` output goes — see Custom Shaders (p. 143)
`bundles/`	Reusable entity hierarchy files
`settings/`	Scene startup references, build target settings, and export configuration
`.doriax/`	The editor's own working directory: generated C++, build trees, models converted from FBX, and the settings that belong to your machine and your session. Recreated as needed and never committed — see Version Control (p. 65)

Folder names may differ by project. The structure is a convention, not a strict requirement — you can reorganize asset folders and update resource paths accordingly.

Scene files

Scene files are serialized as YAML. They store the full scene hierarchy: entities, component data, transform values, camera settings, and child scene references. Where you left the *editor* camera is not part of the scene — that is per-user state, kept out of the file so it never appears in a commit.

Keep scene files focused. For large games, split gameplay areas into separate scenes and compose them with child scene references. This keeps individual YAML files small and merge-friendly in version control.

Scene types

The editor offers three scene workflows:

Type	Editor defaults	Typical content
3D	Perspective editor camera, 3D gizmos, lighting panel	Models, lights, camera, terrain, physics bodies
2D	Orthographic camera, 2D canvas gizmos	Sprites, tilemaps, polygon shapes, 2D physics
UI	Screen-space canvas, anchor gizmos	Buttons, text, images, panels, scrollbars

All three share the same `Scene` class at runtime — the type mainly affects editor defaults and authoring tools. A 3D scene can contain UI entities, and a UI scene can reference 2D or 3D child scenes.

Child scenes

Child scenes let one scene reference and load another. This is useful for:

- **Persistent UI layers** — load a HUD scene as a child of the gameplay scene.
- **Shared environment** — a common lighting rig or skybox loaded by all levels.
- **Level assembly** — large worlds assembled from smaller room or chunk scenes.
- **Streaming** — load and unload sections independently at runtime.

Each scene runs its own physics world, so bodies in different scenes don't collide. See Physics across scenes (p. 323) before you split gameplay physics across chunks.

To add a child scene reference, use the Structure panel's **Add Child Scene** option and point it to a saved scene file.

Bundles

Bundles are prefab-like reusable entity hierarchies. A bundle can represent an enemy, an interactive prop, a UI card, or any repeated arrangement of entities and components. They are defined visually in the editor and stored as YAML bundle files.

Why bundles?

Advantage	Detail
Shared across scenes	The same bundle can be placed in any number of scenes without duplicating data
Single source of truth	Changing a bundle updates every instance in every scene that references it
Runtime factory	During export, bundles become factory functions; `BundleManager` creates and destroys instances at runtime
No coupling	Bundles are independent of the scenes that use them — they can be loaded on demand

Creating a bundle

1. Build the entity hierarchy you want to reuse in any scene.
2. In the Resources Browser, open the folder the bundle should be saved to.
3. Select the root entity in the Structure panel, right-click it and choose **Save as Bundle** — or simply drag the entity onto the Resources Browser.

Either way the editor writes `<EntityName>.bundle` into the open folder and replaces the hierarchy in the scene with an instance of the new bundle. Rename the file in the Resources Browser if you want a different name. A `.bundle` file can live in any project folder, so `bundles/` is a convention rather than a requirement.

Placing bundles

Drag a `.bundle` file from the Resources Browser onto the Structure panel — drop it on the scene root, or on an entity with a `Transform` to parent the instance there.

Runtime bundle usage

Bundle registration is generated automatically by the export step. At runtime you call `createBundle` (bundle **name first, then scene**) and `destroyBundle`. Entity IDs are scene-local, so parent under an existing entity with `createBundle(name, scene, "parentName")` or pass an object that already carries its scene:

Lua

```lua
-- Create a bundle instance at runtime
local root = BundleManager.createBundle("enemy_grunt", scene)
-- Parent the new instance under a named entity in that scene
BundleManager.createBundle("enemy_grunt", scene, "spawn")

-- Destroy it later
BundleManager.destroyBundle(scene, root)
```

C++

```cpp
// Registration is normally emitted by export:
BundleManager::registerBundle(1, "enemy_grunt", [](Scene* scene, Entity parent) -> bool {
    // bundle factory body
    return true;
});

Entity root = BundleManager::createBundle("enemy_grunt", &scene);
BundleManager::createBundle("enemy_grunt", &scene, "spawn");
```

A bundle no scene instantiates is only built and registered while it is listed in **Project → Bundles** — see Bundles in the build (p. 114).

See BundleManager (p. 597) for the complete API.

Settings

Settings are split by what they belong to, which decides where they are stored and whether they travel with the project:

Settings area	Opened from	Stored in	Holds
Project settings (p. 55)	Project → Project Settings	`project.yaml`	Start scene, application identity, canvas, window, VSync, asset/Lua/script directories, native resource packaging, and the per-platform settings for Web, Linux, Windows, macOS, iOS and Android
Editor settings (p. 68)	Edit → Editor Settings	`settings.yaml`, with API keys and the MCP token in `ai_keys.dat`	Compiler kit, CMake and Emscripten paths, default export directory, editor VSync, UI scale, AI Chat keys and limits, the MCP server (p. 172)
Export settings	File → Export Project	`project.yaml` (`export`) and the project's `.doriax/user/build.yaml`	Shader overrides and graphic backends in the project; output directories per machine
Machine settings	—	`.doriax/user/build.yaml`	The compiler kit, build jobs and export folders chosen for *this* project on *this* machine
Editor workspace	—	`.doriax/user/workspace.yaml`	Open tabs and their order, the selected scene, each scene's editor camera and viewport guides, the terrain brush
Window and layout state	—	`settings.yaml`	Window size, maximized state, recent projects, panel visibility, Resources Browser preferences

Two dividing lines run through that table. The first is the machine: a compiler path, an SDK location, an output folder. The second is the session: which tabs you had open and where you left each camera. Neither belongs to the game, so both are kept out of `project.yaml` and out of version control — see Version Control (p. 65).

Everything in the last three rows lives beside the project rather than in the editor's own configuration, so **Save Project As** carries it along and a project copied to another drive keeps working.

Shaders have no directory setting: each fork picks its own location when you create it, and compiled `.sdat` output always goes to the project's `shaders` folder. See Custom Shaders (p. 154).

The rest of this page covers the behavior behind a few of those settings. For the field list of each dialog, see Project Settings (p. 55) and Editor Settings (p. 68).

Assets and Lua directories

Project → Project Settings → Directories picks the two folders the running game reads from. Both default to `<Project root>`, which keeps the whole project as the root.

Setting	Root for	Resolved at runtime through
Assets Directory	Textures, models, sounds, fonts	`asset://`, and every plain relative path
Lua Directory	Lua script entries and the data files they read	`lua://`

References are stored **relative to these roots**, not to the project. With the assets directory set to `assets`, a texture in `assets/textures/hero.png` is stored as `textures/hero.png`, which is exactly what the exported game finds next to its executable.

Changing either directory migrates the project in one step:

- referenced files that would fall outside the new root are **moved into it**, keeping their folder layout so a model still finds its `.bin` and textures. Files nothing references, such as one a script loads by path, stay where they are;
- every reference is rewritten in scenes, bundles and `.material` files — scenes that are not open are loaded for the rewrite and closed again;
- folders the moves leave empty are deleted. The project folder and the two new directories stay even when empty, and so does a linked folder, whose target may hold other files;
- if any file cannot be moved, every move is undone and the directories are left unchanged, so the project is never half-migrated.

Before it moves anything, the editor checks that the change can finish, and refuses it with the reason in an alert when:

- a different file is already where a moved file would go. An identical copy is fine, and the reference points at it. Otherwise move or rename one of the two first;
- a move would leave the new directory through a symbolic link;
- the new directory is a file, or a path under one;
- a scene, bundle, `.material` file or `project.yaml` cannot be written, since the rewrite may have to save it.

If a save still fails after the files have moved, for example on a full disk, the alert lists the files that kept the old paths. Open scenes and bundles keep the new paths in the editor, so saving them again fixes them.

Because a reference cannot leave its root, the editor only accepts assets from inside the assets directory: dropping a file from elsewhere previews it but shows a warning instead of assigning it. Maps painted in the Terrain Editor (p. 123) (under `terrain_maps/`) and assets downloaded by the AI assistant are created under the assets root for the same reason. Moving a referenced file out of the assets directory clears the references to it, exactly like deleting it.

Script directories

The **Directories** tab also has a **Script Directories** list, which is about the C++ build rather than what the running game reads. Each folder you add is both an include root and a compile root: headers under it are included by the path below the folder, and every `.cpp` under it is compiled even when no entity references it.

The list starts empty, and only the folders you add are affected — a project without script directories builds exactly as before. Use **Add** to pick a folder and the trash icon to remove one. Keep them inside the project: an outside folder builds in the editor but is skipped when exporting.

See C++ Build Setup → Script directories (p. 211) for the full behaviour, and Customizing the build (p. 212) for `ProjectBuild.cmake`, which covers what this setting does not — linking libraries and other CMake of your own.

Native resource pack

Project → Project Settings → Directories → Native Resource Pack controls whether a native export ships the copied assets and Lua files as loose directories or combines them into one `resources.pak` file. The option is experimental and disabled by default. It is intended for **Desktop** exports and Android builds made from **Source Code** exports; Web exports continue to use Emscripten's own `.data` bundle.

The choice is saved as `packNativeResources` in `project.yaml`, so the Export Window, command-line export, and editor automation all use the same project setting. Packing is an export step and never changes the source assets or Lua directories. Editor Play mode always reads those loose project files.

Packed entries are obfuscated, not encrypted, and each entry is loaded into memory when opened. Engine resource loaders, Lua modules, and `Data` can read the pack transparently, but a direct `File` handle cannot. Review the Native resource pack (p. 160) section before enabling it for a project that opens asset or Lua files itself.

VSync

Open **Project → Project Settings** and use **VSync** to choose whether Play mode and supported desktop builds synchronize frames to the display refresh rate. It is enabled by default. Disable it when profiling the maximum frame rate; leave it enabled for normal use to avoid tearing and unnecessary GPU load.

The value is saved as `vsync` in `project.yaml` and travels with the project. Older projects without the field default to enabled.

Runtime path	VSync off behavior
Editor Play mode	Uncaps the editor render loop while a scene is running; editing frames follow **Editor Settings → Editor VSync** instead
Generated standalone GLFW build	Uses swap interval `0`
Exported Linux GLFW, Windows/Linux Sokol OpenGL, or Windows Direct3D 11 build	Uses swap interval `0`
Exported Vulkan build	Prefers Immediate presentation, then Mailbox; falls back to FIFO if neither is supported by the system
Exported macOS Metal build	Remains synchronized; the generated CMake project prints a warning

> **Platform control**
>
> A graphics driver, compositor, or window system may still impose presentation limits after VSync is disabled. Compare a standalone build when collecting final performance numbers because the editor adds its own rendering overhead.

Editor VSync

The project setting above covers Play mode and builds. The editor's own frames follow **Edit → Editor Settings → General → Editor VSync**, which is enabled by default and saved as `vsync` under `editor` in `settings.yaml`. It belongs to the machine rather than the project, so it is not exported and does not travel with a project you share.

The two settings never apply at once: while a scene is running frames follow the project setting, and the rest of the time they follow this one. Toggling takes effect on the next frame, with no restart. An unfocused editor window drops synchronization either way, so a blocked present can never stall the window's event loop. It is paced to the display's refresh rate instead, so a running scene keeps its frame rate.

Disabling it only lifts the frame cap while you are interacting or a scene is redrawing. An idle editor waits on the window system regardless, so it does not spin. A pointer resting on the scene view does not count, and moving it counts only while a gizmo or the terrain brush is shown.

Window

Open **Project → Project Settings** to control the OS window that desktop builds create at startup:

Setting	Default	Effect
Window Mode	Windowed	Initial window state: `Windowed`, `Maximized`, or `Fullscreen`
Window Width / Height	Canvas size	Initial window size in pixels, and the size restored when leaving fullscreen
Window Resizable	Enabled	Whether the player can resize the window
Window Title	Project name	Title-bar text; leave empty to use the project name
Icon	None	Application icon for desktop builds — see Project Settings → Window (p. 57)

The values are saved as `windowMode`, `windowWidth`, `windowHeight`, `windowResizable`, and `windowTitle` in `project.yaml` and travel with the project. Projects saved before these fields existed start windowed at their canvas size.

`Fullscreen` starts the game on the primary monitor at the current desktop video mode (borderless-style, no display mode switch) and falls back to windowed when no monitor is available. Scripts can still toggle fullscreen at runtime with `requestFullscreen` / `exitFullscreen` (p. 409); leaving fullscreen restores the configured window size.

Runtime path	Behavior
Editor Play mode	Not affected — Play renders inside the editor viewport
Generated standalone GLFW build	All settings honored
Exported Linux GLFW build	All settings honored
Exported Windows/macOS Sokol build	Size, title, and fullscreen honored; `Maximized` falls back to windowed and the window is always resizable
Exported macOS Xcode (apple backend) build	Uses the storyboard-defined window; project window settings are not applied
Web, Android, iOS	Ignored — the game always fills the browser canvas or the device screen

Save strategy

Save (**Ctrl+S**) saves one thing: the script you last edited if the Code Editor had focus, otherwise the selected scene. **Save All** (**Ctrl+Shift+S**) saves every changed scene and script *and* writes `project.yaml`, which is what commits settings changes — the settings dialogs apply to the open project immediately but do not write the file themselves. **Save Project As** moves the project to another folder, taking its machine-local editor settings with it.

Scenes, bundles, and `project.yaml` are written to a temporary file that then replaces the old one, so a save that fails (a full disk, a read-only folder) leaves the previous file intact.

- Save scenes after structural edits (**Ctrl+S**).
- Use **Save All** after changing project or export settings, so `project.yaml` is written.
- Keep generated export output outside the source project folder.
- Commit human-authored files (scenes, scripts, assets) and `project.yaml`; the editor writes the `.gitignore` that excludes the rest — see Version Control (p. 65).

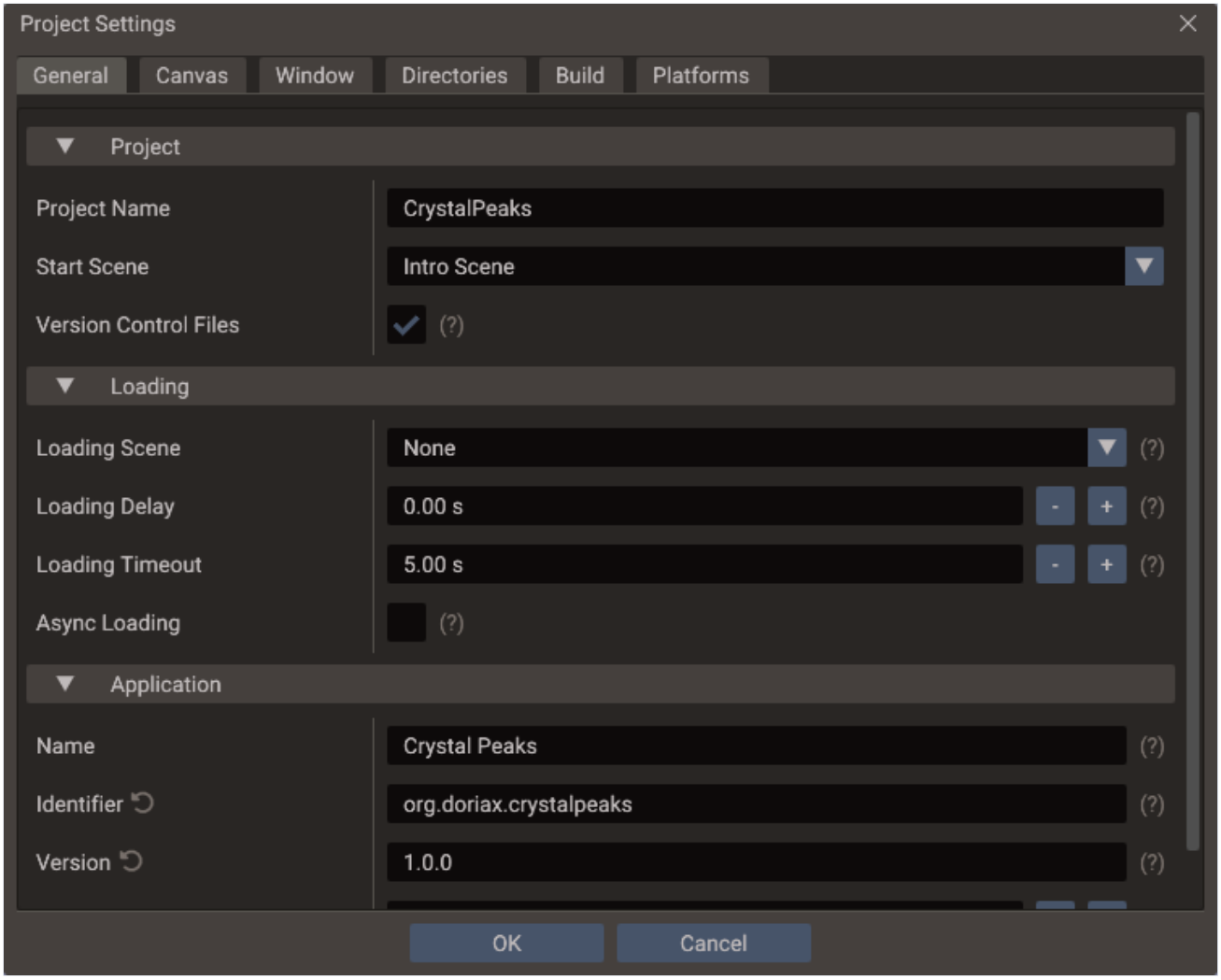
Project Settings

Project → Project Settings holds everything the project itself carries: how the game is identified, how its canvas and window are set up, which folders it reads from, and what each platform adds on top of that. Every value on this dialog is saved in `project.yaml` and travels with the project in version control.

Machine-specific settings — the compiler, the CMake and Emscripten paths, the default export folder — are **not** here. They live in Editor Settings (p. 68).

The dialog is a modal with six tabs and an **OK** / **Cancel** footer: nothing is written until you press OK. Settings that differ from their default show a small restore arrow next to the label; click it to put the default back. Hover the **(?)** marker at the end of a row for the setting's description.

General



Project

Setting	Effect
Project Name	The project's name. Also the default application name, window title, and the identifier the exported executable and CMake target are named after
Start Scene	Scene the exported game loads at launch. Only saved scenes can be chosen
Version Control Files	Enabled. Keeps `.gitignore` and `.gitattributes` written at the project root, so the editor's working directory, your machine's build settings and each developer's own layout stay out of the repository. In a Git repository, it also shows the branch in the footer and marks changed lines in the Code Editor

The Export Window shows the same **Start Scene** and writes back to this setting, so the two never disagree.

Version Control Files refreshes both files whenever the project is saved, and never replaces one the editor did not generate — see Version Control (p. 65). It also turns on the Git status (p. 66) the editor shows.

Loading

The loading screen that `SceneManager::loadScene` shows at startup and between scenes. Exported games and the editor's Play apply these at startup.

Setting	Default	Effect
Loading Scene	None	UI scene shown on top while the game starts or switches scenes, until the new scene has loaded. Only saved scenes can be chosen
Loading Delay	`0 s`	How long the loading scene covers the old scene before it is replaced: time for a fade in and for a click sound to finish
Loading Timeout	`5 s`	How long a load waits without progress before it gives up
Async Loading	Off	Exported games load textures, models and sounds on worker threads, so the loading scene keeps animating and its progress moves. The editor always loads this way. Web builds without thread support load synchronously

See Loading screens (p. 335) for how a transition runs and how the loading scene can animate its progress and fade out.

Application

One identity, shared by every platform. Each platform section can override any of these four values; leaving an override empty inherits from here.

Setting	Default	Used for
Name	The project name	The name every platform shows to the player
Identifier	`com.yourcompany.project`	Reverse-DNS id behind the Apple bundle identifiers, the Android package name, and the web saves (p. 411)
Version	`1.0`	The version shown to the player
Build	`1`	Apple build number and Android version code. Increase it for every release

Because each platform has its own rules for these fields, the editor validates as you type and reshapes the value at export:

Field	Accepted here	Reshaped to
Identifier	Letters and digits between dots, e.g. `com.company.game`	Apple `CFBundleIdentifier`, Android `applicationId`
Version	Up to three numbers from 0 to 65535, e.g. `1.0`	Apple takes at most three parts; Windows pads to four (`1.0` → `1.0.0.0`)
Build	A whole number	Apple `CFBundleVersion`, Android `versionCode`

> **The executable name comes from the project name**
>
> **Name** is the display name. The exported binary, the CMake target, and the Windows `InternalName` / `OriginalFilename` are derived from the **project name** instead, so renaming the application does not rename the executable.

Canvas

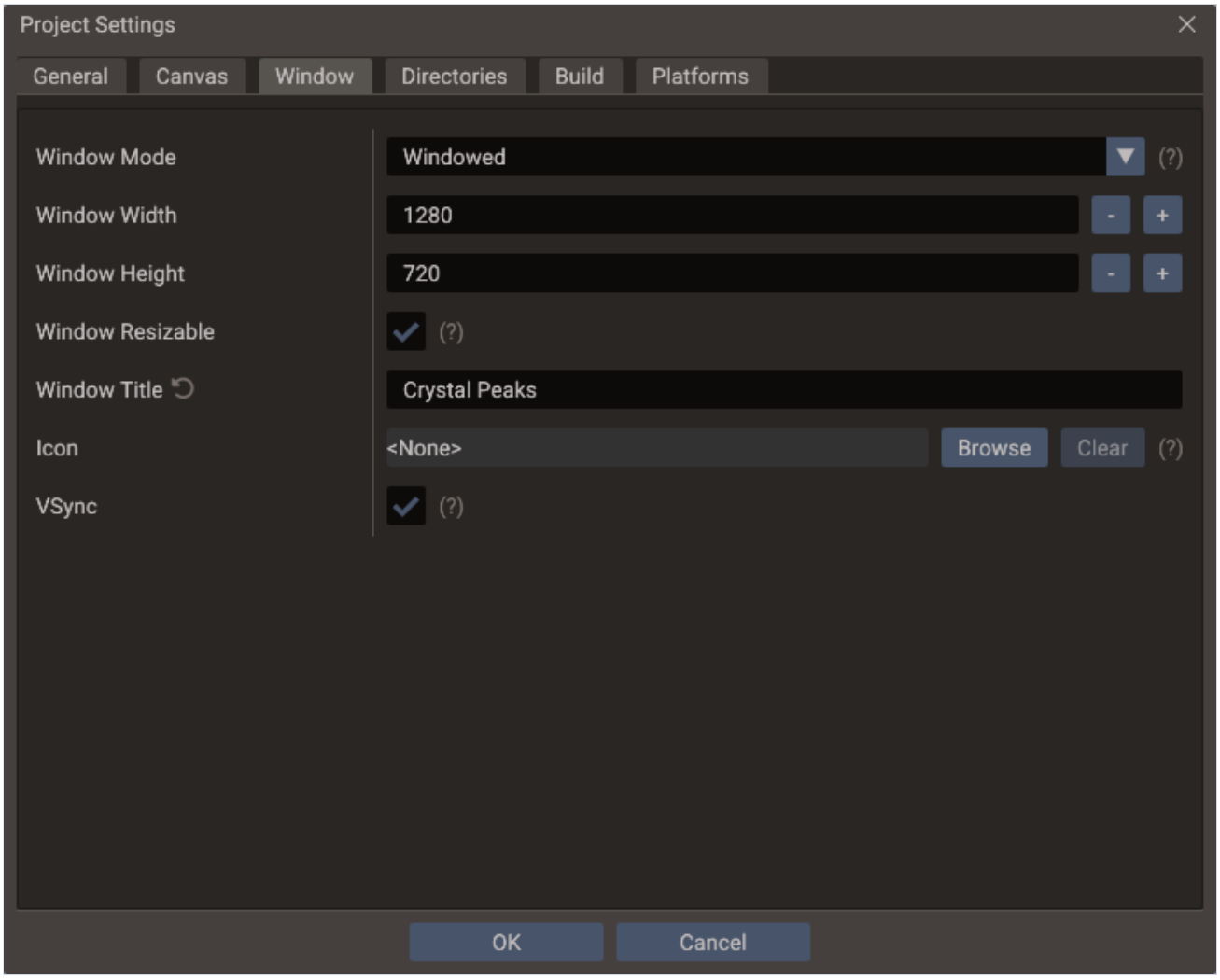
Setting	Effect
Canvas Width / Height	The design resolution the scene is authored against

Setting	Effect
Scaling Mode	How that canvas maps to the real screen. A live preview under the dropdown shows the result
Texture Strategy	Whether non-power-of-two textures are padded (Fit), scaled (Resize) or kept as they are (None, the default). Only very old GPUs need the first two. The editor applies a change when the project is reopened

See Multiple Resolutions (p. 288) for what each scaling mode does at runtime.

Window

Controls the OS window desktop builds create at startup.

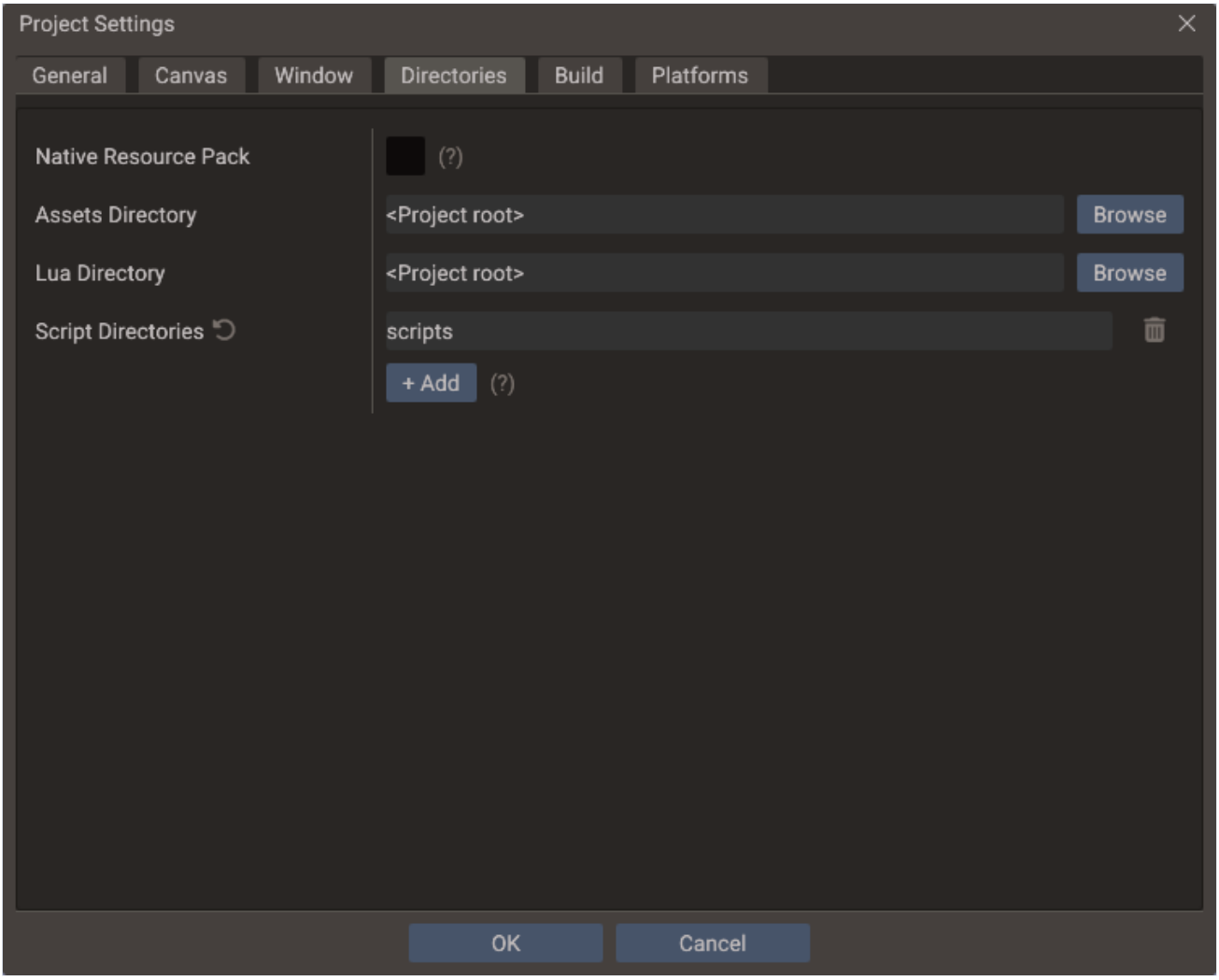


Setting	Default	Effect
Window Mode	Windowed	Initial window state: `Windowed`, `Maximized`, or `Fullscreen`
Window Width / Height	Canvas size	Initial window size in pixels, and the size restored when leaving fullscreen
Window Resizable	Enabled	Whether the player can resize the window. Exported Windows and macOS builds are always resizable
Window Title	Project name	Title-bar text; leave empty to use the project name
Icon	None	Application icon for desktop builds. Square PNG, 256×256 or larger recommended
VSync	Enabled	Synchronize Play mode and supported desktop builds to the display refresh rate

The **Icon** is the desktop icon for all three desktop platforms: it is embedded into the Windows executable, used as the window and taskbar icon on Linux and Windows, and becomes the dock icon on macOS. Apple *bundle* icons and Android launcher icons are separate settings on the Platforms tab. A file inside the project is stored as a project-relative path, so the setting survives moving the project folder.

See Project Workflow — VSync (p. 53) and Window (p. 54) for the per-backend behavior tables, and Editor Settings (p. 68) for the editor window's own VSync.

Directories

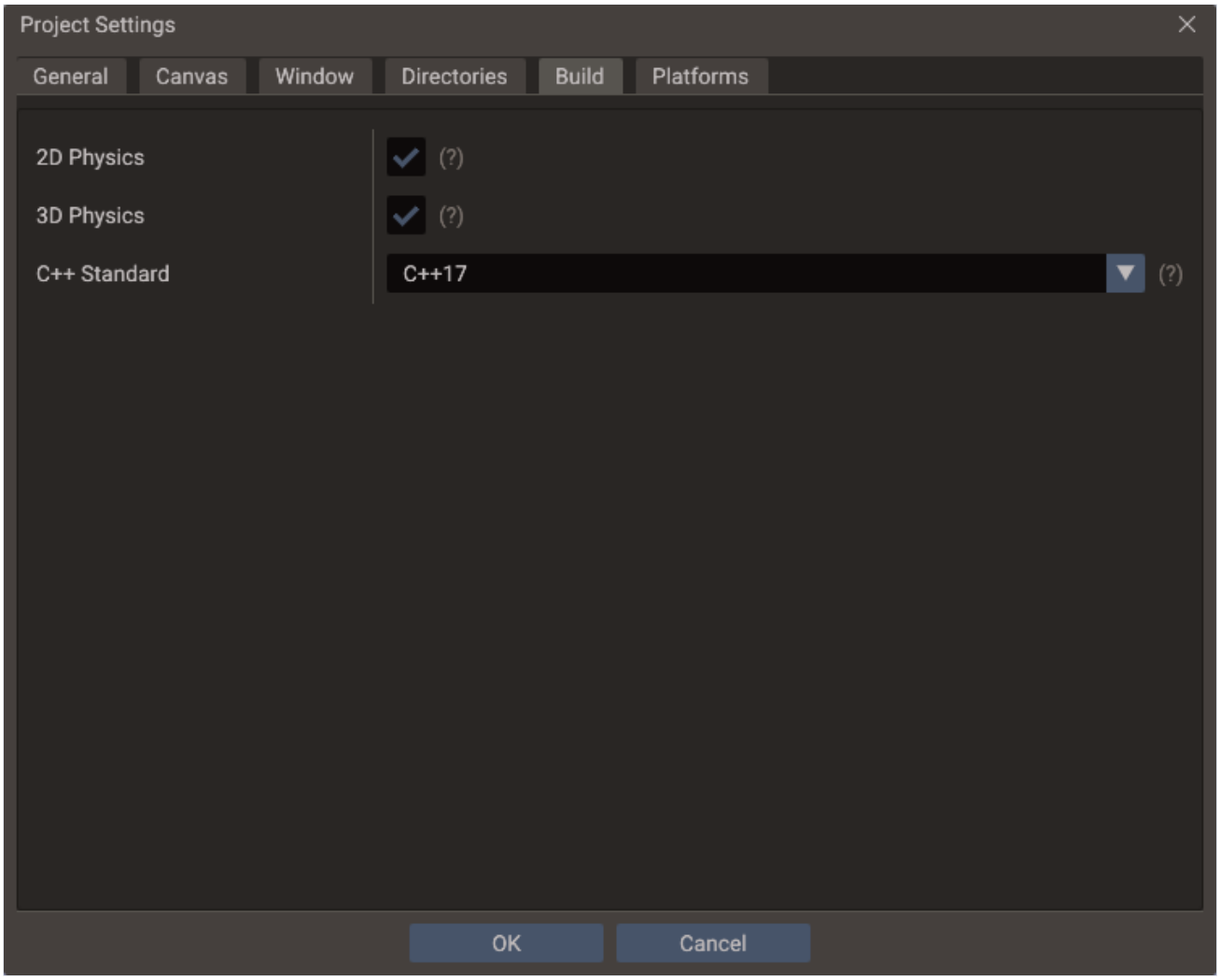


Setting	Effect
Native Resource Pack	Experimental. Packs exported assets and Lua files into one `resources.pak` for Desktop exports and Android builds made from Source Code exports
Assets Directory	Root for textures, models, sounds and fonts — what `asset://` and every plain relative path resolve against
Lua Directory	Root for Lua script entries and the data files they read — what `lua://` resolves against
Script Directories	Extra C++ include *and* compile roots for the build

Asset and Lua references are stored relative to those two roots, so changing one migrates the whole project in a single step — see Assets and Lua directories (p. 52). Script directories are about the C++ build rather than what the running game reads; see C++ Build Setup → Script directories (p. 211).

The native resource pack has runtime restrictions — packed entries are read through `Data`, not `File` — described in Export Window → Native resource pack (p. 160).

Build



Setting	Default	Effect
2D Physics	On	Builds Box2D into exported games
3D Physics	On	Builds Jolt into exported games
C++ Standard	`C++17`	Language standard the project's C++ scripts are compiled with, in Play and in exported games

The standard belongs to the project rather than to your machine — a script using `consteval` needs C++20 wherever it is built — so it is saved in `project.yaml` and travels with the project. Play and export both compile scripts with it, and an export compiles the engine with it too.

Which compilers can *satisfy* it is machine-specific: pick one under Editor Settings → Desktop (p. 68). C++17 works anywhere the editor runs; C++20 and C++23 need a recent MSVC, GCC or Clang.

> **The generated `CMakeLists.txt` follows this setting**
>
> `CMAKE_CXX_STANDARD` in the generated project comes from here, so there is nothing to change by hand — and any edit is overwritten on the next Play or Save. See C++ Build Setup → Customizing the build (p. 212).

Physics backends

Turn off a backend the game never uses and the export leaves it out — its library, its bodies and joints, and their C++ and Lua API — for a smaller build that compiles faster, which matters most on Web and mobile. Desktop, Web and Source Code exports all honor it, including the generated Xcode and Android Studio projects.

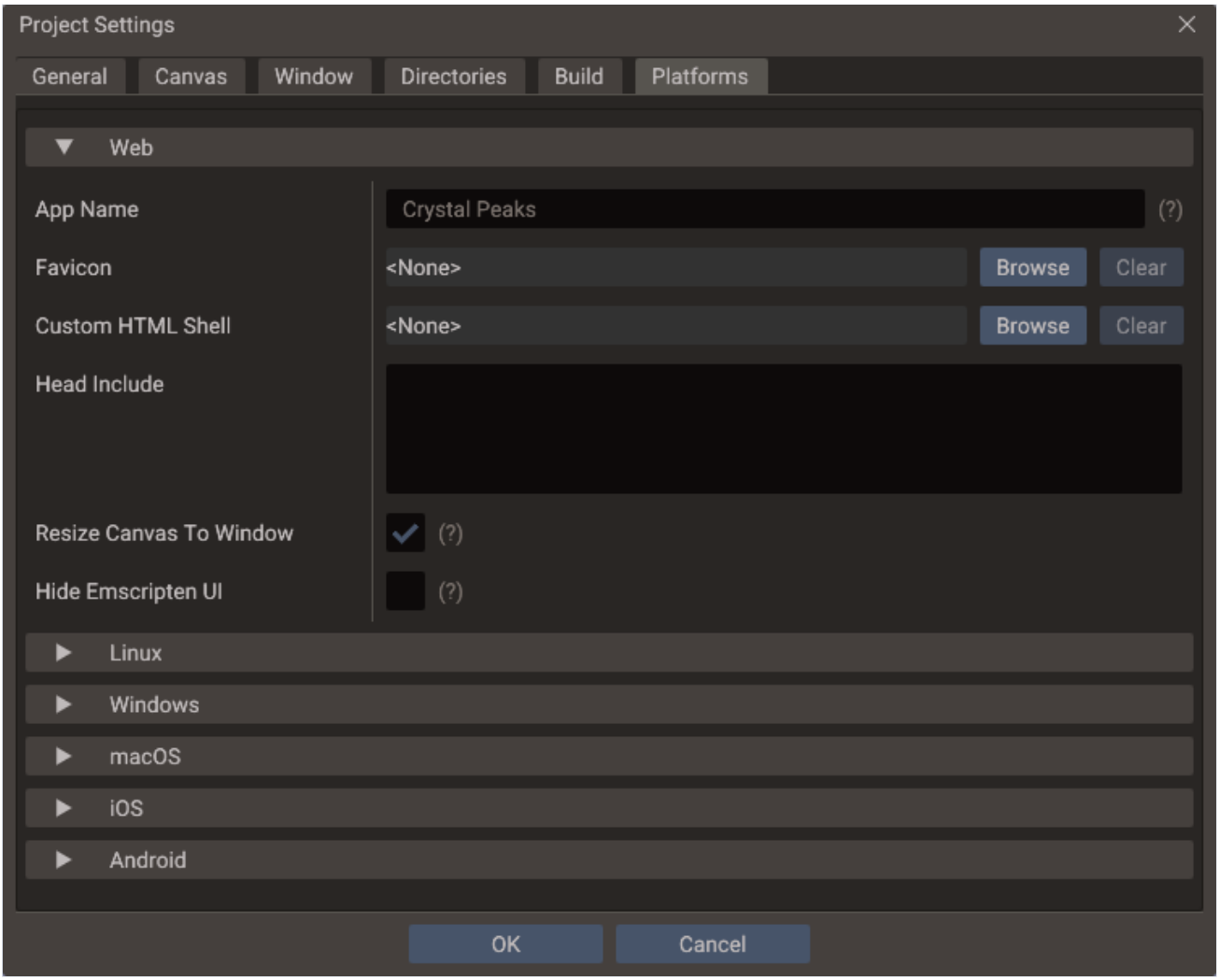
Play is not affected: the editor always runs with both backends, so a scene can still hold a body the export will drop. The export warns when it does, naming the scenes, and the exported game runs without those bodies and joints. A C++ script that includes `Body2D.h` or `Body3D.h` stops the export build with an error naming this setting; code that must build either way can test `DORIAX_PHYSICS_2D` and `DORIAX_PHYSICS_3D`, which are only defined for an enabled backend. For picking and line-of-sight without any physics, cast against mesh bounds with `RayFilter::BOUNDS` (p. 651).

Platforms

Each platform gets its own collapsible section, closed by default. A section holds only what that platform *adds* on top of the shared Application block: any name, identifier, version or build field left empty inherits, and the field's placeholder shows the value it would inherit as you type.

These settings are written into the exported project. Web, Linux and Windows settings apply to every export mode that reaches those platforms; **macOS, iOS and Android settings are applied to Source Code exports**, which is where the Xcode and Android Studio workspaces are generated.

Web



Setting	Default	Effect
App Name	Shared name	Title of the exported web page
Favicon	Doriax icon	Browser-tab icon, copied next to the page and linked from its `<head>`

Setting	Default	Effect
Custom HTML Shell	None	HTML wrapper for the generated page. The file **must** contain a `{{DORIAX_DEFAULT_HTML}}` marker
Head Include	Empty	HTML inserted into the page's `<head>`
Resize Canvas To Window	Enabled	Resizes the rendering surface to the browser window. Aspect ratio and stretching still follow the canvas **Scaling Mode**
Hide Emscripten UI	Disabled	Hides the default logo, status line, controls and output console. The nodes are kept alive, so the SDK's status callbacks keep working
Game Portal	None	The web game portal whose SDK the export builds in, for the `WebPortal` (p. 626) class: CrazyGames, Poki, GameDistribution, Yandex Games or YouTube Playables
Portal Game ID	Empty	GameDistribution only: the game's ID from the GameDistribution dashboard. Without it the SDK is not loaded

A custom shell replaces `{{DORIAX_DEFAULT_HTML}}` with the page Emscripten generated; the build fails with a clear CMake error when the marker is missing. Two more optional markers give you control over placement: `{{DORIAX_TITLE}}` is replaced with the app name, and `{{DORIAX_HEAD_INCLUDE}}` receives the favicon link, the head include, and the styles the two checkboxes add. Without that second marker the block is inserted before `</head>`.

A game sent to several portals needs one export per portal, since each build includes one SDK. Choosing YouTube Playables also adds YouTube's SDK `<script>` to the page head, which YouTube requires to load before the game. See Monetization → Web portals (p. 346).

Linux

Setting	Default	Effect
App Name	Shared name	`Name` in the generated `.desktop` launcher
Comment	Empty	`Comment` — a short description
Categories	`Game;`	`Categories`, following the Desktop Entry spec. Example: `Game;ArcadeGame;`

The launcher entry is written beside the executable at build time, together with `app_icon.png` when the Window icon is set. Installing it is what gives the game an icon in menus and docks — and on Wayland, in the window itself.

Windows

Writes the executable's version resource. The desktop icon comes from the Window tab and is reused here.

Setting	Default	Written to
Product Name	Shared name	`ProductName` and `FileDescription`
Company Name	Empty	`CompanyName` (omitted when empty)
File Version	Shared version, padded to four parts	`FILEVERSION` and the `FileVersion` string
Product Version	Shared version, padded to four parts	`PRODUCTVERSION` and the `ProductVersion` string

Both version fields take up to four numbers from 0 to 65535. Type a shorter one and the dialog shows the padded result underneath — *Exports as 1.0.0.0* — so there is no guessing about what lands in the resource.

macOS

Setting	Default	Written to
App Name	Shared name	Bundle display name
Bundle Identifier	Shared identifier	`CFBundleIdentifier`
Version Name	Shared version	`CFBundleShortVersionString`
Build Number	Shared build	`CFBundleVersion`
Icon	None	Source image for the generated `AppIcon` asset set

Setting	Default	Written to
High DPI	Enabled	Allows high-DPI rendering

> **Bundle settings need a Source Code export**
> They apply to the Apple projects generated by a Source Code export. **Desktop** mode produces a standalone executable rather than an `.app` bundle, so it uses the Window icon and ignores the bundle fields.

iOS

Setting	Default	Written to
App Name	Shared name	Bundle display name
Bundle Identifier	Shared identifier	`CFBundleIdentifier`
Version Name	Shared version	`CFBundleShortVersionString`
Build Number	Shared build	`CFBundleVersion`
Icon	None	Source image for the generated `AppIcon` asset set
Hide Status Bar	Enabled	Hides the status bar while the app runs
Hide Home Indicator	Enabled	Requests hiding the home indicator
High Refresh Rate	Enabled	Allows 120 Hz displays where supported
Google AdMob	Disabled	Links Google Mobile Ads into the app, for the `AdMob` (p. 610) class. Without it, `AdMob` calls report errors
AdMob App ID	Google's sample app	`GADApplicationIdentifier`: the app ID from the AdMob console. Left empty, the export uses Google's sample app, which only shows test ads
Tracking Description	Empty	`NSUserTrackingUsageDescription`, the text of iOS's tracking permission prompt. Needed when the consent message asks for tracking permission
App Store Purchases	Disabled	Compiles in StoreKit 2, for the `InAppPurchase` (p. 618) class. Building it needs Xcode 16.3 or later

Apple bundle identifiers accept letters, digits and hyphens (`com.company.my-game`), and Apple versions accept at most three numbers — the dialog warns inline when a value does not fit.

AdMob App ID and **Tracking Description** apply only with **Google AdMob** on; otherwise the export removes both keys from `Info.plist`. The App ID has the form `ca-app-pub-…~…`, and the dialog warns when it gets an ad unit ID (with `/`) instead. Without **App Store Purchases**, the export leaves the StoreKit adapter out of the app. See Monetization (p. 337) for using both services.

Android

Setting	Default	Written to
App Name	Shared name	The launcher label in `strings.xml`
Package Name	Shared identifier	`applicationId` in `build.gradle`. Must be a valid Java package name
Version Code	Shared build	`versionCode`. A whole number from 1 to 2100000000
Version Name	Shared version	`versionName`, the visible version string
Launcher Icon	None	Fallback launcher icon
Adaptive Foreground / Background	None	Adaptive icon layers. Set **both** to get an adaptive icon; otherwise the launcher icon is used
Min SDK	`21`	`minSdkVersion`

Setting	Default	Written to
Target SDK	`36`	`targetSdkVersion`. Never below Min SDK
Orientation	Unspecified	The activity's requested orientation: Portrait, Landscape, Sensor Portrait, Sensor Landscape, or Full Sensor
Architectures	All four	The ABIs in the APK: `armeabi-v7a`, `arm64-v8a`, `x86`, `x86_64`
Permissions	None	`<uses-permission>` entries in `AndroidManifest.xml`
Allow Backup	Enabled	`android:allowBackup`
Keep Screen On	Disabled	Adds `FLAG_KEEP_SCREEN_ON` to the activity
Fullscreen	Enabled	Uses the fullscreen theme and hides the system bars
Google AdMob	Disabled	Compiles in Google Mobile Ads and Google's consent SDK, for the `AdMob` (p. 610) class
AdMob App ID	Google's sample app	The `com.google.android.gms.ads.APPLICATION_ID` entry of `AndroidManifest.xml`: the app ID from the AdMob console. Left empty, the export uses Google's sample app, which only shows test ads
Google Play Billing	Disabled	Compiles in Google Play Billing, for the `InAppPurchase` (p. 618) class

The permission list covers the common Android permissions by manifest name. Two shortcuts sit above it: **None** clears the list, and **Game Defaults** selects `INTERNET`, `ACCESS_NETWORK_STATE` and `VIBRATE`. Declaring a permission in the manifest is not the same as being granted it — runtime-permission prompts are still your own code's job.

> **At least one architecture**
>
> Clearing every ABI leaves nothing to build, so **OK** is disabled until one is ticked. A Source Code export made with no architecture selected fails with the same message.

Google's SDKs go into the export only when their service is enabled, so a game without ads or purchases ships without them. Both need newer Android versions than the default Min SDK: the export raises it to 24 for AdMob and 23 for Play Billing, with a warning, and compiles against SDK 35 or newer. When **Min SDK** is lower, the dialog says so under the checkboxes. As on iOS, it also warns when **AdMob App ID** holds an ad unit ID.

With AdMob on, also declare in the Play Console that the app contains ads and uses the advertising ID. See Monetization (p. 337) for using both services.

Where the values are stored

Everything on this dialog is written to `project.yaml`, and only values that differ from their default are written — a project that never opens the Platforms tab keeps a clean file.

Block	Holds
Top level	`name`, `canvasWidth`, `scalingMode`, `vsync`, `windowMode`, `windowTitle`, `windowIcon`, `assetsDir`, `luaDir`, `scriptDirs`, `cxxStandard`, `physics2D`, `physics3D`, `packNativeResources`, …
`application`	`name`, `identifier`, `version`, `build`
`web`, `linux`, `windows`, `macos`, `ios`, `android`	That platform's overrides only
`export`	Shader overrides and the graphic backends picked in the Export Window (p. 155)

Two things that used to live in `project.yaml` no longer do: the compiler kit (`cmakeCCompiler`, `cmakeCxxCompiler`, `cmakeGenerator`) and the parallel job count (`cmakeBuildJobs`). They are machine-specific, so they moved to the editor's `settings.yaml` — see Editor Settings (p. 68). Opening an older project migrates those four values automatically, once, and never over settings the current machine already has.

See also

- Editor Settings (p. 68) — the machine-level counterpart
- Project Workflow (p. 48) — project anatomy, scenes, bundles, save strategy
- Version Control (p. 65) — what to commit, and where per-user state is kept
- Export Window (p. 155) — where these settings are applied

Version Control

A Doriax project is designed to be committed. Scenes, bundles, scripts, assets and `project.yaml` are the game, and every one of them is text or an ordinary asset file.

What the editor keeps *out* of the repository matters just as much. Your open tabs, the place you left each viewport camera, your compiler path and the build tree are yours alone — committing them means a diff every time anyone opens the editor, and a conflict every time two people do. All of it goes under `.doriax/`, which is not committed.

What to commit

Committed	Not committed
`project.yaml`	`.doriax/` — everything below
`*.scene`, `*.bundle`	`CMakeLists.txt` (regenerated on every build)
Scripts (`.lua`, `.h`, `.cpp`)	`build/`, `.trash/`
Assets, `.material`, shaders	`.vscode/`, `.idea/`, `*.user`
`ProjectBuild.cmake`	`UserSettings.xml` written by the running game
`AGENTS.md`	Editor logs

`CMakeLists.txt` is generated from `project.yaml` and your scripts every time you press Play, so there is nothing in it to preserve. It also holds no path belonging to your machine: asset and Lua roots resolve from `${PROJECT_ROOT}`, and the paths that *are* machine-specific — the shader cache, the engine library — come from `.doriax/LocalPaths.cmake`, which the editor rewrites and which is not committed. See C++ Build Setup (p. 213).

Inside `.doriax`

Path	Holds
`user/workspace.yaml`	Your open tabs and their order, the selected scene, each scene's editor camera and viewport guides, the terrain brush
`user/build.yaml`	The compiler kit, build jobs and export folders for this machine
`user/rundata/`	Whatever the game writes through `data://` while you Play, so a playtest never dirties the working tree
`LocalPaths.cmake`	The machine-specific paths `CMakeLists.txt` includes
`engine-api/`, `generated/`, `scene_scripts.cpp`	The engine API snapshot and generated C++ glue
`build/`, `externalbuild/`	CMake build trees
`imported/`	Models converted from FBX, laid out like the assets directory (see FBX models (p. 251))
`editor.lock`	Locked while an editor has the project open, so a second editor cannot open it too. The lock ends when that editor closes, even if it crashes

The editor recreates all of it, so a fresh clone needs nothing but the editor opened once. Deleting `.doriax/` costs you the editor layout you had, a rebuild, and the FBX conversions, never project content.

The generated ignore rules

With **Version Control Files** enabled in Project Settings → General (p. 55) — the default — the editor keeps `.gitignore` and `.gitattributes` written at the project root and refreshes them whenever the project is saved.

`.gitignore` covers the table above. `.gitattributes` normalizes line endings so that scenes and `project.yaml` diff and merge the same way on every machine, and marks the common asset formats binary:

```
* text=auto eol=lf

*.scene   text eol=lf
*.bundle  text eol=lf
*.yaml    text eol=lf
...
*.png     binary
*.glb     binary
```

A file the editor did not generate is never replaced. Both files start with a `# Generated by Doriax Editor.` marker line, and the editor only rewrites one that still carries it. Lines you add to a generated file are kept: the editor writes its own rules first and your lines after them, in their order, so a `!` rule that re-includes something its rules ignore keeps working. Turn the setting off to stop the editor touching the files at all.

> **Keeping your own rules**
> Add your rules anywhere in the generated file; the next save moves them below the editor's. To manage the whole file yourself, delete its marker line — the editor then leaves it alone.

Git in the editor

With **Version Control Files** on and the project inside a Git repository, the editor shows where your work stands, as VS Code does:

- **Footer** — the branch at the far left, followed by `*` when there are uncommitted changes, `+` when some of them are staged, and `!` during a merge or rebase or with unresolved conflicts. A detached HEAD shows its tag, or the short commit. When the branch tracks a remote one, the commits behind and ahead follow, as `1↓ 2↑`. Hover them for the details.
- **Code Editor** — the lines that differ from the staged version of the file are marked in the gutter, with the change shown when you point at a mark. See Code Editor — Changed lines (p. 139).

The editor reads the repository with the `git` command, so Git has to be installed and on the `PATH`. It never stages, commits or switches branches; that stays in your Git tool, and the editor picks up what it did within a second or two. While the editor is in the background it only watches for staging, commits and checkouts, and it reads the rest again when you come back to it.

Working as a team

Scenes and `project.yaml` are line-based YAML, so Git merges them like any other text file. Two things make that reliable:

- **Scene ids are unique across branches.** Each scene carries an id that `startSceneId` and child scene references point at. Ids are drawn at random rather than counted up, so two people adding a scene on separate branches do not collide.
- **Nothing per-user is in the shared files.** Moving the camera, opening a tab or changing the terrain brush never marks `project.yaml` as modified.

Scenes still merge best when they are small. Splitting gameplay areas into separate scenes and composing them with child scene references keeps each file focused — see Scene files (p. 49).

If a scene file is missing from your checkout — a branch you have not fetched, or an asset still uploading — the editor keeps its entry in `project.yaml` and warns instead of quietly dropping the scene from the project. Saving the project will not remove it for everyone else.

A scene keeps the parts of each model as it was saved. After replacing an FBX, open every scene that uses it so its models reload, save them, and commit those scenes with the FBX. An export, on any machine, stops at a scene saved with an older version of an FBX it uses — see Export — FBX models (p. 160).

Migrating an existing project

Projects created before the split kept the camera, tabs and terrain brush in `project.yaml`. Opening one in a current editor copies those values into `.doriax/user/workspace.yaml`, so nothing is lost; the keys stay in `project.yaml` until you save the project, which is the only thing that rewrites it.

Two one-time steps for a repository that already has history:

1. `CMakeLists.txt` is usually already committed. Stop tracking it once:

   ```
   git rm --cached CMakeLists.txt
   ```

2. If the project has no `.gitignore` yet, open **Project → Project Settings**, confirm **Version Control Files** is on, and press **OK** to write it.

The first commit after that shows `project.yaml` shrinking to the settings that describe the game, and one extra newline at the end of each scene file. Both are expected and happen only once.

Editor Settings

Edit → Editor Settings holds the settings that belong to *this machine* rather than to the project: where your compiler and SDKs are, where exports go by default, how the editor itself renders, and how it talks to AI models and agents. They are saved with the editor's own settings, never to `project.yaml`, so they are not shared through version control and a teammate opening the same project keeps their own toolchain and keys.

Anything the game itself carries — application identity, canvas, window, directories, per-platform settings — is in Project Settings (p. 55) instead.

The dialog is a modal with five tabs and an **OK** / **Cancel** footer.

General

Setting	Default	Effect
Default Export Directory	Not set	Pre-fills the output folder in the Export Window for projects that have no remembered export directory yet
Editor VSync	Enabled	VSync for the editor's own frames
UI Scale	`100%`	Size of the editor text and panels, on top of the display scale the system reports

The default export directory is only a starting point: once you export a project, the folder you actually used is remembered per project *and per export mode*, and that remembered value wins the next time you open the Export Window.

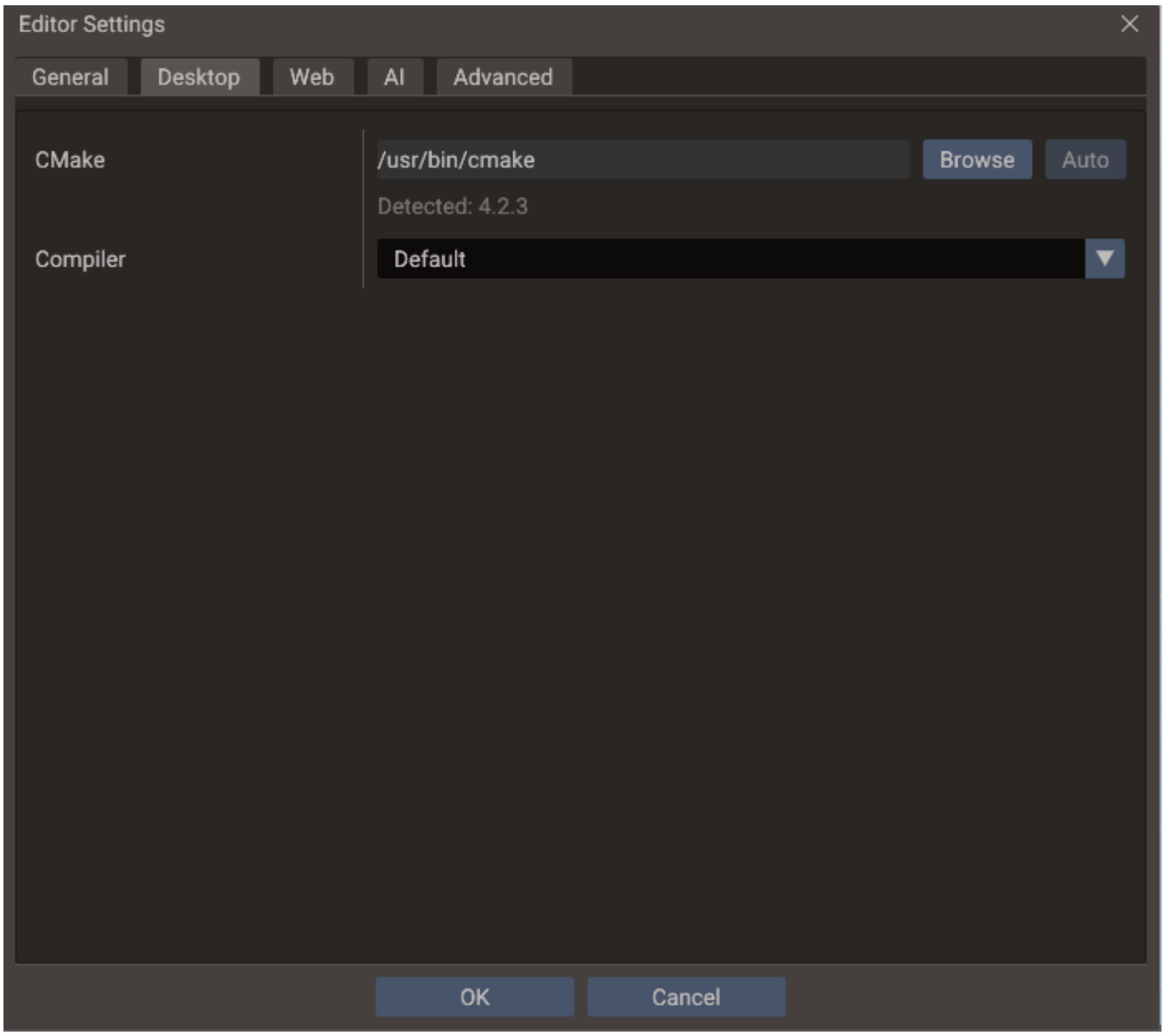
Editor VSync covers editing frames only. While a scene is running, frames follow the project's own VSync (p. 57) setting instead, so the two never apply at once. Toggling takes effect on the next frame with no restart, and an idle editor waits on the window system either way, so disabling it does not make the editor spin.

UI Scale goes from 50% to 300% and multiplies the display scale your system reports. On a screen the system scales to 150%, the default 100% keeps the editor at 150%, and 120% makes it 180%. The new size applies when you press **OK**, with no restart. Docked panels resize with the text, but a panel beside the scene view grows to at most 30% of the window's width, or 40% of its height for panels along the bottom, so the scene keeps its room.

On Linux the editor reads the display scale from `Xft.dpi`, which desktops such as GNOME, KDE, and Xfce set, or from the `GDK_SCALE` and `QT_SCALE_FACTOR` environment variables. When none of them is set it uses 100%, as GTK and Qt apps do. To make every app larger, raise the scale in your desktop's settings. To change only the editor, use **UI Scale**.

Desktop

The C++ toolchain used both when **playing** a scene with C++ scripts and when running a **Desktop** export.



Setting	Default	Effect
CMake	Auto-detect	Path to the `cmake` executable. **Browse** validates the pick by running it, and reports a folder with no working CMake instead of storing it. **Auto** returns to looking it up on `PATH`
Compiler	Default	The compiler kit used to build C++ scripts

The **Compiler** dropdown lists the kits the editor detected (GCC, Clang, MSVC/Visual Studio), each paired with a compatible CMake generator, and shows the resolved C and C++ compiler paths under the selection. Kits that cannot work are listed greyed out — hover one to read why. **Default** leaves the toolchain to CMake on Linux and macOS; on Windows it resolves to the best ABI-compatible detected kit, because a bare `cmake` there can pick a toolchain that cannot link the editor's engine.

If the kit stored for a project is no longer detected — a compiler that was uninstalled, or a project that came from another machine — the dropdown falls back to **Default** and warns that builds still use the stored kit until you press OK.

> **The compiler is stored per project, and remembered for new ones**
>
> Each project keeps its own kit, so a project that needs MSVC and one that needs MinGW can coexist. The kit is saved beside the project in `.doriax/user/build.yaml`, which is not committed, so a teammate opening the same project keeps their own toolchain. The kit you last chose also becomes the default for projects opened afterwards that have none, so in practice you set it once.

See C++ Build Setup (p. 209) for what each toolchain requires, the Windows ABI rules, and how to read a failing build.

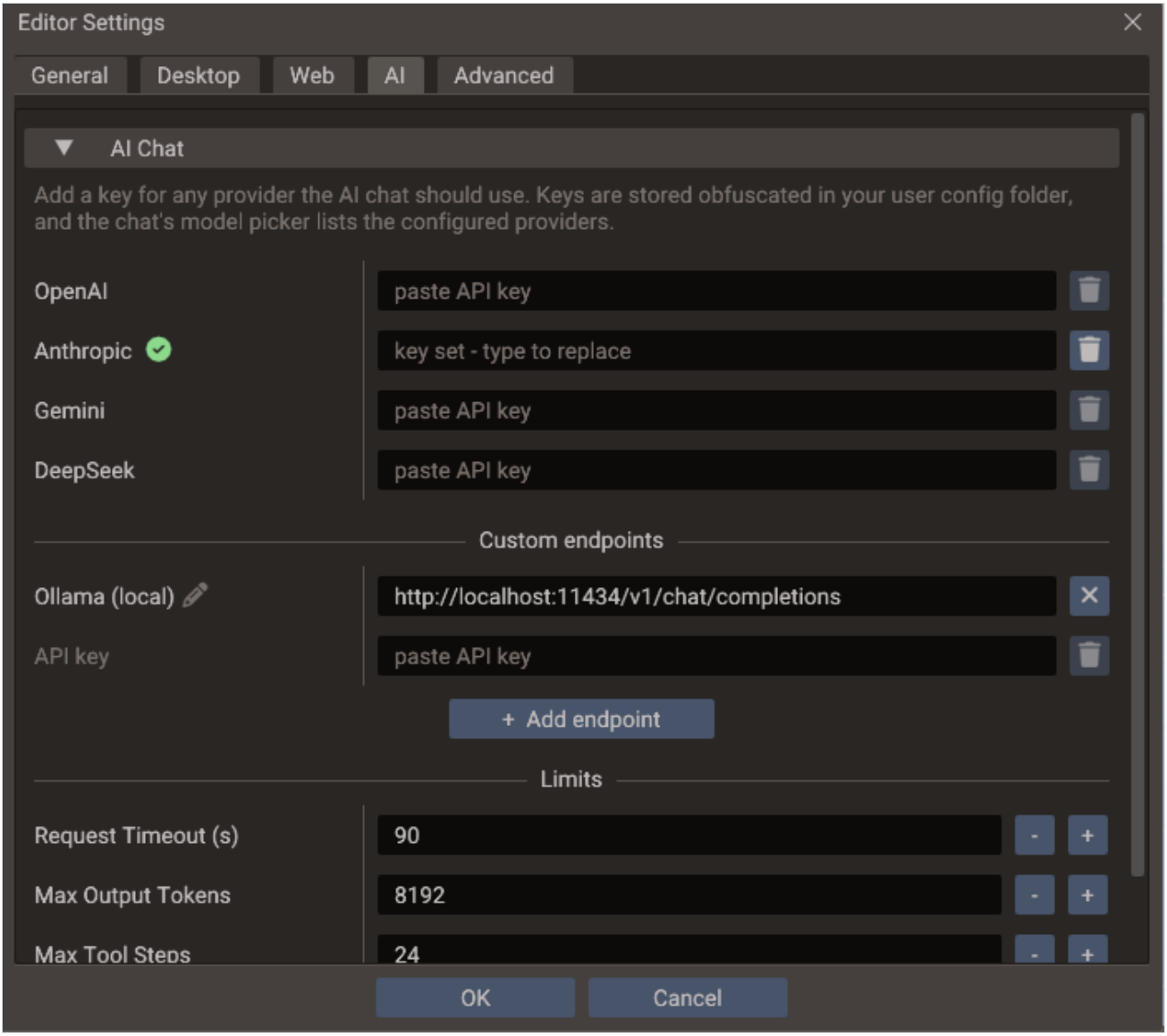
Web

Setting	Default	Effect
Emscripten SDK	Auto-detect	Root of your `emsdk` installation, used by **Web** exports. **Auto** returns to detecting it from the `EMSDK` environment variable or `emcmake` on `PATH`
Status	—	Whether the SDK was found, and where

The status line updates as soon as you pick a folder, so you can confirm the SDK before leaving the dialog. See Building for HTML5 (p. 760) for installing the SDK.

AI

Two collapsible sections: **AI Chat**, open by default, and **MCP Server** below it. The chat's gear button, and its prompts to add a key, open the dialog on this tab.



AI Chat

The keys and limits of the built-in AI Chat. The model and the approval mode stay in the chat, beside its message box.

Setting	Default	Effect
OpenAI, **Anthropic**, **Gemini**, **DeepSeek**	Empty	One API key per provider. A green check marks a stored key, typing replaces it, and the trash button deletes it at once
Custom endpoints	None	OpenAI-compatible Chat Completions URLs, each with its own key and model list. **Add endpoint** offers presets for OpenCode Zen, OpenRouter and a local Ollama, or a blank one. The pencil renames an endpoint; the cross removes it, and its key on **OK**
Request Timeout (s)	`90`	How long to wait for each model response. Raise it for slow local models
Max Output Tokens	`8192`	The longest reply a model may write in one turn, including a whole script written in one tool call
Max Tool Steps	`24`	How many tool-using model turns one message may take. Raise it if long tasks stop at the tool-step limit

A custom endpoint works without a key, since a local server usually needs none. The chat's model picker lists every provider and endpoint that is ready to use.

MCP Server

Runs the editor's MCP server (p. 172), which lets AI agents outside the editor — Claude Code, Codex, Gemini CLI — use the AI Chat's tools on the open project. That page covers connecting a client and the security model. The section starts collapsed.

Setting	Default	Effect
Enable Server	Off	Starts the server on `127.0.0.1`, now and whenever the editor starts
Status	—	**Running**, **Stopped**, or why the server could not start
Port	`3674`	The local port the server listens on
Allow Changes	On	Off leaves agents only the read-only tools
Token	Generated	The secret clients send with every request. Replacing it takes effect at once
URL	—	The endpoint, with a copy button
Clients	—	Ready commands that add the editor to Claude Code, Codex or Gemini CLI, each with a copy button

Advanced

Clear Shader Cache deletes the compiled-shader cache shared by every project on this editor version. Shaders are rebuilt on demand afterwards, so the only cost is the next build being slower. The button is disabled while a scene is playing.

Where the values are stored

Everything but the secrets goes to the editor's `settings.yaml`, beside `editor.log` — the FAQ (p. 769) lists the per-platform path.

Key	Holds
`cmake.path`	The CMake executable override
`emsdk.path`	The Emscripten SDK override
`export.default_dir`	The default export directory
`editor.ui_scale`	The UI scale as a multiplier, `1.25` for 125%

Key	Holds
`project_builds`	Per project: compiler kit and parallel build jobs, keyed by the absolute path of its `project.yaml`
`project_exports`	Per project and export mode: the last output directory used
`ai_assistant`	The AI Chat's provider, model, approval mode, custom endpoints and limits
`mcp_server`	The MCP server's `enabled`, `port` and `allow_changes`

API keys and the MCP token never go to `settings.yaml`. They are kept in `ai_keys.dat` in the same folder, obfuscated with a key derived from that folder's path, so the file does not work when copied to another machine.

The per-project entries are keyed by path, so moving a project through **Save Project As** re-keys them and your toolchain choice follows the project on this machine.

> **Build jobs are edited in the Export Window**
>
> The parallel job count lives in the same `project_builds` entry as the compiler, but it is edited in the Export Window's **Desktop** mode as **Build Jobs**. The value applies to Play builds too. See Parallel builds (p. 211).

Migrating from older projects

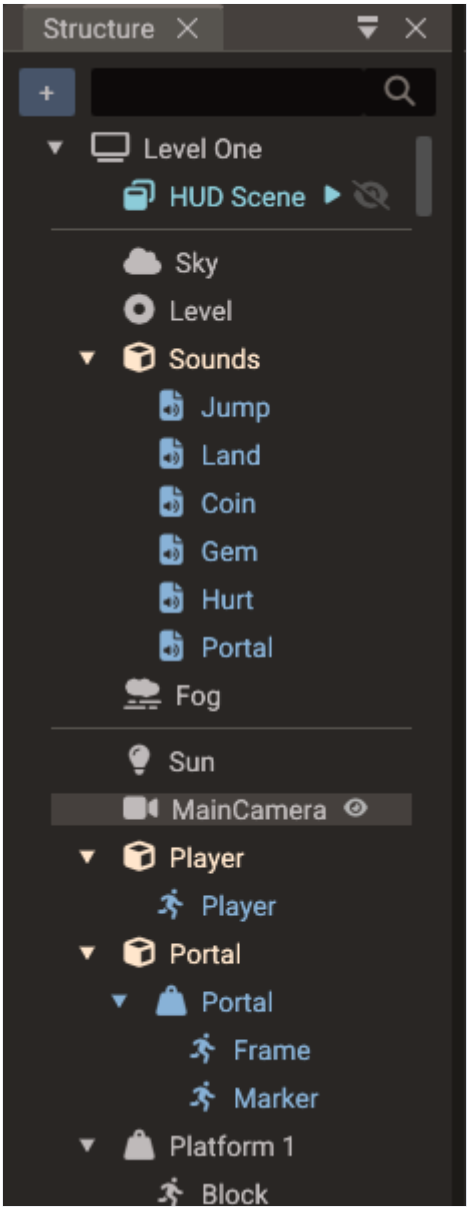
Editors before this dialog kept the compiler kit and job count in `project.yaml` as `cmakeCCompiler`, `cmakeCxxCompiler`, `cmakeGenerator` and `cmakeBuildJobs`. Opening such a project copies those values into `project_builds` once, and never over an entry this machine already has. The keys stay in the old file until the project is saved again, so an older editor can still read it.

See also

- Project Settings (p. 55) — the per-project counterpart
- C++ Build Setup (p. 209) — toolchain requirements and troubleshooting
- Export Window (p. 155) — where the SDK and toolchain settings are used
- MCP Server (p. 172) — connecting outside AI agents to the editor

Structure Panel

The Structure panel is the editor's tree view for scenes and entities. It is also the clearest way to understand Doriax's ECS model: everything shown under a scene is an entity, but only entities with a `Transform` participate in the spatial hierarchy.



Scene root

The top node is the selected scene. Child scenes are shown before entities. If a child scene is expanded inline, its entities are shown under that child scene node.

Child scenes

A scene can reference other scenes as **child scenes** so they load and run together as one scene stack (p. 184). Child scene nodes appear above the parent's entities, tinted soft teal to set them apart.

Add a child scene in one of two ways:

- Right-click the scene root and choose **Add child scene → *SceneName*** (the submenu lists every other scene that is not already attached and does not already contain this one).
- Drag a `.scene` file from the Resources Browser (p. 106) onto the scene root.

A scene can't end up inside itself, directly or through other child scenes. An add that would create such a cycle is refused with an error in the Output panel. So is an undo or redo that would bring one back after another scene changed, and the history then stays where it is.

Right-click a child scene node for its menu:

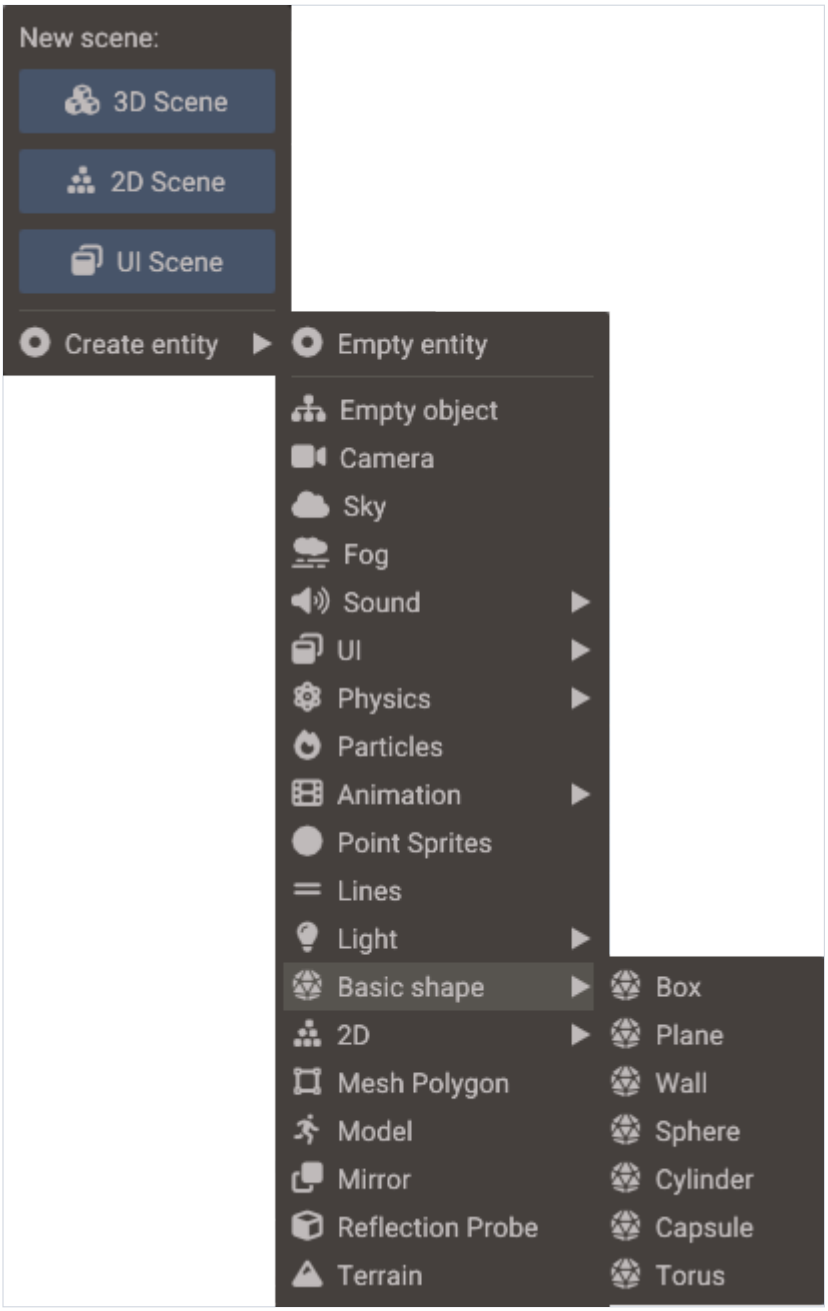
Action	Effect
Start active	Toggles whether the child scene is added to the engine automatically when the parent loads. On (the default) means it runs immediately; off means it is built with the parent but hidden, ready for `SceneManager.addChildScene` to show it with no loading cost. A scene does not have to be a child scene to be added at runtime — being one is what makes it *preloaded*.
Remove child scene	Detaches the reference (it does not delete the scene file).

Each child scene row also shows a small **play** or **pause** mark next to the name: play means **Start active** is on; pause means it starts inactive. Hover the mark for a tooltip. Click the **eye icon** on a child scene node to load it *inline* — its entities appear nested under the node so you can view and edit them in the parent's context. This is an editing convenience and does not affect runtime behavior.

The entities shown inline still belong to the child scene. Dragging one only creates a reference to it: drop it on a script's entity property in Properties, or on a script open in the Code Editor (p. 137). Drop targets that would change the parent scene refuse it, with the tooltip *Entity belongs to another scene*. That covers reparenting it in this tree, saving it as a bundle in the Resources Browser, and assigning it to a component field such as a joint's bodies. To make those changes, open the child scene itself.

Order matters: scenes render as layers, with the main scene at the bottom and each child scene drawn on top of the ones listed above it — so the last child scene is the topmost layer. The order follows the order you added the child scenes; to change it, remove them and add them back in the sequence you want (top-most last). See Child scenes (p. 184) in the manual for the full runtime model.

Empty entity vs empty object



Open the create menu with the **+** button at the top of the panel (**Create entity**; the same menu starts a new 3D, 2D, or UI scene), by right-clicking empty space in the panel, or with **Create child** in an entity's right-click menu, which creates the new entity under it.

The create menu has two intentionally different entries:

Entry	Components added	Meaning
Empty entity	None	A pure entity ID. Use it for logic, global scripts, non-spatial data, or components that do not need a transform.
Empty object	`Transform`	A spatial entity. It can be positioned, parented, rendered, and shown in the hierarchy.

This distinction matters because the entity itself owns nothing. Components decide what the ID can do.

The rest of the create menu adds ready-configured entities (camera, light, sky, fog, sound, **mirror**, **reflection probe**, **water**), basic shapes, 2D and UI objects, physics bodies, and more — each one is just an entity with the right components already attached.

The **2D** submenu includes **2D Light** and **2D Occluder** for the 2D lighting system — see 2D Graphics — 2D lighting (p. 244).

Basic shapes

Basic shape creates a mesh entity with procedural geometry: Box, Plane, **Wall**, Sphere, Cylinder, Capsule, and Torus. A **Plane** lies flat (normal points up, `+Y`); a **Wall** stands upright facing the camera (normal points `+Z`) — handy for walls, backdrops, and mirror surfaces. You can change a mesh's geometry later from the **Create Shape** dropdown in Properties (p. 90), unless the mesh comes from a model file.

Mirror

Mirror is a one-click convenience entry: it creates a Wall with a Mirror component (p. 100) already attached, giving an upright planar-reflection surface with no camera or texture setup. See Rendering Pipeline — Mirrors (p. 269).

Reflection Probe

Reflection Probe creates an entity with a Transform and a Reflection Probe component (p. 100): a box-shaped volume that gives the meshes inside it a local reflection environment instead of the global sky. Size the box to the room it represents in Properties (p. 100). See Rendering Pipeline — Reflection probes (p. 263).

Water

Water creates an entity with a Transform and a Water component (p. 100): a 50 × 50 animated water surface at the entity's height, with waves, ripples, refraction, and shore foam on by default. Move the entity to set the water level, and adjust its size and waves in Properties (p. 100). See Rendering Pipeline — Water (p. 270).

Water Exclusion

Water Exclusion creates an entity with a Transform and a Water Exclusion component (p. 101) set to a 1 × 1 × 1 box: a volume the water stays out of, for a cabin, a cave, or a diving bell. To keep the water out of a boat, add the component to the boat itself instead; its default **Hull** shape wraps the boat's mesh.

Hierarchical area

Entities with `Transform` appear in the hierarchy area. Their parent-child relation is stored in `Transform::parent`, and their order is managed by the scene registry. Moving a parent updates child world transforms through that parent chain.

Common entities in this area include objects, sprites, models, cameras, lights, 3D sounds, physics bodies, UI widgets, points, lines, terrain, and mesh polygons.

Tree marks and collapse

Mark	Meaning
Clone icon	The entity has an Instanced Mesh (p. 94) with one or more instances (tooltip shows the count)
Play / pause	Child scene **Start active** on / off (see Child scenes)

Imported **Model** entities with child nodes start **collapsed** once their children finish loading, so multi-mesh GLTFs and full node trees do not flood the tree. Bones also start collapsed. Expand a row when you need to edit children.

When a parent is collapsed in Structure, clicking its hidden children in the Scene view (p. 83) selects the nearest expanded ancestor instead — so picking in the viewport matches what you can see in the tree. Expand the model (or other parent) if you need to select a specific child mesh or node.

GLTF files with animation clips, or more than one skin, import the **full node tree** under the Model (helpers, joints, and mesh nodes). Static multi-mesh files still create one child mesh entity per mesh node. See 3D Graphics — GLTF node hierarchy (p. 249).

Organizing the parts of a model

The child mesh entities of a multi-mesh GLTF belong to their Model, but you can arrange them inside it. Parts that can move are drawn in the normal text colour, and hovering one shows *Drag to organize the parts of this model*. Parts that stay put are greyed out, and their tooltip says why.

What you can move:

- Every part of a **static** model (no animation clips, at most one skin, not merged).
- The **skinned** parts of an animated model. A skinned mesh is placed by its joints, so moving the part changes only the tree, never the geometry.
- A **rigid** part the file does not animate — not directly and not through a parent node — such as a loose prop exported next to a character.
- A plain entity (an **Empty object**) holding parts. Drag it into the model, then drag parts under it to group them; while it holds parts it counts as part of the model.

What stays where the file put it, with the tooltip reason:

Tooltip	Why
Joints cannot be reparented	The joint's world transform feeds skinning, and its keyframes are relative to its parent. Bones can still receive drops.
Model nodes cannot be reparented	Transform-only helper nodes of a full node-tree import.
Animated parts cannot be reparented	The part has its own translation, rotation, or scale keyframes, which are relative to the file's parent.
Parts inheriting animation cannot be reparented	A rigid part under an animated node moves with that node; pulling it out would stop its animation.
Parts holding joints cannot be reparented	The part has joints below it in the file.

Where you can put a movable part:

- Under another part, or under a group entity inside the model.
- On a **bone** or any other node of the model — the way to hang a prop on a hand. Bones also accept plain entities (lights, cameras, empties, other models), so you can attach anything to a joint by dropping it there.
- On the Model row to put it back under the root, or between siblings to reorder.

Reparenting keeps the world placement, so a drop never moves geometry; position the part afterwards with the gizmo if it should sit somewhere else. The file is not touched: the arrangement is saved with the scene, and reopening the scene or re-assigning the same model file keeps it (see 3D Graphics — Reloading a rearranged model (p. 250)).

Parts always stay inside their model — a drop on the scene root, on an entity outside the model, or inside a nested model is refused. Parts and the groups holding them are also deleted and duplicated only together with the whole model, so **Delete** and **Duplicate** act on the Model row instead.

Right-click the Model and choose **Reset mesh parenting** to move every part back under the node the file gives it (the model root for a static model) in one undoable step. A part that is locked away from its file node — after a re-export changed the file, for example — can always be dropped back on that node even though it cannot go anywhere else.

Merge static model

Multi-node GLTF models normally keep geometry on child entities (the full node tree when the file is animated or has several skins, otherwise one child mesh per mesh node). Instanced Mesh (and other same-entity mesh features) only draw geometry that lives on the root entity, so those models need a flatten step first.

Right-click a Model entity in Structure:

Action	Effect
Reset mesh parenting	Moves every mesh part back under the node the file gives it (available while parts are rearranged). Undoable.
Merge static model	Bakes child mesh transforms into the root `MeshComponent`, removes the child mesh entities, and sets the model's `mergeStaticMeshes` flag. Undoable.
Restore model mesh children	Reloads the model with the hierarchy restored (available after a merge).

Merge is only available for static multi-mesh models. It is disabled (with a tooltip) when the model is skinned, animated, has morph targets, has fewer than two mesh nodes, or would exceed the root submesh limit. It is also disabled while the parts are rearranged — choose **Reset mesh parenting** first. See 3D Graphics — Merging static model meshes (p. 250).

Non-hierarchical area

Entities without `Transform` appear separately before the transform hierarchy. They are valid scene entities, but they do not have a spatial parent or local/world transform.

Typical non-transform entities include:

Entity type	Why it may not need `Transform`
Empty entity	Logic-only entity or script host
Sound source	Non-spatial audio
Sky/Fog	Scene environment data
Joints	Constraint data linking physics bodies
Actions/animations	Time-based behavior targeting another entity
Particles action	Playback behavior targeting another entity

An entity can be selected and inspected even if it is not in the transform hierarchy. Add a `Transform` if it should become spatial or parentable.

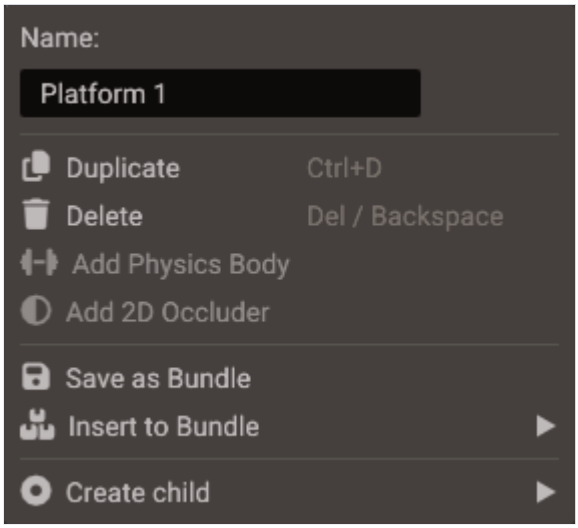
Search

Type in the field at the top of the panel to filter the tree by entity name, or by id for an entity without one. Matching ignores case unless you turn on **Match Case** from the magnifier button next to the field. Matching rows stay visible together with their ancestors, so you can see where each one sits.



Renaming

Right-click an entity or the scene root and edit the **Name** field at the top of its menu, or select the row and press **F2** to edit the name in the Properties window (p. 90). A child scene node keeps its name read-only here; open the child scene to rename it.



Drag and drop rules

Reparenting is a transform operation. Dragging an entity under another entity only makes sense when the moved entity has `Transform`. Non-transform entities can still be reordered in their separate area or associated virtually with a target, such as an action targeting a transformed entity.

Entities the editor generates for a component — bones, skeletons, imported animations, the label of a button, and so on — are greyed out and can only be reordered among their siblings. The mesh parts of a model are the exception: the ones the file does not animate can be regrouped anywhere inside their own model, bones included (see Organizing the parts of a model).

Drag and drop also crosses window boundaries:

Drag	Drop	Result
Entity (or selection) from Structure	Resources Browser	Saves the hierarchy as a `.bundle` file and replaces it with a bundle instance
Entity from Structure	Entity-reference field in Properties	Assigns the entity to that field (which fields accept another scene's entity (p. 105))
Entity from Structure	Script file in the Code Editor	Inserts an entity reference property (details (p. 137))
Action entity from Structure	Animation Timeline (p. 130) tracks	Adds an action frame at the dropped track and time
`.bundle` file from Resources	Scene root or an entity with `Transform`	Creates a bundle instance there
`.scene` file from Resources	Scene root	Adds it as a child scene

Of these targets, an entity shown inline under a child scene node is accepted only by a script entity property and the Code Editor. See Child scenes.

Bundles in the tree

Right-click any entity outside a bundle and choose **Save as Bundle** to turn its hierarchy into a new `.bundle` file (saved in the folder the Resources Browser is showing) — the same operation as dragging it onto the Resources Browser.

A bundle instance appears as a root node with its member entities nested under it. Editing members edits the bundle (and every other instance); right-click menus on bundle nodes let you **Insert into bundle**, **Remove from bundle**, or move outside entities into an existing bundle with **Insert to Bundle**. See Bundles (p. 111) for the complete workflow.

While playing

Entities the running game creates (objects a script spawns, models it loads, bundles it instantiates) are listed too, drawn in green. Each one sits under its parent, or at the scene root when that parent is not listed, such as the editor's view camera in a scene without a camera of its own. Entities without a `Transform` go in the non-hierarchical area as usual.

Scripts often create entities without a name, and their rows show the entity id instead, like `(1004)`, in a dimmer green. Set the object's `name` (p. 711) in the script to show a name.

These rows are for inspecting, not editing, because **Stop** destroys the entities behind them:

- Selecting one shows its components in Properties (p. 90), all read-only.
- It cannot be renamed, deleted, duplicated, dragged, given children, set as the main camera, or saved into a bundle.
- It can be picked in the Scene view (p. 83) only while the camera is detached (p. 89); otherwise select it here.

The list refreshes twice a second, so a new entity can take up to half a second to appear. One the game destroys disappears at once and leaves the selection, and **Stop** removes the rest.

Terrain foliage (p. 312) chunks are engine data and are never listed. Only the scene you pressed Play on lists runtime entities; a child scene expanded inline keeps showing its authored entities.

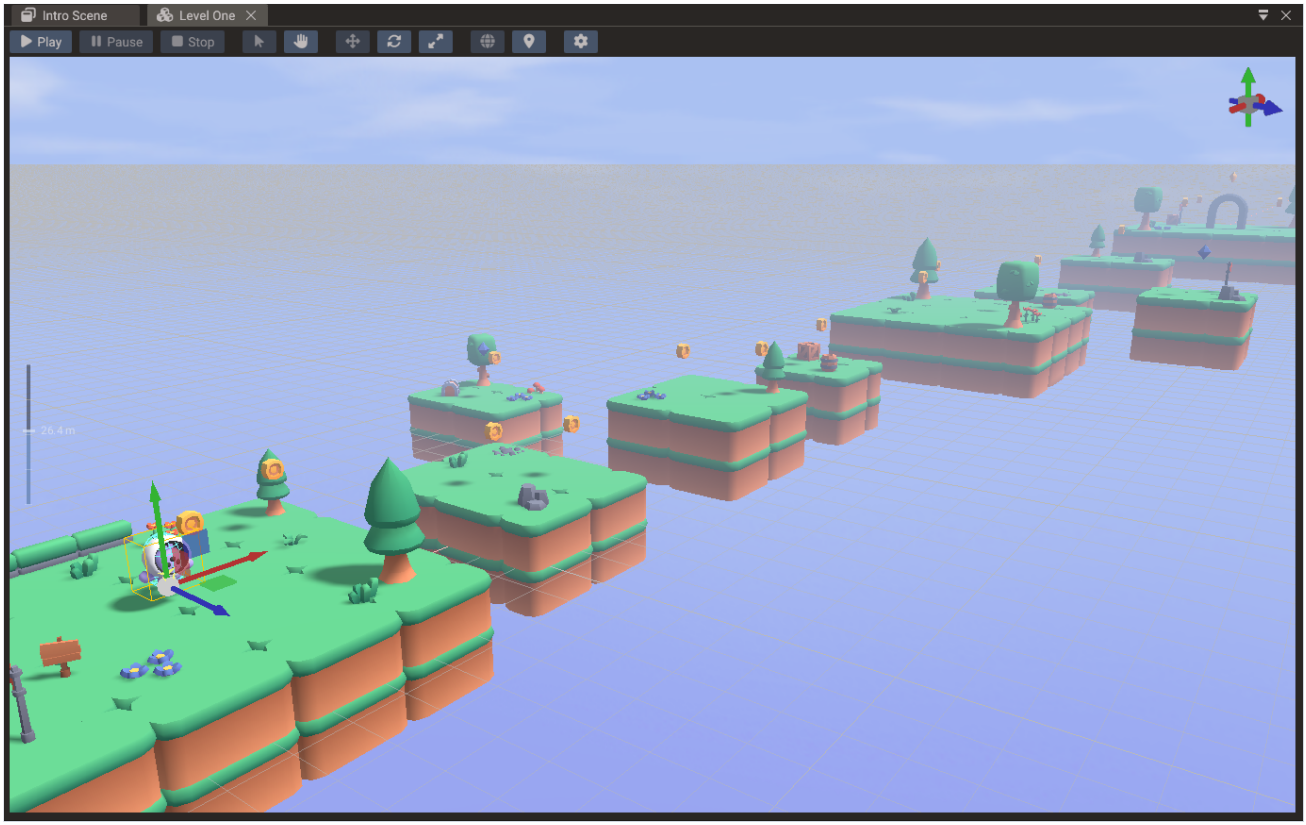
Practical model

Use the Structure panel as a quick diagnostic:

- Entity appears in the tree hierarchy: it has `Transform`.
- Entity appears above the hierarchy: it has no `Transform`.
- Entity cannot be parented: add `Transform` or choose Empty object instead.
- Visual entity is missing from the hierarchy: check whether `Transform` was removed.
- Entity drawn in green while playing: the game created it. It is read-only and disappears at **Stop**.

Scene View

The **Scene view** is the main canvas for authoring scenes. It renders the current scene using editor cameras and overlays gizmos, selection highlights, and tool guides on top. You place, rotate, scale, and test entities here, and you see the result of component changes made in the Properties window immediately.



Navigation

Action	Input
Orbit	Right-click drag (3D)
Pan	Middle-click drag, or Shift + right-click drag
Zoom	Mouse wheel
Focus on selection	**F** key
Reset camera	Home key or double-click empty space

In 2D mode the camera is orthographic; orbit is replaced with a simple pan.

Gizmo tools

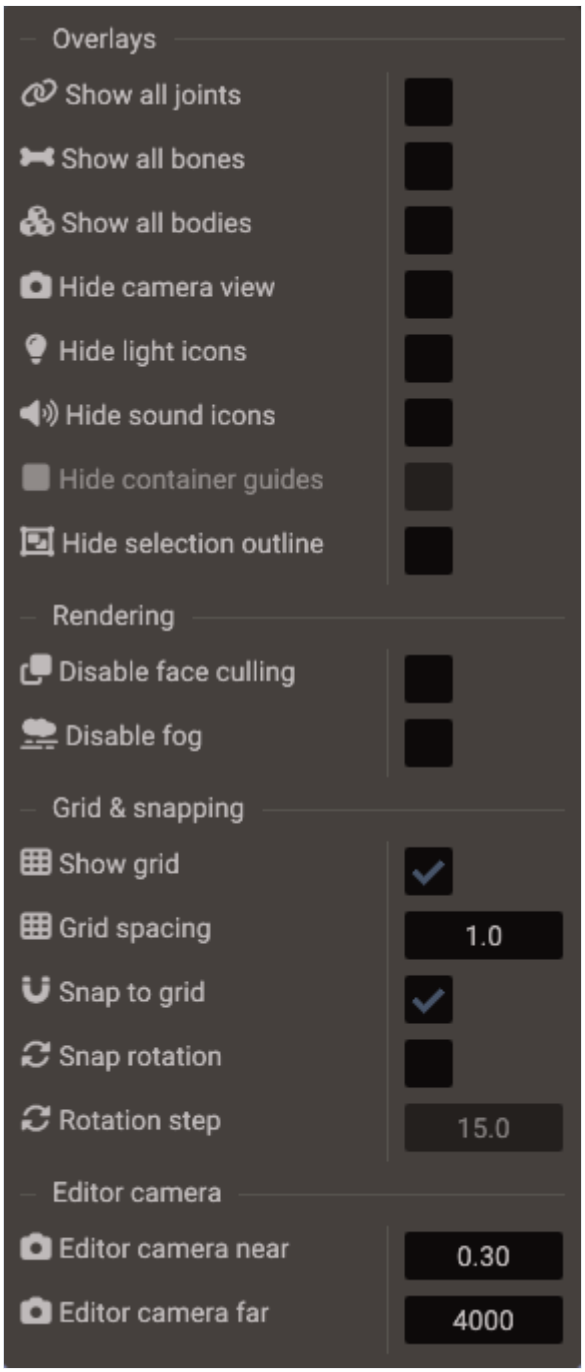
Select a gizmo mode from the toolbar or with the keyboard:

Gizmo	Key	Purpose
Translate	W	Move selected entities along axes or planes
Rotate	E	Rotate selected 3D entities
Scale	R	Resize entities uniformly or per axis
Object2D	—	Manipulate 2D objects in canvas space (position and size)
Anchor	—	Edit UI anchor points and layout bounds

Hold **Ctrl** while transforming to snap to the configured grid interval. Hold **Alt** to move in world space instead of local space.

Viewport settings

Open the **gear** button on the scene view toolbar to toggle display and snap options for the current scene. They are saved per user in `.doriax/user/workspace.yaml`, not in the scene file. They are grouped into **Overlays**, **Rendering**, **Grid & snapping**, and **Editor camera** sections.



Setting	Applies to	Purpose
Show all joints / **Show all bones** / **Show all bodies**	All scene types	Draw every physics joint, skeleton bone, or collision shape, not only the selected entity's. Unavailable while playing
Hide camera view	All scene types	Hide camera icons and frustums

Setting	Applies to	Purpose
Show full camera frustum	All scene types	Draw camera frustums out to their far clip plane, not just a short preview
Hide light icons	3D	Hide light icons
Hide sound icons	All scene types	Hide sound source icons
Hide container guides	2D / UI	Hide the outlines of UI containers
Show origin axis	2D / UI	Draw the axes through the scene origin
Hide selection outline	All scene types	Hide the outline drawn around selected entities
Disable face culling	3D	Draw back faces in the editor view. Unavailable while playing
Disable fog	All scene types	Draw the editor view without the scene's Fog. Unavailable while playing
Disable underwater	3D	Draw the editor view as if the camera were never inside a Water (p. 100), so the underwater fade does not cover it when the camera dips below the surface. Unavailable while playing
Show grid	All scene types	Draw the editor grid overlay
Grid spacing	All scene types	Interval used by the grid and by **Snap to grid**. In 2D and UI scenes it is shown while the grid is on
Snap to grid	All scene types	Snap transforms to the configured grid spacing
Snap tile	2D / UI	Snap Tilemap tiles to their own width and height so they pack edge-to-edge
Snap rotation / **Rotation step**	All scene types	Snap rotations to the configured step (degrees)
Editor camera near / **Editor camera far**	3D	Clip planes of the editor camera

Snap tile takes priority over **Snap to grid** while you move or resize an individual tile inside a Tilemap. Turn it on when laying out floors, walls, or other tiled content that should sit flush without relying on the global grid.

Selection

Click an entity in the viewport to select it; the Properties window updates immediately. Hold **Ctrl** and click to add to or remove from a multi-selection.

Selection respects the Structure panel (p. 76) expand state: if the hit entity is under a **collapsed** parent, the editor selects that visible ancestor instead of a hidden child. Expand the parent in Structure when you need to pick a specific child mesh or node (common with multi-node and animated models).

For fine-grained selection in dense scenes, use the **Structure panel** — clicking a row there selects the entity without needing to click through overlapping objects.

Entities the game creates while playing can only be picked in the viewport while the camera is detached; otherwise select them in the Structure panel (p. 80).

Drag and drop from the Resources Browser

Files dragged from the Resources Browser (p. 106) can be dropped directly into the viewport. The editor shows a live preview while you hover, and the change is only committed (undoably) when you release the mouse:

Creating entities

File	Scene type	Result
Model (GLTF/GLB/OBJ/FBX)	3D	Creates a Model entity at the drop position, named after the file
Image	2D, on empty space	Creates a **Sprite** entity at the drop point, sized to the image
Image	UI, on empty space	Creates an **Image** widget at the drop point, sized to the image

While dragging an image over empty space, a half-transparent ghost of the texture follows the cursor so you can place it precisely before dropping.

Modifying the entity under the cursor

Dropping a file *onto* an existing entity assigns it instead of creating something new. The hovered entity previews the change live; moving away before releasing restores the original value:

File	Target entity	Result
Image	Mesh-based entity (sprite, mesh, model)	Sets the base color texture of the material
Image	UI widget	Sets the widget texture
Material (`.material`)	Mesh-based entity	Links all submeshes to the material file (live preview while hovering)
Font (TTF/OTF)	Text entity	Sets the text font

This makes texturing a scene fast: drop a texture onto each model to assign it, or drop a saved material onto several meshes to keep them consistent.

Sprites and UI images that get their texture from a drop clamp it at the edges (`CLAMP_TO_EDGE`). With the default repeat, linear filtering would pull the opposite edge into a thin line along the border. To tile the image instead, set the wrap back to repeat in the Texture Settings (p. 92) popup.

Shared materials across meshes

When you drop the same `.material` file onto different mesh entities, each submesh **links** to that file. Editing the material in Properties (while linked) or saving the `.material` file on disk updates every linked mesh.

To create a new shared material from an existing look:

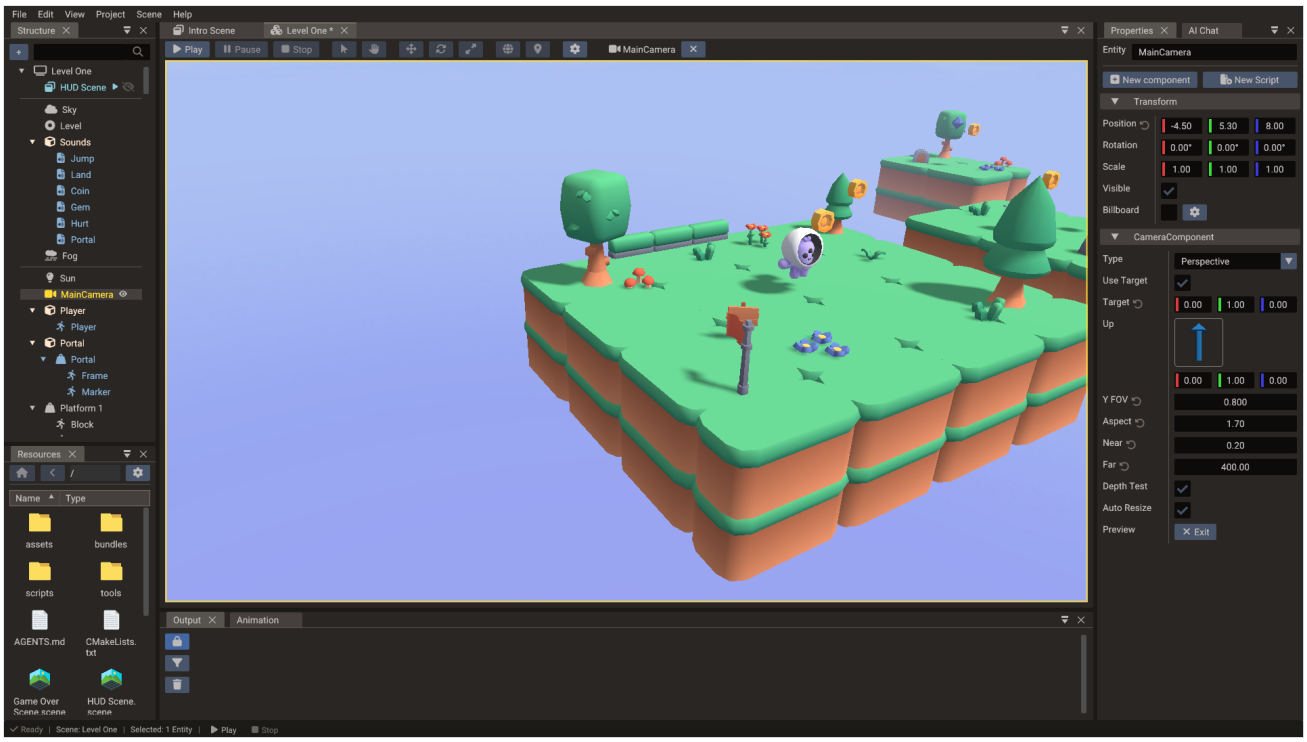
1. Tune the material on one mesh in Properties.
2. Drag its **material preview** into the Resources Browser to create `Material.material`.
3. Drag that file onto other meshes in the Scene view.

Use the unlink control in Properties when a mesh needs a one-off variation copied from a shared file.

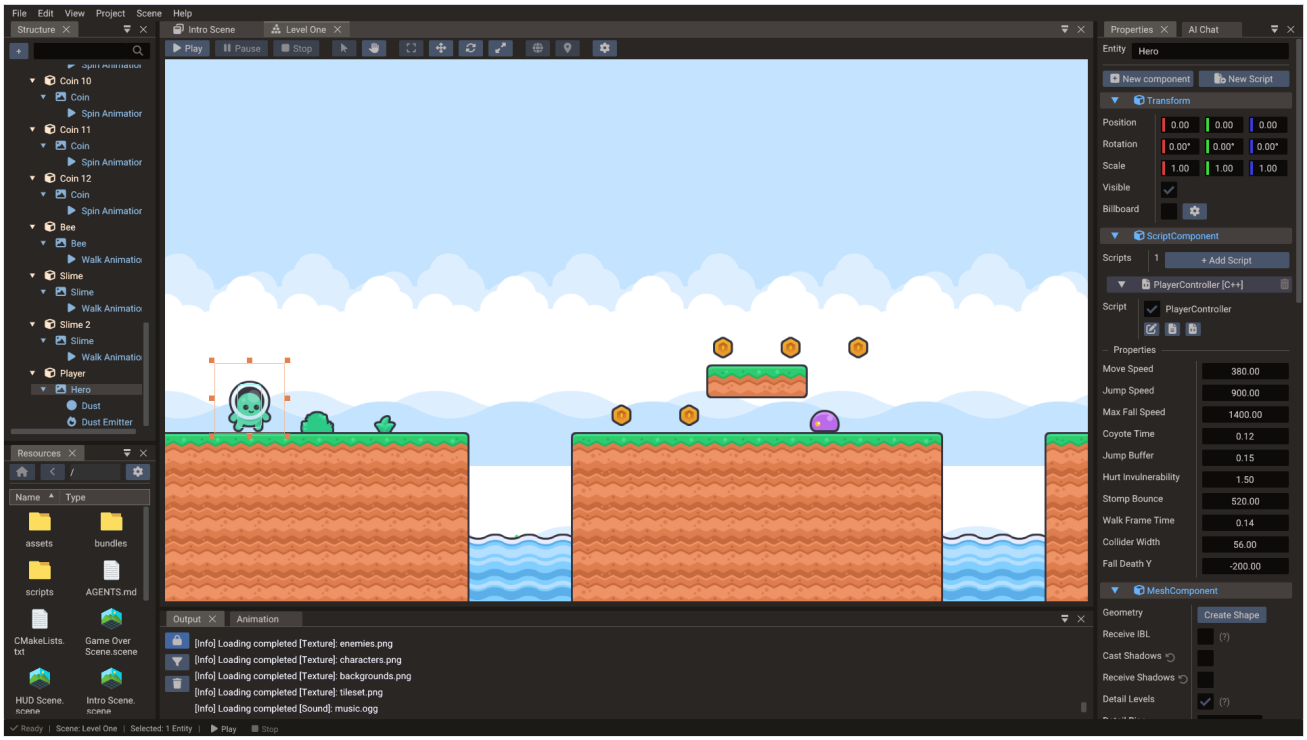
> **Bundles and scenes drop on the Structure panel**
>
> `.bundle` and `.scene` files are instanced by dropping them on the Structure panel (p. 73), not the viewport — the tree controls where the instance is parented. See Bundles (p. 111).

Viewport camera vs game camera

The **editor camera** is only for authoring navigation. The **game camera** is a `Camera` entity you place in your scene; it defines what the player sees at runtime. Always verify gameplay framing with the game camera preview before testing: select the camera and click **View** on the **Preview** row of its CameraComponent. The toolbar shows the camera's name while you look through it; click **Exit** (or the **×** beside the name) to return to the editor camera. While the game runs, the viewport shows the game camera; detach the camera to look around with the editor camera.

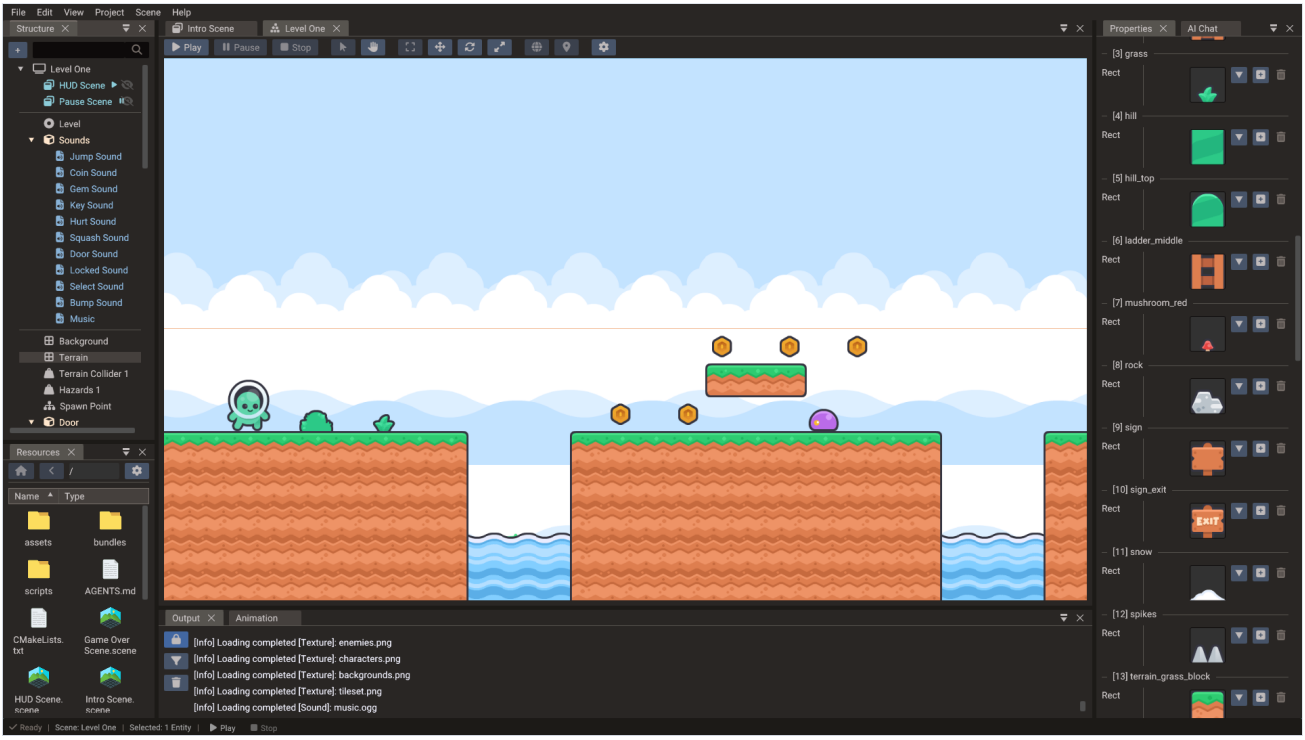


2D and tilemap editing



In a 2D scene the editor uses orthographic projection and a canvas overlay. Sprites, tilemaps, UI widgets, and polygons should be positioned in a consistent logical coordinate system so that canvas scaling stays predictable across different screen sizes.

Tilemap cells can be painted directly in the scene view when a Tilemap entity is selected and the tile-paint mode is active.



Placing and grouping tiles

Drag a tile rect from the Tilemap palette onto the viewport to place it. With **Snap tile** enabled in the viewport settings:

- Dropped tiles snap to multiples of their own size in the tilemap's local space.
- Dragging or resizing a selected tile with the Object2D gizmo snaps the same way, so neighboring tiles line up edge-to-edge.

Use **Snap to grid** instead when tiles should align to the scene grid rather than to each other. See also Tileset Slicer (p. 119).

2D lights and occluders

2D lights show a **bulb icon** in the viewport — click the icon to select the light and drag it to move it. While selected, the light's range is drawn as a circle.

Selecting an entity with an **Occluder2D component** shows its outline. For polygon occluders, each point gets a small handle:

- **Click a handle** to select that point (it also highlights in the Properties point list), then **drag the move cross** to reposition it. The drag is a single undo step and respects snap-to-grid.
- **Click the occluder body** to go back to entity-level selection, or click another handle to switch points.

Standalone occluder entities (no sprite) are selected by clicking anywhere inside the polygon's bounds. See 2D Graphics — 2D shadows (p. 245).

2D physics shapes

Selecting an entity with a **2D Physics Body** draws its shapes. Polygon and Chain shapes get a handle at each vertex, and they work like occluder points: click a handle to select the vertex, then drag the move cross to move it. The drag is a single undo step and respects snap-to-grid. Click elsewhere on the entity to go back to entity selection.

On a tilemap, a vertex handle takes the click before the tile under it. Shapes with the **Entity Tilemap** source have no handles, because their boxes come from the tiles. See Tileset Slicer (p. 121).

The **Box** and **Centered Box** buttons under a polygon's **Reset shape** size the box to the entity's sprite, UI layout, or tilemap.

3D scene editing

3D scenes use perspective projection and full 3D gizmos. Switch between the editor camera and the game camera preview to check framing. Use the **F** key to center the view on a selected entity.

Editing movement paths (TranslateTracks)

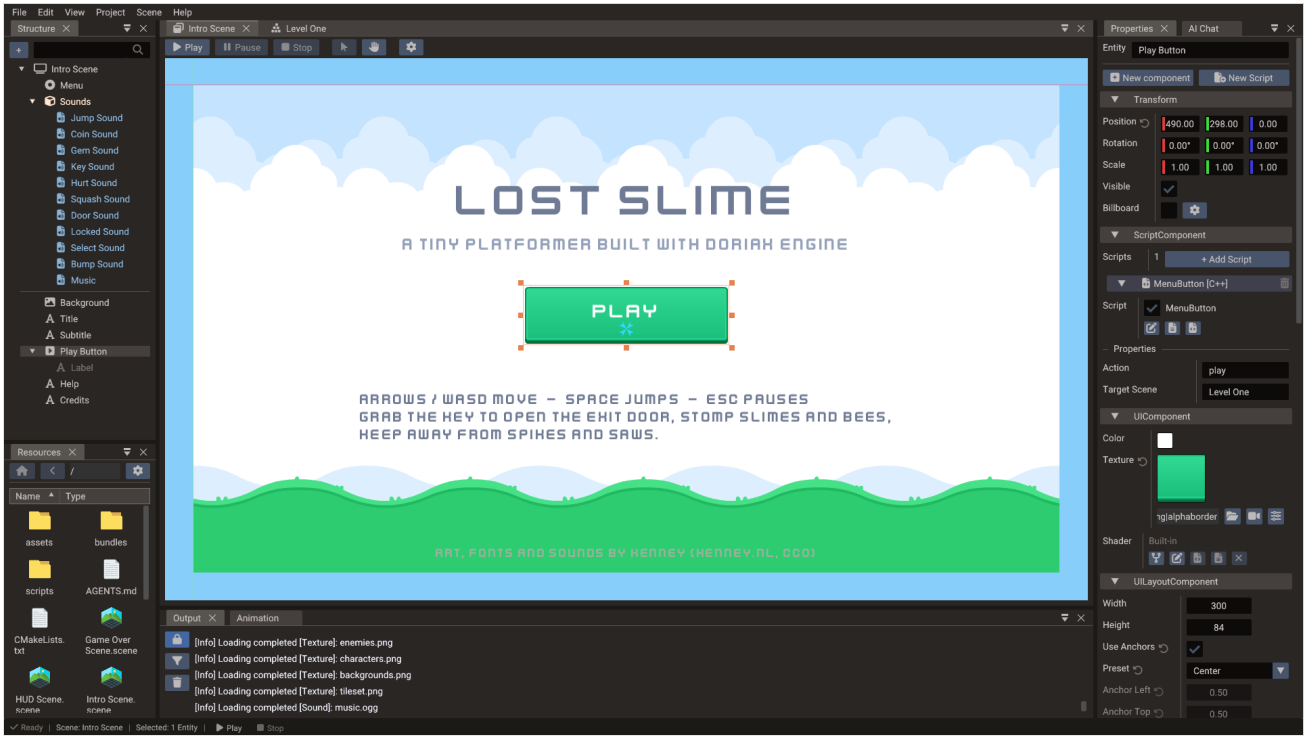
Selecting a **TranslateTracks** entity draws its movement path in the viewport: a polyline through the keyframe positions with a handle at each point, shown in the space the values play back in (the action target's parent space). The path renders on top of scene geometry, so waypoints lying on floors or inside meshes stay visible.

- **Click a handle** to select that point — it highlights and the translate gizmo (move cross in 2D scenes) jumps to it.
- **Drag the gizmo** to move the point. The drag is a single undo step, respects snap-to-grid, and updates the track's `values` list live in the Properties window.
- **Click empty space** or another entity to return to normal selection.

Keyframe *times* and per-segment easing are edited in the **Properties** window; see Keyframe tracks (p. 303).

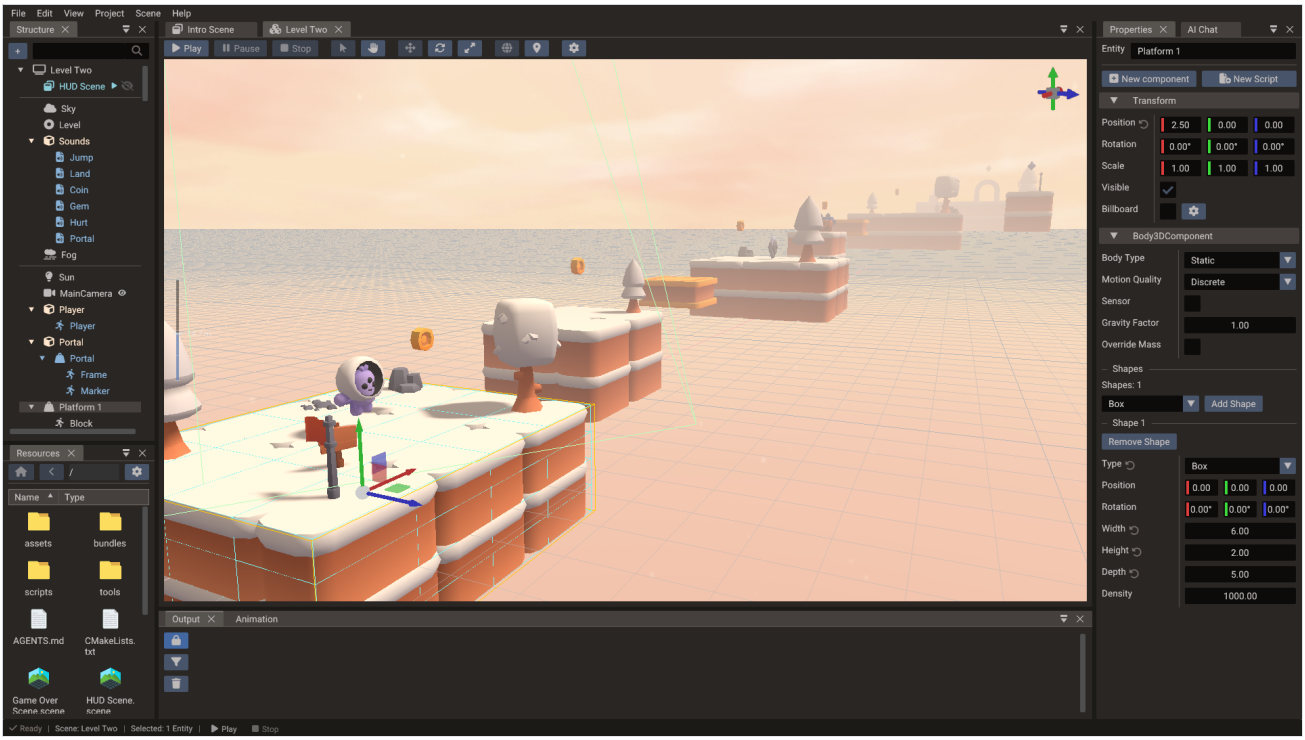
UI scene editing

UI scenes display the canvas in screen-space overlay mode. Anchor gizmos show the layout boundaries of UI elements. Drag anchors directly to reposition or stretch widgets relative to their parent.



Physics visualization

When a `Body2D` or `Body3D` is attached, the editor draws the collision shape outline in the viewport. This lets you verify that the physics shape matches the visual mesh without running the game. Shapes are drawn for the selected entity; turn on **Show all bodies** in the viewport settings to draw every body.



Play mode in the viewport

Press **Play** to run the scene inside the viewport. All input and logic operate normally. While the viewport has focus the game gets every key except **F8** and **F11**, so a pause menu bound to **Escape** works. When you press **Stop**, the scene is restored to the pre-play snapshot. Entities spawned by scripts during play are removed even if they never appeared in the Structure panel. Pending sprite frame selections from play are cleared so they cannot overwrite the restored texture rectangles after Stop. Use play mode for fast local iteration; export to a real build for platform-specific testing.

Play mode follows **Project → Project Settings → VSync**. With VSync disabled, the editor render loop is uncapped while a scene is running; after you stop, frames follow **Editor Settings → Editor VSync** instead. See Project Workflow — VSync (p. 53) and Editor VSync (p. 53).

The editor footer on the right shows FPS, frame time, draw calls, and triangles for the last drawn frame — the same counters `Engine::getFrameStats()` returns. For a scripted capture with VSync forced off, use `doriax-editor benchmark` (p. 169).

Touch simulation

The **phone** button after the play controls makes Play behave like a touch device. `System::isTouchDevice()` (p. 408) returns `true`, and the left mouse button acts as one finger: pressing starts a touch, dragging moves it, and releasing ends it. As on a phone, the game gets touch events instead of mouse events, so there is no hover, right click, or scroll. The keyboard still works. The setting is saved per user in `.doriax/user/workspace.yaml` and can only be changed while nothing is playing.

Detached camera

The **camera** button next to **Stop** (or **F8**, or **Scene → Detach Camera**) switches a running or paused scene from the game camera to the editor camera. The game keeps running: scripts, physics, and audio still use the game camera, and `Scene::getCamera()` still returns it.

While detached:

- The editor overlays come back, including the game camera's frustum and the icons of lights, cameras, and sounds the game creates.
- You can navigate, frame entities with **F**, and select them by clicking or with a selection box, including entities the game created.
- Gizmos stay hidden, so entities can't be moved from the viewport.
- The game gets no mouse or keyboard input, and a captured or hidden cursor is released.

Press **F8** again to return to the game camera. **Stop** always returns to the editor view. While the scene is paused on the game camera, the viewport ignores the mouse; detach to look around.

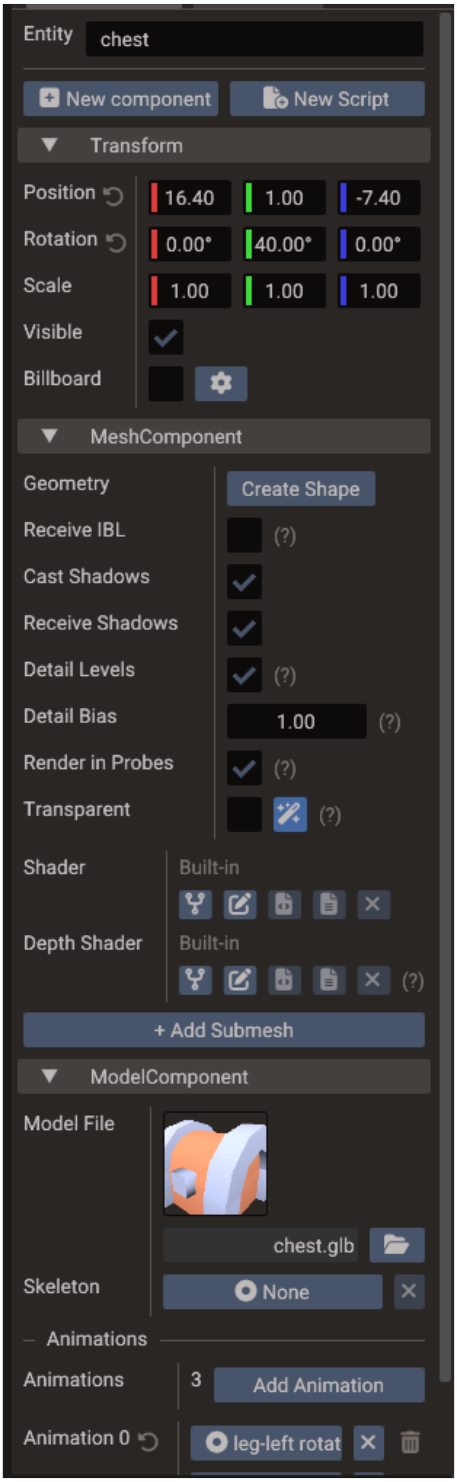
Full screen

The **maximize** button next to the **camera** button (or **F11**, or **Scene → Full Screen**) shows a running or paused scene over the whole editor window, covering the menu bar and every panel. The game gets the bigger canvas, as if you had enlarged the viewport. It fills the editor window, not the monitor, so maximize the editor first for the biggest view.

Escape still goes to the game, so press **F11** again to leave; a hint at the top reminds you for a few seconds. **Stop** also leaves full screen.

Properties & Components

The **Properties window** displays every component attached to the selected entity and lets you read and change their values. It is the primary place to configure transforms, materials, physics, scripts, UI layout, audio, animations, and any other component data.



Selecting an entity

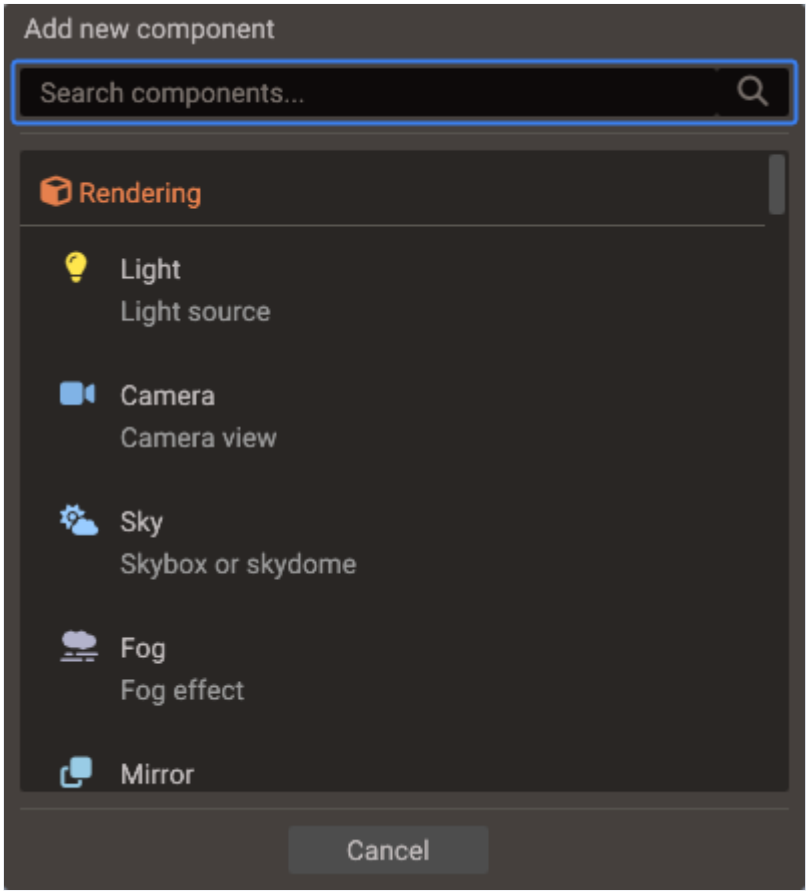
Click an entity in the **Structure panel** or directly in the **Scene view** to select it. The Properties window updates immediately to show its components.

The field at the top holds the entity name: edit it to rename the entity, or press **F2** anywhere in the scene windows to jump straight into it with the name selected. With several entities selected the field is read-only; with none selected, the window shows the scene settings instead and the same field renames it.

While a scene is playing, an entity the game created (drawn in green in the Structure panel (p. 80)) is shown read-only: you can watch its values change, but not rename it, edit its fields, or add and remove components, because **Stop** destroys it.

Adding components

Use the **New component** button at the top of the Properties window. A searchable dialog lists all available component types grouped by category; type in its search field to narrow the list. Components are added to the entity immediately and their defaults appear in the panel for editing.



Component groups

The dialog groups the components into these categories:

Category	Components
Core	Transform
Rendering	Mesh, Light, Camera, Sky, Fog, Mirror, Reflection Probe, Water, Model, Terrain, Instanced Mesh
User Interface	UI Base, UI Layout, UI Container, Image, Text, Button, Panel, Scrollbar, Progressbar, Text Edit
2D Graphics	Sprite, 2D Light, 2D Occluder, Sprite Animation, Tilemap, Polygon, Mesh Polygon
Primitives	Points, Lines, Particles
Physics	2D Physics Body, 3D Physics Body, 2D Joint, 3D Joint

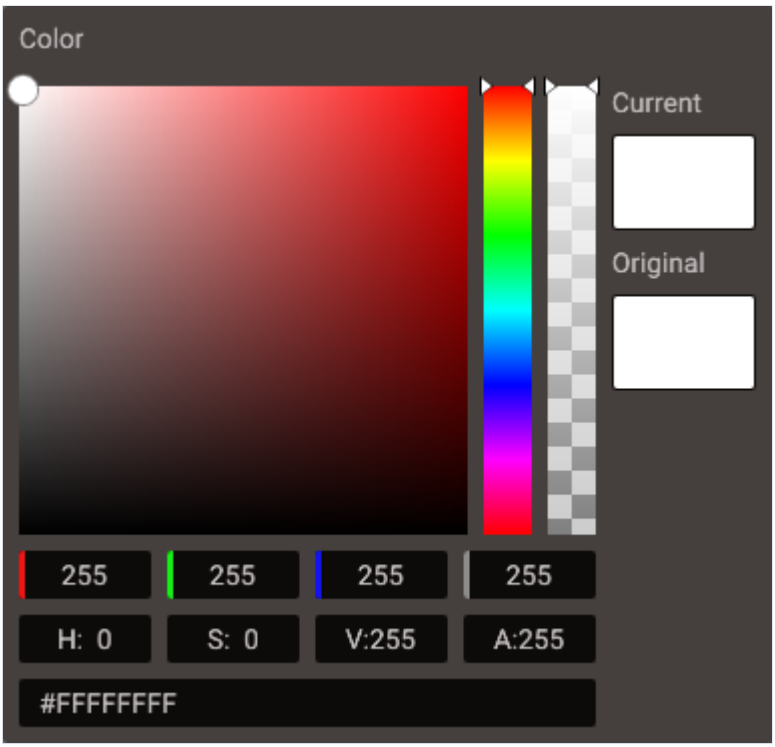
Category	Components
Animation	Animation, Action, Timed Action, Alpha / Color / Position / Rotation / Scale Action, Bone, Skin, Keyframe Tracks, Morph / Rotate / Scale / Translate Tracks
Sound & Script	Sound, Script

Editing component fields

Most fields are edited with the mouse or keyboard directly in the panel:

Field type	Editing method
Numbers (float, int)	Click and drag, or click without dragging to type a value
Vectors (Vector2, Vector3)	Per-component drag or type fields
Colors	Color picker with RGBA, HSV, and hex fields
Booleans	Toggle checkbox
Strings	Inline text input
Fonts	Pick or drag a TTF/OTF/TTC file for the main font; use the stacked layers button for ordered fallback fonts
Textures / assets	Drag from the Resources Browser, type the asset path, or pick a **camera** as the source (camera button / drag a camera entity) for render-to-texture. For image sources, a sliders button opens the **Texture Settings** popup (filter, wrap, alpha border and SVG scale)
Entity references	Drag an entity from the Structure panel
Enumerations	Drop-down selection

A typed number can also be a math expression, such as `288*3` or `(10+5)/2`. Number fields accept `+`, `-`, `*`, `/`, and parentheses, and integer fields round the result. The value is applied when you press **Enter** or **Tab**, or click somewhere else, and **Esc** cancels the edit. An expression that can't be calculated leaves the value unchanged. This works in every number field of the editor, not only in Properties.



When a field differs from the component's default value, a small reset arrow appears next to its label; clicking it restores the default. Edits apply to every selected entity and every change is undoable.

On a button, the UIComponent **Color** and **Texture** are greyed out: the button replaces them with the ones of its current state whenever it loads or changes state. Set **Color Normal** and **Texture Normal**, and the hovered, pressed and disabled ones, in the ButtonComponent instead. MCP agents are refused those two fields on a button for the same reason.

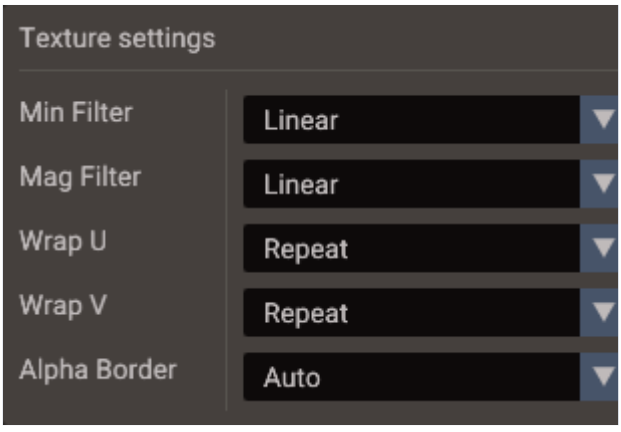
A texture field set to a camera shows that camera's live output (render-to-texture). The camera is switched to render-to-texture automatically and can no longer be the scene's main camera. This is the manual building block behind effects like minimaps and monitors; for reflections, prefer the Mirror component, which manages its own camera.

Once a camera is in render-to-texture mode, its own properties gain a **Render Target** section for setting the framebuffer **Width**, **Height**, and sampling **Filter** (Nearest or Linear). These control the resolution and look of the output texture, are saved with the scene, and are applied in exported projects.

A camera's properties also list the view shader parameters (p. 287) the scene's shaders declare, such as the opening of the engine's see-through, under **Shader Parameters**.

When the field holds an image texture (not a camera link or an empty slot), a sliders button next to the camera button opens the **Texture Settings** popup:

- **Min Filter** / **Mag Filter** — sampling filters for minification and magnification. Picking a mipmap variant for Min Filter enables mipmap generation for the texture.
- **Wrap U** / **Wrap V** — wrapping mode per texture axis.
- **Alpha Border** — whether transparent pixels next to visible ones take their color on load, so filtering does not draw a dark outline. **Auto** fixes sprites, tilemaps, polygons, UI and particles and keeps 3D materials as they are; see Texture.alphaBorder (p. 740).
- **SVG Scale** (only for `.svg` sources) — a multiplier on the vector's intrinsic size (a drag field plus 0.5×–4× presets), so the image rasterizes crisp when drawn larger or on high-DPI displays. The scale is stored on the texture reference, so different slots can use the same SVG at different resolutions.



Like any other field, each row shows the reset arrow when it differs from the component's default, and the settings are saved with the scene and applied in exported projects. See Vector images (SVG) (p. 239) for the underlying behavior.

Text fonts

The Text component's **Font** row controls the main font. Pick a `.ttf`, `.otf`, or `.ttc` file with the folder button, drag one from the Resources Browser, or use the clear button to return to the built-in default. Font files must be inside the project's assets directory (p. 52).

The stacked layers button opens **Additional fonts** with three fallback slots. The engine tries **Fallback 1**, **Fallback 2**, and **Fallback 3** in order whenever the main font does not contain a character; empty slots are skipped, and the built-in Roboto and Arabic fonts finish the chain. Each slot has its own picker, drag target, clear button, and reset action. A highlighted layers button means at least one fallback is configured.

Dropping a font directly onto a Text entity in the Scene view replaces only the main font and keeps its fallbacks. The preview is live and the final assignment is undoable. Every slot is saved, generated into exported scene and bundle code, included in asset path updates, and copied with the project assets.

The editor's AI assistant also treats a Font property as this ordered chain. A full-chain edit replaces the main font and fallbacks, clearing slots omitted from the request; a main-font-only edit preserves the current fallbacks. The assistant accepts only safe, project-relative font paths inside the assets directory.

See Text and fonts (p. 293) for runtime behavior and the Text API (p. 529) for C++ and Lua access.

Model component

The **Model** component loads a GLTF/GLB/OBJ/FBX file onto the entity (an FBX through its glTF conversion (p. 251)). Multi-node GLTF files usually create child entities under the model root — the full node tree when the file has animation clips or more than one skin, otherwise one child mesh per mesh node. See 3D Graphics — GLTF node hierarchy (p. 249).

Control	Purpose
Model File	Path to the model asset (browse or drag from Resources)
Skeleton	Optional skeleton entity reference
Animations	Clips imported with the model

For static multi-node models that need same-entity features such as Instanced Mesh, use **Merge static model** on the entity in Structure (p. 78) (or **Restore model mesh children** to undo). That sets the serialized `mergeStaticMeshes` flag and reloads the model in one undoable step. See 3D Graphics — Merging static model meshes (p. 250).

Setting **Model File** rebuilds the child entities. Re-assigning the **same** file keeps what you arranged: a static model whose parts you have rearranged in Structure (p. 77) keeps those parts and refreshes them in place, an animated model is rebuilt and your moved parts are put back by node name, and entities attached to bones or parts are kept in both cases. See 3D Graphics — Reloading a rearranged model (p. 250).

Each skinned mesh of a model gets a **Skin** component with its bone matrices. The loader adds and fills it, so Properties only shows how many bones the skin has. A mesh without one draws rigid, in its bind pose. Scenes save the Skin only for meshes you build yourself; a model rebuilds its skins when it loads.

Instanced Mesh

Instanced Mesh draws many copies of the *same entity's* mesh geometry with per-instance transforms. Add it from **Add Component**, then populate the instance list in Properties or from script via `Mesh::createInstancedMesh` / `addInstance`.

Instancing only uses geometry on the entity that owns the component. If a Model still stores its meshes as child entities (the default multi-node or animated GLTF layout), the editor shows a warning and writes to the Output panel: instances will not render until you merge the static model (p. 78).

Property	Purpose
Max Instances	GPU instance-buffer capacity
Billboard / **Cylindrical Billboard**	Turn every instance to face the camera

Property	Purpose
Cull Instances	Skip instances outside the camera and shadow views. Turn off for shaders that move instances away from their bounds
Cull Distance	Drop instances farther than this from the main camera, in world units; `0` = unlimited. A hard cut, so hide it with fog. Needs Cull Instances
Distance Fade	Shrink instances between **Fade Start** and **Fade End** (model-space distance) instead of popping. Fade End must be greater than Fade Start

A clone mark next to the entity in Structure indicates the instance count. See Rendering Pipeline — GPU instancing (p. 274).

Mesh materials and IBL

The **Mesh** component exposes rendering flags beyond per-submesh material slots:

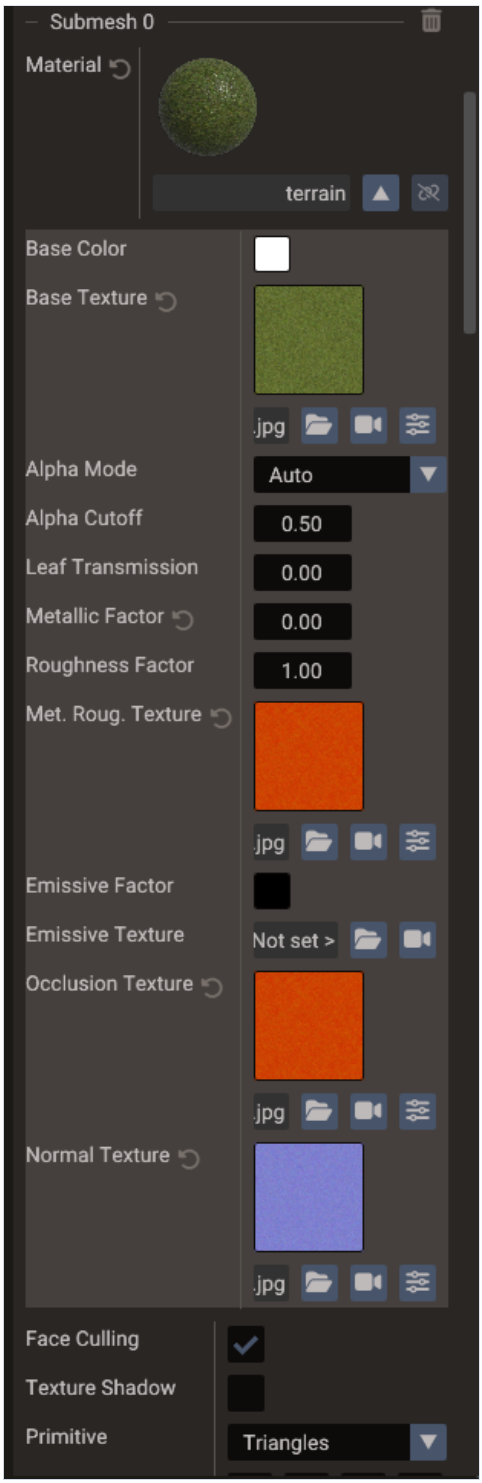
Property	Purpose
Receive IBL	When enabled, the mesh is lit with image-based lighting (diffuse irradiance + specular reflections) in addition to punctual lights — from the scene's Sky environment, from a Reflection Probe (p. 468) covering it, or both. Sky lighting requires a Sky entity with a cubemap texture.
Receive Lights	Master switch for dynamic lighting (must be on for IBL to apply). Turn it off for an unlit mesh that shows its colours as they are, like a glow or a light beam.
Cast / Receive Shadows	Shadow map participation.
Detail Levels	Let this mesh use simplified meshes (p. 275) while the scene's Mesh Detail is on. Turn off to always keep it at full detail.
Detail Bias	Scales how far detail is kept: above 1 keeps it longer, below 1 drops it sooner. Shown only while the scene's Mesh Detail is on.
Render in Probes	Whether the mesh appears in Reflection Probe (p. 468) captures. Turn it off so an object wrapped around a probe is not seen in its own reflection.

Effects such as wind, see-through or a dissolve are set in the mesh's **Shaders** section, as a vertex offset or a coverage function (p. 144).

Each **Submesh** section contains a **Material** row with a shaded preview sphere. The preview reflects the current **Receive IBL** setting so you can see environment reflections before entering play mode.

On a mesh that came from a model file, every field you change in the Submesh section is kept as an override on the model and re-applied whenever the file reloads, while the fields you leave alone keep following the file. Geometry and submeshes always come from the file, so these meshes show **From model file** instead of **Create Shape**, and submeshes can't be added or removed. See 3D Graphics — Editing an imported model's submeshes (p. 254).

Expand the Material row to edit **Alpha Mode** and **Alpha Cutoff** alongside the base colour, textures, metallic, and roughness controls:



Alpha mode	Use
Auto	Compatibility mode for editor-created and older materials. Texture/factor alpha can select transparent rendering automatically.
Opaque	Render the material fully opaque, regardless of base-colour alpha.
Mask	Cut out pixels below **Alpha Cutoff**; useful for leaves, hair cards, fences, and similarly sharp-edged transparency.
Blend	Render smooth or partial transparency through the transparent pass. Depth testing stays on; depth writes are off in the colour pass.

Alpha Cutoff ranges from `0` to `1`, defaults to `0.5`, and only affects **Mask**. Masked cutouts use the same threshold in the visible surface, shadows, depth, and SSR G-buffer.

Leaf Transmission (`0`–`1`, default `0`) lets light through a thin surface when it is lit from behind — backlit foliage. Leave it at `0` on bark, trunks, and solid geometry. Like the other material fields it is stored as an override on imported models; a glTF can also carry it as the material extra `doriax_foliage_transmission`. See 3D Graphics — PBR materials (p. 252).

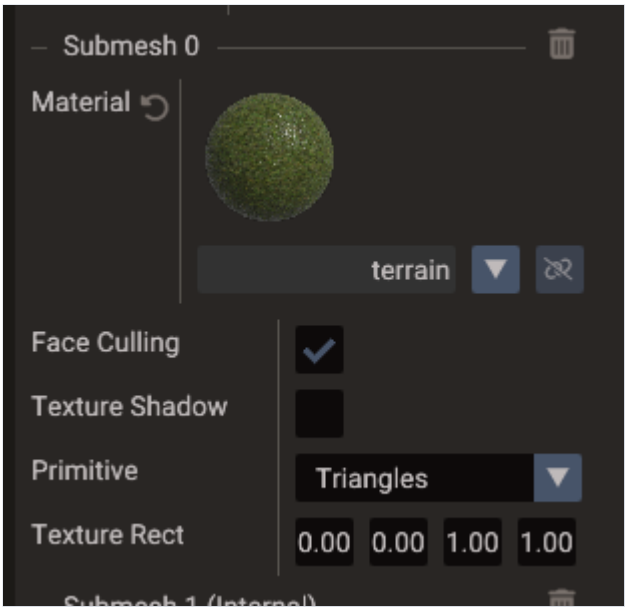
For explicit **Opaque**, **Mask**, and **Blend** modes, the engine derives **Texture Shadow** from the alpha mode: it is enabled for Mask and disabled for the other two. The separate **Texture Shadow** checkbox remains meaningful in **Auto** mode for legacy materials and sprites.

Linking material files

The Material row supports three workflows:

1. **Edit inline** — expand the material section and change factors and textures directly in the scene (values stored on the mesh component).
2. **Link to a file** — drag a `.material` file from the Resources Browser onto the Material row or preview; the mesh submesh links to that file and reloads when it changes on disk.
3. **Create a shared file** — drag the material preview **from Properties into the Resources Browser**; the editor writes a new `.material` file and links the submesh to it automatically.

When linked, the material name shows the file name and an **unlink** button (broken chain icon) appears. Unlinking keeps the current values on the mesh but stops syncing with the file.



You can also drop a single **image** onto the Material row to assign only the base colour texture without replacing the rest of the material.

Custom shaders

Mesh, UI, Points, Lines, Sky, and Water components expose a **Shader** row. By default it shows **Built-in**; **Fork** opens a dialog to choose where the fork is created and what it is called, then writes it and opens the GLSL in the Code Editor. The remaining buttons pick the `.vert`/`.frag` files, reopen the assigned shader, and reset to the engine default. You can also drag an existing `.vert`/`.frag` from the Resources Browser onto the row.



A Mesh has a second **Depth Shader** row with the same controls. It forks the shader behind the mesh's shadow maps and depth pre-pass, for colour forks that move vertices or discard fragments, so the shadow follows — see Custom Shaders — The Depth Shader row (p. 144). A Water has an **Underwater Shader** row instead, for the fade seen from below the surface — see Custom Shaders — Water shaders (p. 145).

Each fork, and each effect file of a Mesh, gets a **Parameters** section with one field per `#pragma param` line it declares. The values are saved on the component, shared by all of its shaders, and scriptable through `setShaderUniform`. See Custom Shaders — Shader parameters (p. 147).

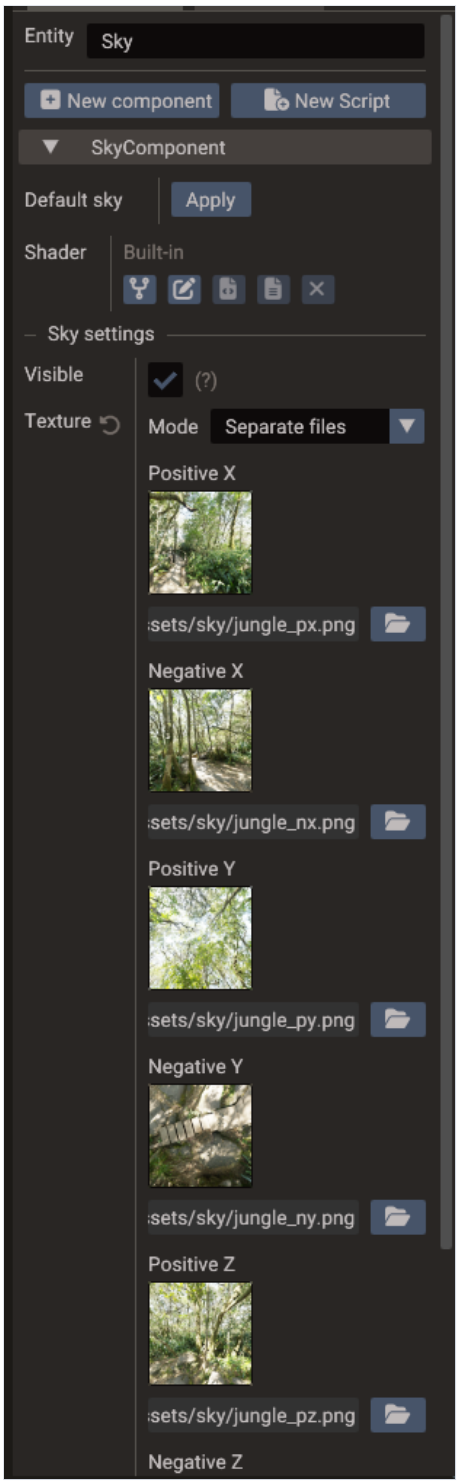
A scene can also define a **default custom shader per type** (in the scene settings' **Default Shaders** section). Components with no shader of their own inherit it, and their Shader row shows its name followed by *(scene default)*. A shader assigned on the component always takes priority over the scene default. When the default declares parameters, the section has fields for them too: those values belong to the scene, and an inheriting component reads them for every parameter it does not set itself.

The scene settings also have a **Post-processing** section: an ordered list of fullscreen passes, each one a forked shader under a collapsible header with reorder and remove buttons, and an enable checkbox in its Shader row. Each `#pragma param` the pass declares becomes a field here.

See Custom Shaders (p. 143) for the full workflow, scene defaults, post-process passes, includes, and project settings.

Sky component

The **Sky** component controls the scene cubemap background and the IBL environment:



Property	Purpose
Visible	Draw the sky in the viewport. When off, the sky is hidden but still feeds IBL to meshes with **Receive IBL** enabled.
Texture	Cubemap source (six faces or bundled layout). Changing it rebuilds IBL maps.
Color	Tint multiplied with the sky texture.
Rotation	Rotate the sky around the Y axis.

Use **Default sky** to restore the built-in editor cubemap.

Mirror component

The **Mirror** component turns its mesh into a planar reflection surface. Add it to a flat mesh (a Wall (p. 76) works out of the box), or use the **Mirror** entry in the Structure create menu, which sets up the wall and component together.

Property	Purpose
Normal	The reflecting surface direction in the mesh's local space (default `+Z`, matching a Wall). It is rotated with the entity to form the world mirror plane.

The reflection camera is created and managed by the engine — there is nothing else to wire. If the reflection is clipped on the wrong side, flip the sign of **Normal**. See Rendering Pipeline — Mirrors and planar reflections (p. 269) for how it works and its cost.

Reflection Probe component

The **Reflection Probe** component gives meshes inside its box-shaped influence volume a local reflection environment instead of the global sky — see Rendering Pipeline — Reflection probes (p. 263) for how probes are selected, blended, and captured.

Property	Purpose
Mode	**Static** (authored cubemap, or captured once at load) or **Dynamic** (re-captured at runtime).
Update	*(dynamic)* Capture policy: On Load, On Move, Interval, or Manual.
Update Interval	*(dynamic, Interval)* Seconds between captures.
Cubemap	*(static)* Authored six-face cubemap; leave empty to capture at load.
Intensity	Reflection strength multiplier.
Priority	Higher-priority probes win where influence boxes overlap.
Box Offset / Box Size	The influence volume, in the entity's local space. The cubemap is always captured at the entity origin.
Blend Distance	Fade band at the box edges where the probe blends into the sky IBL.
Resolution	Capture face size (16–1024).
Near / Far	Capture camera clip planes.
Include Sky	Whether the sky appears in captures.

Selecting the probe draws its influence box in the viewport — cyan for static, purple for dynamic — with a gold marker at the capture origin when **Box Offset** moves the box away from it. The **Refresh Probe** button forces a re-capture (or re-bake of the authored cubemap) regardless of mode.

Water component

The **Water** component draws an animated water surface at the entity's height. Use the **Water** entry in the Structure create menu, or add the component to any entity with a Transform. See Rendering Pipeline — Water (p. 270) for how each feature is drawn and what it costs.

Under the **Shader** and **Underwater Shader** rows, the fields are grouped into sections:

Section	Property	Purpose
Surface	**Size**	Width (X) and depth (Z) of the surface, centred on the entity and scaled by its transform.
	Subdivisions	Grid cells along each side (16–512). Short waves need more cells to keep their shape.
Color	**Shallow Color**	Colour of shallow water. With Refraction or Underwater, also the tint of the scene seen through it.
	Deep Color	Colour the water turns to as it deepens.
	Depth Fade	Depth where the deep colour takes over. Shallower water is lighter and clearer.

Section	Property	Purpose
Waves	**Height**	How far the crests rise above the water level.
	Length	Length of the longest wave. Longer waves also travel faster.
	Speed	Multiplier on the wave travel speed.
	Direction	Angle around Y the waves travel toward, in degrees. `0` is `+X`.
	Steepness	`0` gives round waves, `1` sharp crests. Lowered where the waves would fold over.
Ripples	**Normal Map**	Tiling normal map of the small ripples; the built-in one when empty. Use a mipmap filter so distant ripples do not shimmer.
	Tile Size	World size of one tile of the normal map.
	Strength	How much the ripples bend the lighting and reflections.
	Speed	How fast the ripple layers drift.
Reflection	**Reflectivity**	Strength of the reflection.
	Specular	Strength of the sun glints.
	Roughness	Spread of the sun highlight. Low values give sharp glints.
	Receive Shadows	Shadows of the scene lights darken the glints and the light under the surface.
	Planar Reflection	Reflects the scene, not only the sky. Draws the scene a second time at half resolution.
	Distortion	*(Planar Reflection)* How much the ripples bend the reflection.
Refraction	**Refraction**	Shows the scene behind the water, bent by the ripples and tinted by the water it crosses. Copies the scene once per frame.
	Distortion	*(Refraction)* How much the ripples bend the scene behind the water.
	Caustics	*(Refraction)* Brightness of the sunlight patterns on the floor seen through the water, 0 turns them off. Needs Depth Effects.
	Caustics Scale	*(Refraction)* World size of one cell of the caustics.
Underwater	**Underwater**	A camera below the surface sees the scene fade into the water with distance.
Foam	**Depth Effects**	Shore foam, soft edges, and depth tint from the scene depth. Adds a depth pass when SSAO and SSR are off.
	Foam Color	Colour of the foam.
	Shore Foam	Water depth the shore foam covers. Needs Depth Effects.
	Crest Foam	How much of the surface breaks into whitecaps, `0` for none.
Buoyancy	**Buoyancy**	Floats the dynamic bodies that have Body3D buoyancy.
	Depth	*(Buoyancy)* How far below the surface bodies still float, `0` has no bottom. Where waters overlap, the one with the highest bottom is used.
Wake	**Area**	World size of the square around the camera that keeps the wakes of moving Water Exclusions. Larger reaches farther with less detail.
	Fade	Seconds a wake takes to fade.

In the editor the waves and ripples hold still while the scene is stopped or paused, and move during Play. To keep the editor view clear when the camera dips below the surface, turn on **Disable underwater** in the viewport settings (p. 82).

A dynamic 3D Physics Body (p. 319) floats on the water once its **Buoyancy** is above `0`, unless the water's own **Buoyancy** is off.

Water Exclusion component

The **Water Exclusion** component keeps water out of a volume: the water surface is not drawn inside it, a camera inside sees no underwater fade, and the dynamic bodies inside get no buoyancy. Add it to a boat to

keep the waves out of it, or create a **Water Exclusion** entity from the Structure create menu for a volume of its own. See WaterExclusion (p. 503).

Property	Purpose
Shape	**Hull** wraps the meshes of this entity, like a boat, and follows them when they change. **Box** and **Sphere** fill a volume of their own.
Center	*(Box, Sphere)* Centre of the volume in local space.
Size	*(Box, Sphere)* Width, height, and depth of the volume in local space.

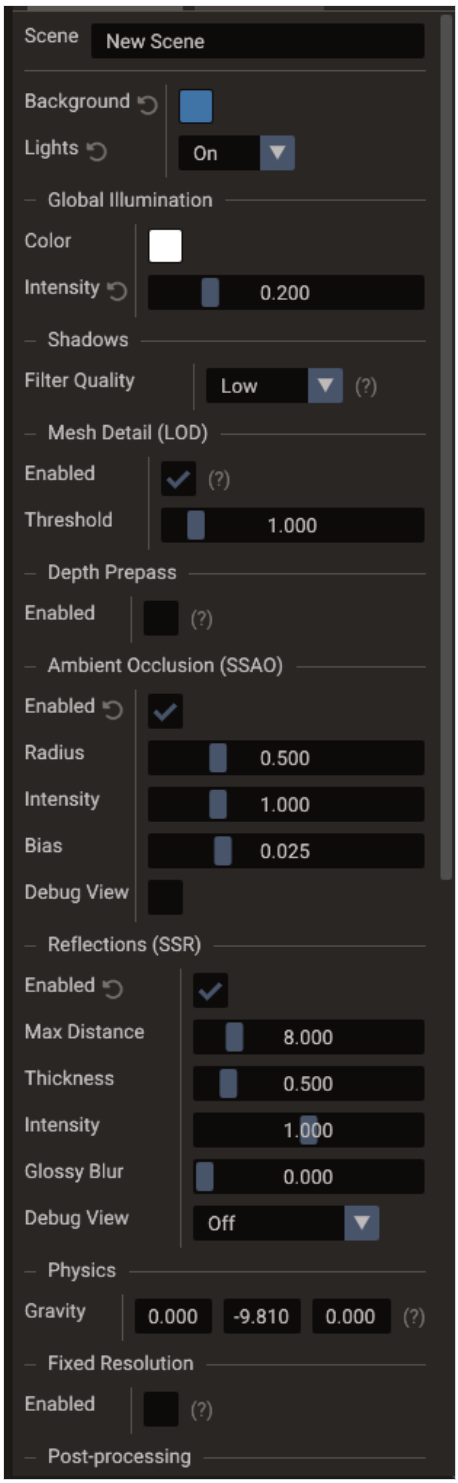
A Hull on an entity without meshes shows *No mesh on this entity to wrap*. While selected, the volume is drawn as an orange wireframe.

Under **Wake**, the volume leaves a wake (p. 504) on the water while it moves. It is made only during Play, from how the entity moves.

Property	Purpose
Foam	Foam left behind and along the sides while moving, more near hull speed. `0` leaves none, up to `2`.
Waves	Kelvin wake waves left while moving. `0` leaves none, up to `2`. The water's **Fade** sets how long they last.

Scene settings

With no entity selected, Properties edits the scene. Lighting, shadows, SSAO, SSR, fixed resolution, default shaders, and post-processing live here — see Rendering Pipeline (p. 258) for those. The scene values of shader parameters (p. 287), such as the wind, are under **Shader Parameters**.



Two scene-wide performance switches sit with the lighting block:

Section	Purpose
Mesh Detail (LOD)	Draw simplified meshes (p. 275) when their geometric error no longer shows. **Enabled** is off by default, and while it is off no levels are built; **Threshold** is the allowed screen-space error in pixels (lower keeps more detail).
Depth Prepass	Write depth before the opaque colour pass (p. 276) so each pixel is shaded once. Off by default; pays off with heavy overdraw.

Each mesh can still opt out with **Detail Levels** on the Mesh component.

Scripts

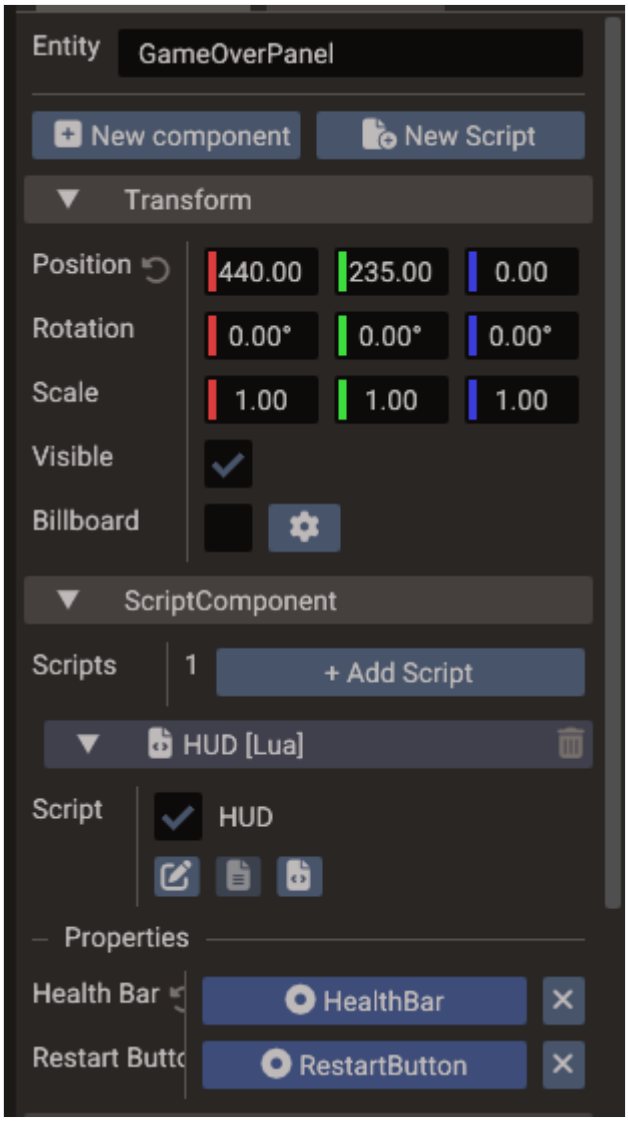
Two buttons at the top of the Properties window create scripts:

- **New Script** — opens a dialog that generates the script files (Lua module, or C++ header + source), adds a `ScriptComponent` if the entity lacks one, and links an enabled entry to the new files — all in one undoable step. Its **Attach existing** option links script files that already exist instead, without changing them.
- **New component → Script**, then **Add Script** inside the component — adds an *empty* script entry without creating files. Use the entry's pencil button to set the class name and link existing header/source (or `.lua`) files.

Each script entry shows an enable checkbox, buttons to open its files in the Code Editor, and a right-click menu to reorder or remove it.

Script properties

When a script is linked, the Properties window shows every property declared with the `DPROPERTY` macro (C++) or in the `properties` table (Lua). These fields are serialized with the scene and injected into the script instance at play and export time.



See Creating Scripts (p. 199) for the full workflow and Script Properties (p. 223) for declaring and typing custom script properties.

Entity references

Some fields take another entity as their value. For example, a physics joint links two Body entities, an action animates a target entity, and script properties of pointer types show an entity picker. Drag the target entity from the Structure panel into the field, or use the picker icon.

Which scene the entity can come from depends on the field:

Field	Accepts
Component fields: joint `bodyA` / `bodyB`, action target, a collision shape's source entity, UI parts, and so on	Only entities of the **same scene**. The value is a bare entity ID, and entity IDs are numbered per scene.
Script entity properties (pointer types (p. 224))	Entities of the same scene, or of a child scene shown inline in the Structure panel. The reference also stores the other scene's ID.

An entity from a child scene dropped on a component field is refused with the tooltip *Entity belongs to another scene*. A joint can therefore never connect bodies in two scenes, since each scene runs its own physics world (p. 323).

A script property that points into another scene is resolved when the script starts. That scene must already be built at that point, for example as a child scene of the same stack. If it isn't, the property is `nil` in Lua, and `nullptr` in C++ with an error in the log.

Transform gizmo

Selecting a transform field highlights the corresponding gizmo in the Scene view. You can edit numeric values in the Properties window and see the result in the viewport at the same time, or manipulate the gizmo and watch the numbers update live.

Undo / redo

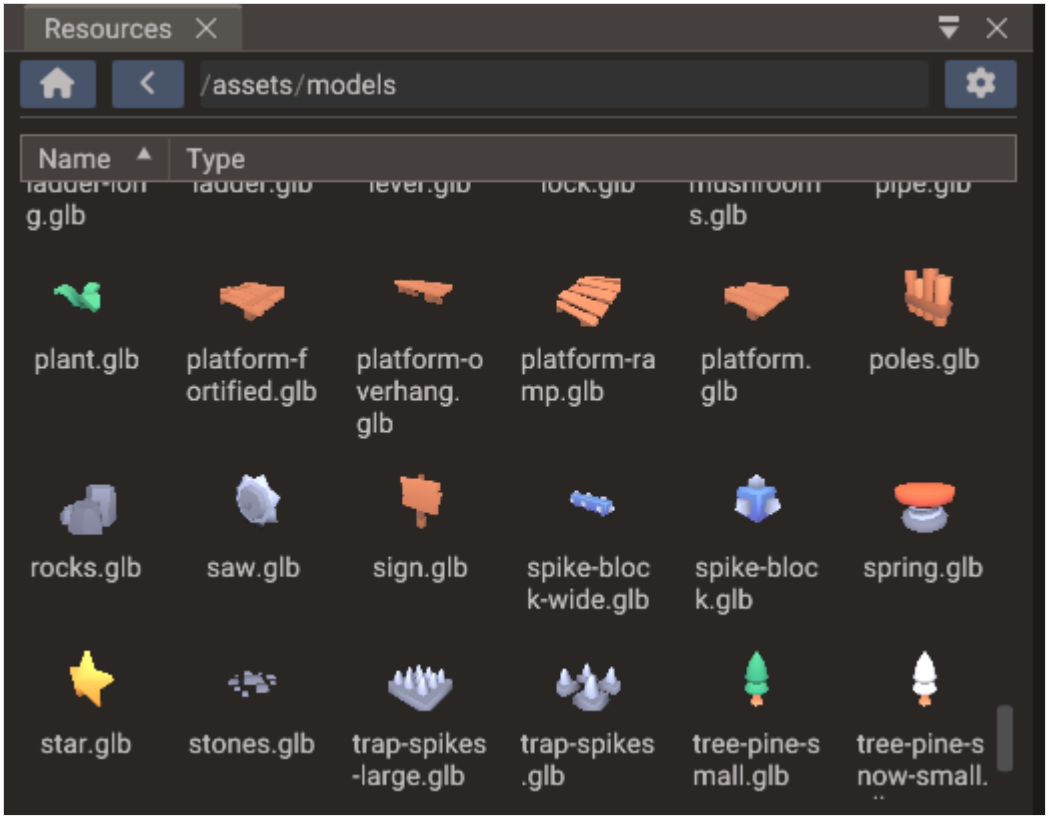
Every property edit is recorded in the command history. Use **Ctrl+Z** / **Ctrl+Y** to step back and forward through changes made in the Properties window or in the viewport.

Best practices

- Prefer editor-exposed `DPROPERTY` fields for any value a designer should tune, rather than hard-coding it in script logic.
- Keep generated or runtime-only data out of serialized component fields.
- After changing physics, script, or animation fields, test in play mode immediately because those affect runtime behavior, not just visual appearance.
- Use bundles (p. 111) for repeated configured entity hierarchies so the same property values do not have to be re-entered manually in many scenes. For entities inside a bundle instance, the component header's right-click menu offers **Make Unique** (override this instance) and **Revert to Bundle** (re-link to the shared values).

Resources Browser

The **Resources Browser** is the file manager for your project's assets. It lets you import new files, preview existing resources, drag assets into scenes, and assign them to components.



Supported resource types

Type	Extensions	Typical use
Textures	PNG, JPG, BMP, TGA, PSD, HDR, SVG	Sprites, UI images, albedo/normal/roughness/metallic maps, skyboxes
Models	GLTF, GLB, OBJ, FBX	3D meshes, skins, node hierarchy, animations, morph targets
Materials	`.material` (YAML)	Reusable PBR material definitions shared across meshes
Audio	OGG, WAV, MP3, FLAC	Sound effects, music, 3D spatial audio
Fonts	TTF, OTF, TTC	Text and UI rendering
Scenes	YAML scene files	Saved scenes and child scene references
Bundles	YAML `.bundle` files	Reusable entity hierarchies — see Bundles (p. 111)
Shaders	Shader data files	Built, serialized, and loaded shader programs
Scripts	LUA, CPP, H	Gameplay logic files

Scenes and bundles the project does not use are drawn **dimmed**: a `.scene` file that is not one of the project's scenes, and a `.bundle` no scene instantiates that is not listed in **Project → Bundles**. Hover one to see why — a dimmed bundle is not built into the game, so `BundleManager` cannot spawn it. See Bundles in the build (p. 114).

Previews

The browser includes inline preview renderers. Select a resource to see:

- **Texture preview** — full image with mip-level and channel controls.

- **Model preview** — interactive 3D view with orbit navigation, lighting, and material display.
- **Material preview** — shaded sphere with the PBR material applied.
- **Font preview** — sample text at the font's supported sizes.
- **Audio preview** — waveform display and play button.

Use previews to catch missing textures, wrong scale, or broken imports before dropping assets into a scene.

Importing assets

Copy files into your project folder or drag them from your OS file manager into the Resources Browser. The editor scans the asset directories on startup and whenever the project folder changes.

A dragged-in file never replaces one with the same name: it arrives with a number added, such as `grass_2.png`. To update an asset in place, overwrite it from the file manager instead.

> **File naming**
> Use lowercase, hyphen-or-underscore-separated file names without spaces. Consistent naming avoids case-sensitivity issues on Linux and Android and makes asset paths predictable in scripts.

FBX models

FBX files are used like any other model: drop one into a scene, pick it for a component, or load it by its `.fbx` path from a script. The editor converts each one to a binary glTF the first time it is needed (the browser's preview counts), keeps the conversion in `.doriax/imported/`, and converts it again when the FBX or one of its textures changes. Models in open scenes then reload with the new conversion. Warnings, such as a texture that could not be found, appear in the Output panel.

Reimport in the context menu converts a file again on demand. See 3D Graphics — FBX models (p. 251) for what is converted and how textures are found.

Dragging assets out of the browser

Files dragged from the Resources Browser are accepted by several editor windows:

Asset type	Drop target	Result
Model (GLTF/GLB/OBJ/FBX)	Scene view (3D scene)	Creates a Model entity at the drop position
Image	Scene view, empty space (2D / UI scene)	Creates a Sprite (2D) or Image widget (UI) sized to the texture
Image	Scene view, onto a mesh or UI entity	Assigns the texture (base color / UI texture) with live preview
Material	Scene view, onto a mesh entity	Applies the material to all submeshes with live preview
Material (`.material`)	Properties → Material preview	Creates a new `.material` file in the open folder and links the source submesh
Font	Scene view, onto a Text entity	Assigns the main font with live preview and preserves its fallback slots
Texture / audio / asset	A component field in Properties	Assigns the file to that field
`.scene` file	Structure panel, scene root	Adds the scene as a child scene
`.bundle` file	Structure panel	Creates a bundle instance (as child of the target entity, or at the root)

Only files inside the assets directory (p. 52) can be assigned. One from elsewhere in the project still previews while you drag, but the drop is refused with a warning, since its path could not be stored or exported. Material files are project files and are not affected.

See Scene View (p. 83) for the viewport behaviors in detail.

Material files

A `.material` file stores a PBR material as YAML: base colour factor, metallic and roughness factors, alpha mode and cutoff, per-texture UV sets, and texture paths for albedo, normal, metallic-roughness, occlusion, and emissive slots. Material files live anywhere under the project and appear in the Resources Browser with a shaded-sphere thumbnail preview.

Creating a material file

Drag the **material preview** from the Properties window (Mesh → Submesh → Material row) into the Resources Browser. Drop it on the folder where you want the file. The editor:

1. Creates `Material.material` (or `Material_1.material`, … if the name already exists).
2. Writes the current PBR settings into that file.
3. **Links** the submesh you dragged from, so future edits to the file propagate back to that mesh automatically.

This is the fastest way to turn a tuned material into a reusable project asset.

Applying and sharing materials

Action	How
Apply to a mesh	Drag the `.material` file from the browser onto a mesh in the Scene view, or onto the Material row in Properties
Share across many meshes	Link each mesh submesh to the same `.material` file — all linked meshes stay identical
Edit once, update all	Change the `.material` file on disk (or via linked Properties fields); the editor reloads linked meshes when the file timestamp changes
Stop sharing	Click the **unlink** button next to the material name in Properties

Linked materials are tracked per submesh. The scene stores the link; the `.material` file is the single source of truth for colour factors, alpha settings, UV selections, and texture paths. Linking a submesh of an imported model keeps the link across reloads of that model — see 3D Graphics — Editing an imported model's submeshes (p. 254).

> **Organize shared materials**
> Keep reusable `.material` files under something like `assets/materials/` and link props, terrain chunks, and instanced meshes to the same file instead of duplicating values in every scene.

Dragging entities in: creating bundles

The Resources Browser is also a drop *target*: drag an entity (or a multi-selection) from the **Structure panel** into the browser to save it as a `.bundle` file in the currently open folder. The dragged hierarchy is replaced in the scene by an instance of the new bundle. Since the file is created wherever you drop it, bundles can be organized in any directory of the project.

See Bundles (p. 111) for the full bundle workflow.

Context menu

Double-click a file to open it in its editor tool (image viewer, scene, or code editor). Right-click a file or folder to access:

- **Open (Add)** — `.scene` files only: opens the scene alongside the one already open instead of replacing it.
- **Reimport** — `.fbx` files only: converts the file to glTF again, even when nothing changed (see FBX models).
- **New** — folders only: creates a folder or file inside the folder (see Creating files and folders).
- **Copy** / **Cut** / **Paste** (`Ctrl+C` / `Ctrl+X` / `Ctrl+V`) — copies or moves the selected files. Paste puts the clipboard contents into the right-clicked folder, or next to the right-clicked file. Nothing is overwritten: when the name is taken, as when pasting into the same folder, the new file gets a number, such as `player_2.lua`.
- **Duplicate** (`Ctrl+D`) — copies the selected files into the same folder with a number added to the name, and selects the copies. A copied scene, from Duplicate or Paste, gets its own scene ID, so the original and the copy can both be part of the project.
- **Delete** — removes the file from the project and clears every reference to it in scenes, bundles and `.material` files, including C++ and Lua script entries. Undo restores the file, not the references.
- **Rename** (`F2`) — renames the file on disk and updates scene references. `F2` acts on the selected file, or on the one under the mouse when nothing is selected.
- **Open in File Manager** — shows the selection in the OS file manager (Explorer, Finder, or the Linux default). A folder opens itself; a file opens the folder that contains it.

Right-click empty space in the listing for **Import Files**, **Import Folders**, **New**, **Paste** into the current folder, and **Open in File Manager** for the current folder.

Creating files and folders

New opens a submenu of the items it can create. Each asks for a name; typing the extension is optional.

Item	Creates
Folder	An empty folder
Lua Script	A `.lua` module that returns an empty table named after the file
C++ Source	An empty `.cpp` file
C++ Header	A `.h` file starting with `#pragma once`

New files are not attached to any entity. Load a Lua module from other scripts with `require()` (p. 198), include a header from your scripts, or attach a script later with **Add Script** in Properties.

- **Lua Script** is enabled only inside the project's Lua directory (p. 52), which `require()` and script entries resolve against.
- A **C++ Source** compiles on its own only under one of the script directories (p. 211); elsewhere it compiles only as an attached script, and the dialog says so.

Undo moves a new file to `.trash`, and redo brings it back with any edits.

Sprite and tileset slicing

Sprite sheets and tilesets are cut by two slicer tools. Both open from the **Slicer Tool** button of a component in the Properties window (p. 90), not from the Resources Browser.

Sprite Slicer

The **Sprite Slicer** divides a Sprite's texture into named frames on a regular grid, for sprite animation or individual sprite display.

See Sprite Slicer (p. 116) for the complete workflow.

Tileset Slicer

The **Tileset Slicer** divides a Tilemap's texture into tile rects on a regular grid. Each rect has an index and a name; you place tiles by dragging a rect into the viewport, and scripts refer to rects by index.

See Tileset Slicer (p. 119) for the complete workflow.

Organization guidelines

- Keep source art, imported assets, and generated data in separate folders.
- Separate audio by category: `sounds/effects/`, `sounds/music/`, etc.
- Prefer GLTF for animated or material-rich 3D assets; OBJ for simple static geometry. FBX files work as they come from your content tool, through their glTF conversion.
- Store collision meshes separately from visual meshes.
- Remove unused large assets before exporting mobile or web builds to keep bundle sizes small.
- Do not name a top-level file or folder in the assets directory `resources.pak` or `resources.pak.tmp`. Those names are reserved for the optional native resource pack.
- On Windows 10 before version 1903, the editor and exports skip files and folders whose names use characters outside the system code page, such as Chinese names on a system set to a Western language. Later versions handle any name.

See also

- Bundles (p. 111)
- Sprite Slicer (p. 116)
- Tileset Slicer (p. 119)
- Resources & Assets (p. 238)
- 2D Graphics (p. 243)

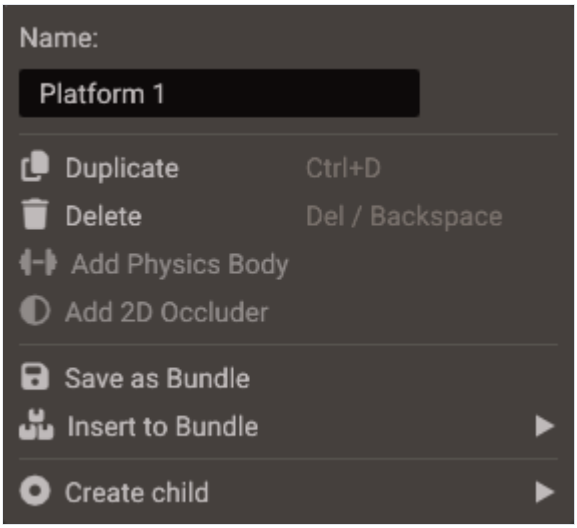
Bundles

A **bundle** is a reusable entity hierarchy saved as a YAML `.bundle` file — Doriax's prefab. Build an entity once (a configured enemy, a UI panel, a streetlight with light and physics), turn it into a bundle, and instance it as many times as you need. Editing the bundle updates every instance.

A `.bundle` file can live in **any folder of your project directory**, so you can organize bundles alongside the assets they use (`enemies/`, `ui/`, `props/`, etc.).

Creating a bundle

1. Build and configure the entity (and its children) in the scene.
2. In the Resources Browser, navigate to the folder where the bundle should be saved.
3. Either **right-click the entity** in the Structure panel and choose **Save as Bundle**, or **drag the entity** from the Structure panel and drop it on the Resources Browser.



Both routes are the same operation, and both save into the folder the Resources Browser is currently showing — the **Save as Bundle** tooltip names that folder so you can check it before committing. The menu entry is disabled for locked entities and for entities without a `Transform`.

The editor writes `<EntityName>.bundle` into the open folder (a numeric suffix is added if the name is taken) and **replaces the dragged hierarchy with an instance of the new bundle** — the entities now live in the bundle, and the scene holds an instance root pointing at the file. Dragging a multi-entity selection works the same way and stores all of the selected branches in one bundle. The whole operation is undoable.

The instance root takes the entity's position, rotation, and scale, so the bundle stores its content at the origin and each new instance starts at its own root. A multi-entity selection puts the root at the selection's center, where the move gizmo sits, with no rotation or scale. Anchored UI keeps its root at the origin, since its anchors already place it.

> **Values relative to the parent are not converted**
>
> As when an entity is moved under a new parent, values expressed relative to the saved entity's parent are not converted: a camera's **Target** when **Use Target** is on, and the transform values that actions, keyframe tracks, or scripts use on the entity itself. To keep them working, put that entity under an Empty object (p. 75) and save the parent as the bundle.

Instancing a bundle

Drag the `.bundle` file from the Resources Browser onto the **Structure panel**:

Drop target	Result
Scene root node	Instance added at the top level of the scene
An entity with `Transform`	Instance added as a child of that entity

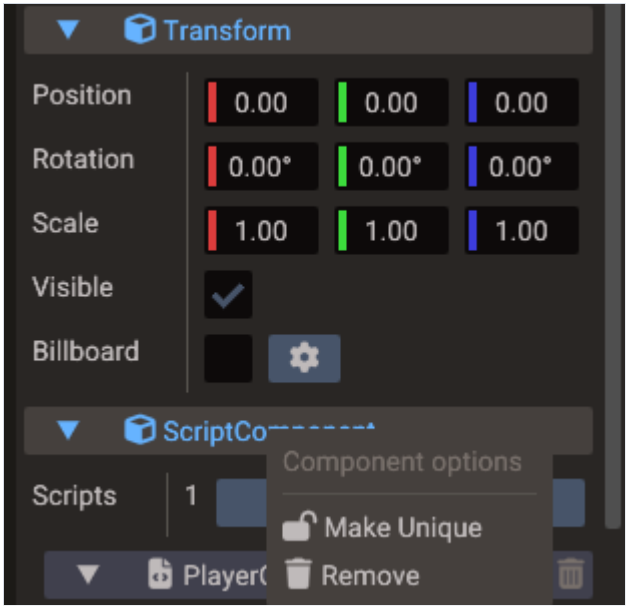
Each drop creates an independent instance with its own root `Transform`, so instances can be positioned, rotated, and scaled individually. Bundles can also be instanced inside other bundles (nested bundles).

How instances stay in sync

The bundle owns the component data. All instances read from it, so:

- **Editing a component inside any instance edits the bundle** — the change appears in every instance of that bundle, in every scene that uses it.
- **To customize a single instance**, right-click the component header in the Properties window and choose **Make Unique**. That component becomes a local override for this instance and stops following the bundle.
- **Revert to Bundle** on an overridden component discards the local values and re-links it to the bundle.

In the Properties window, a component that still follows the bundle has a cube icon and a blue header; once it is made unique it is drawn like any other component.



The instance root itself is not part of the bundle, so its components are always per-instance. Its `Transform` is what lets you place each copy somewhere different, and it cannot be removed. Any other component you add to the root, such as a script, stays on that instance only and is saved in the scene, not in the `.bundle` file.

Models inside a bundle

A bundle shares a **Model** component's file and its **Merge static model** setting, not its nodes: each instance builds its own joints, parts, and animations from the file when the scene loads, so they are not stored in the scene. After **Make Unique**, loading another file into the Model creates the nodes in the editor, and those are saved with the scene like any other entities.

Revert to Bundle on a Model removes the instance's nodes and builds them again from the shared file. Entities you attached to the old nodes, such as a prop on a hand bone, move to the model entity and keep

their place in the world, because anything under rebuilt nodes would not be saved. Undo puts them back on their nodes.

Model materials inside a bundle

Editing a submesh of an imported model stores the change on the member's **Model** component, so it follows that component's shared or unique state: shared reaches every instance and the `.bundle` file, **Make Unique** on the Model keeps it to one instance. See 3D Graphics — Editing an imported model's submeshes (p. 254).

References to entities outside the bundle

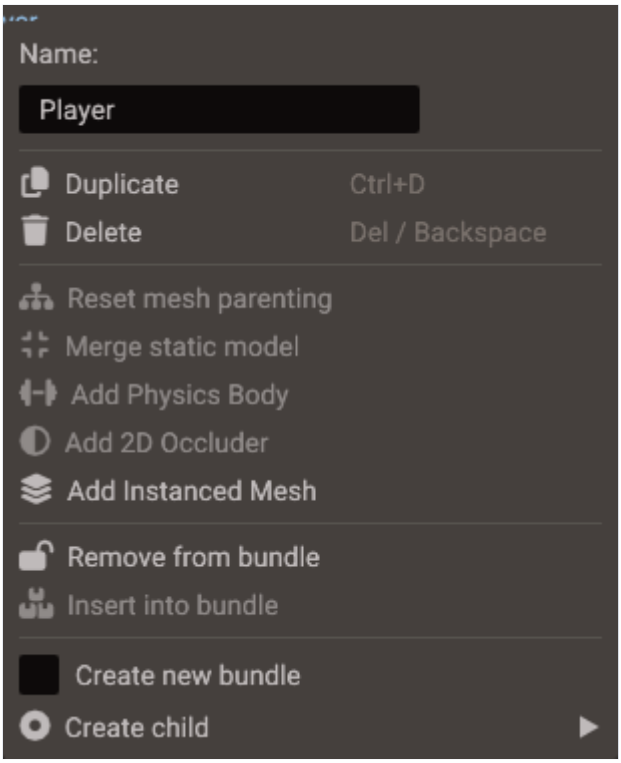
A component or script property on a bundle member can point at an entity that is **not part of the bundle** (some other entity in the scene). That wiring is **per-instance**: it is not stored in the `.bundle` file and does not propagate to other instances — each instance in each scene keeps its own assignment, and a freshly placed instance starts with the field set to **None**. References to entities **inside** the bundle stay shared as usual, including clearing them.

Per-instance references are not yet persisted across scene reloads: unless the component is a local override (**Make Unique**), the field comes back as **None** when the scene is reopened and needs to be reassigned.

Adding and removing member entities

Right-click entities in the Structure panel to change what belongs to the bundle:

Menu item	Where	Effect
Insert into bundle	A non-member child under a bundle instance	Entity becomes part of the bundle (appears in all instances)
Remove from bundle	A member entity inside an instance	Entity leaves the bundle but stays in this scene as a regular entity
Save as Bundle	Any entity outside a bundle	Saves the hierarchy as a **new** `.bundle` file and replaces it with an instance
Insert to Bundle ▸	Any entity outside a bundle	Moves the entity into one of the scene's **existing** bundles



New entities created under a bundle root from the Structure create menu can join the bundle directly. Nested bundle instances have matching *(nested)* menu entries to control membership in the outer bundle.

Saving

Bundle edits mark the bundle as modified, and modified bundles are written to their `.bundle` files **when you save the scene** (Ctrl+S). Unsaved bundle changes count toward the scene's unsaved-changes indicator and the close-confirmation prompts, so they are not lost silently.

Renaming or moving a `.bundle` file through the Resources Browser context menu updates the scenes that reference it.

Bundles in the build

Only the bundles the project uses are compiled into the game and registered with BundleManager (p. 597):

- every bundle **instantiated in a scene**, and
- every bundle listed in **Project → Bundles**.

A bundle that no scene instantiates — one that exists only to be spawned from a script — has to be listed, or `createBundle` reports `bundle 'name' not found` at runtime. Listing it also compiles the C++ scripts on its entities: each spawned instance runs its scripts, Lua and C++, the same way a scene does (see `createBundle` (p. 598)).

Project → Bundles lists every `.bundle` file in the project. The ones a scene instantiates are checked and disabled, with the scenes using them in the **Used by** column; check any other bundle to build and register it as well. The list is stored as `standaloneBundles` in `project.yaml`, follows the file when it is renamed or moved, and drops entries whose file is deleted.

In the Resources Browser, a `.bundle` the project does not build is drawn dimmed, and so is a `.scene` file that is not part of the project.

Instances in an exported build

An exported scene creates each placed instance through the bundle's generated factory, the same one `BundleManager` registers. Member entities get their ids when that factory runs, so the id a member has in the editor is not the id it has in a build, and generated code never hardcodes one: whenever the scene needs a member — a **Make Unique** override, a non-member child kept under a member, or a component or script property on a scene entity that points into the instance — it takes the entities the factory created and addresses them by position. Overrides and references into instances therefore behave in a build exactly as they do in the editor, nested bundles included.

Scripts should reach an instance's entities by name under its root with `findEntity(name, root)` (p. 379), never by an id copied from the editor.

Bundles vs child scenes

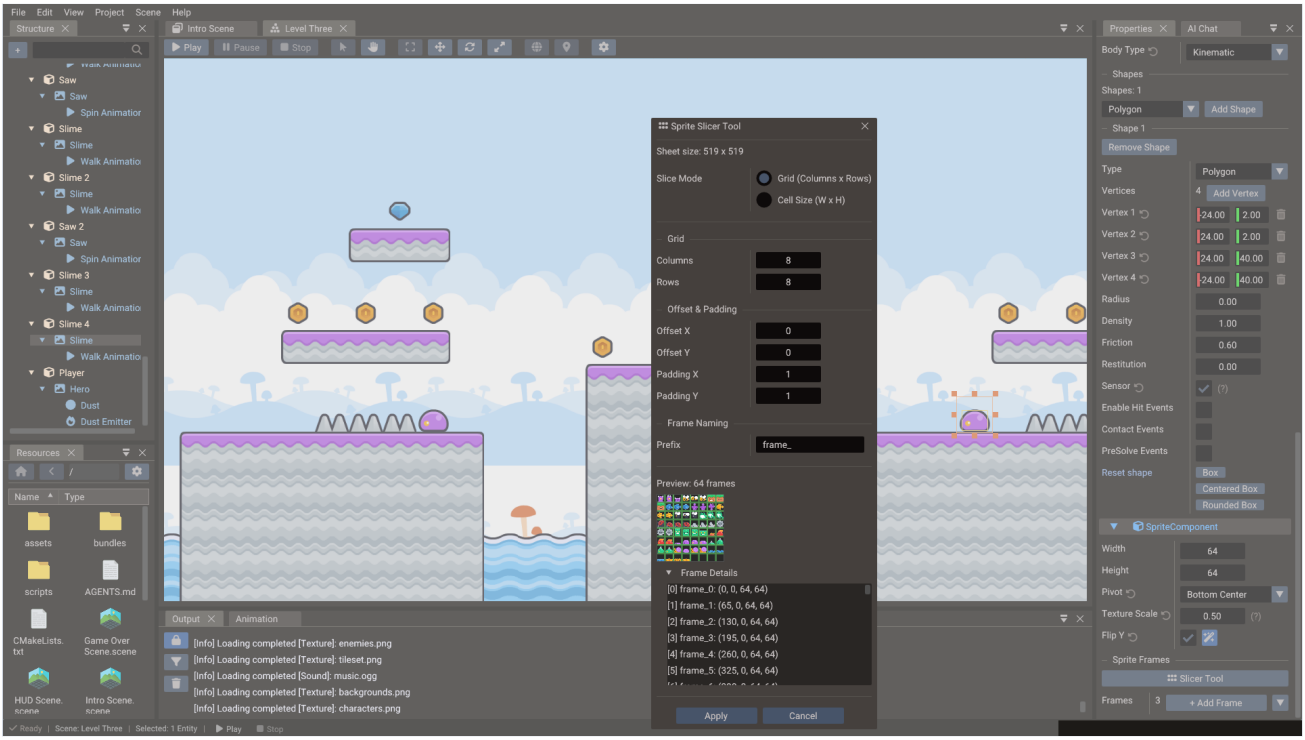
	Bundle	Child scene
File	`.bundle`	`.scene`
Use for	Repeated entity hierarchies (props, enemies, UI widgets)	Large scene sections, levels, screens
Instances per scene	Many	One reference per child scene node
Per-instance overrides	Yes (Make Unique per component)	No

See also

- Structure Panel (p. 73) — the tree where bundles are created and instanced
- Resources Browser (p. 106) — where `.bundle` files live
- Properties & Components (p. 90) — component editing and override menus

Sprite Slicer

The **Sprite Slicer** tool cuts a single texture into a series of named frames that the engine can use for sprite animation, atlas-based rendering, or individual sprites. Once sliced, frames are referenced by name or index and do not require the original atlas to be split into separate files.



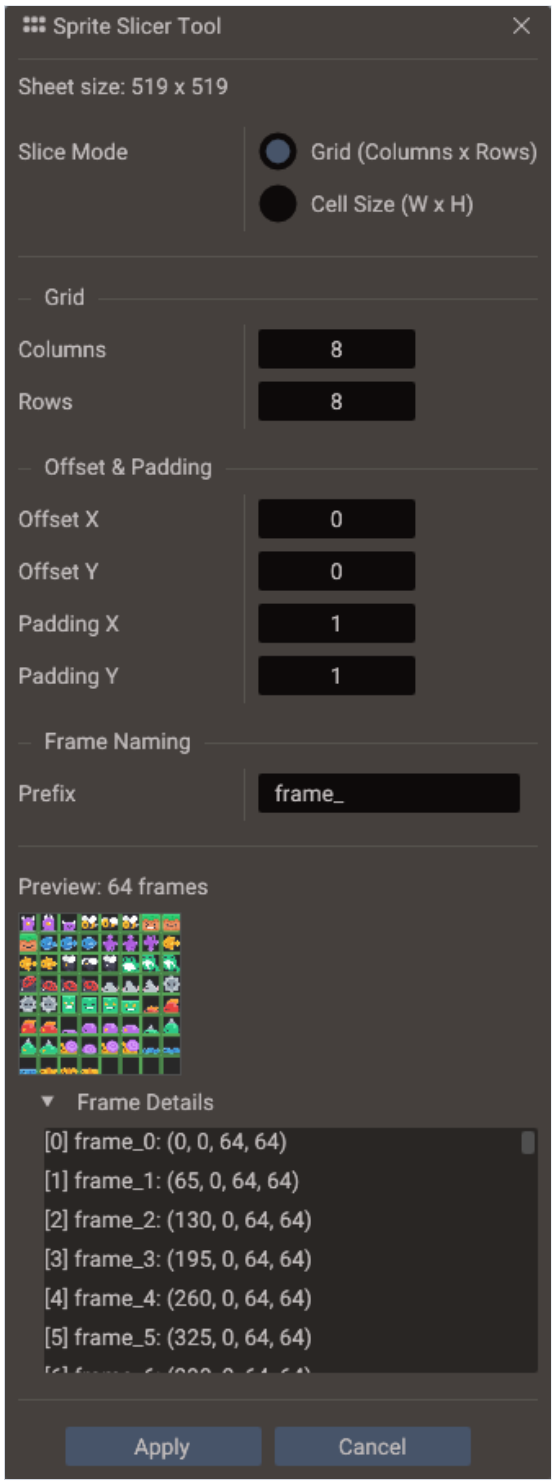
Opening the Sprite Slicer

Select an entity with a **Sprite** component. In the Properties window (p. 90), click **Slicer Tool** in the component's **Sprite Frames** section. The slicer works on the texture the sprite already uses, so assign the sprite sheet first.

Slice modes

Mode	When to use
Grid (Columns x Rows)	You know how many frames the sheet has across and down; the cell size is derived from the sheet size
Cell Size (W x H)	You know the pixel size of one frame; the number of columns and rows is derived

Both modes cut a regular grid, the fastest workflow for sprite sheets such as animated characters, explosion effects, or item icons. For frames of different sizes, slice the regular part and adjust the rects in the component's **Sprite Frames** list afterwards.



Grid slicing

1. Pick a slice mode, then enter **Columns** and **Rows**, or **Cell Width** and **Cell Height**.
2. Set **Offset X / Y** (inset of the first cell) and **Padding X / Y** (gap between cells) if the sheet has margins or gutters.
3. Change the **Prefix** under **Frame Naming** if you want names other than `frame_0`, `frame_1`, …. Frames are numbered left to right, top to bottom.

4. Check the **Preview**: it shows how many frames the grid produces over a thumbnail of the sheet. Expand **Frame Details** to see each frame's name and rect in pixels. Cells that run past the edge of the sheet are clipped to it.
5. Click **Apply**. The sprite's frame list is replaced with the new frames, as one undoable step.

Frame names are used by `Sprite::setFrame(name)` and `Sprite::startAnimation(name, …)`. `SpriteAnimation::setAnimation` uses frame indices; identify the action entity by its entity name if you need a searchable label.

Working with frames at runtime

After slicing, use frame names or indices in scripts and components:

Lua

```lua
sprite = Sprite(scene)
sprite:setTexture("characters/hero.png")

-- Show a specific frame by name or index
sprite:setFrame("walk_01")

-- Animate through a frame index range (interval in milliseconds)
anim = SpriteAnimation(scene)
anim:setTarget(sprite)
anim:setAnimation(1, 8, 100, true)   -- startFrame, endFrame, interval (ms), loop
anim:start()
```

C++

```cpp
Sprite hero(&scene);
hero.setTexture("characters/hero.png");

// Show a specific frame by name or index
hero.setFrame("walk_01");

// Animate through a frame index range (interval in milliseconds)
SpriteAnimation anim(&scene);
anim.setTarget(&hero);
anim.setAnimation(1, 8, 100, true);
anim.start();
```

Tips

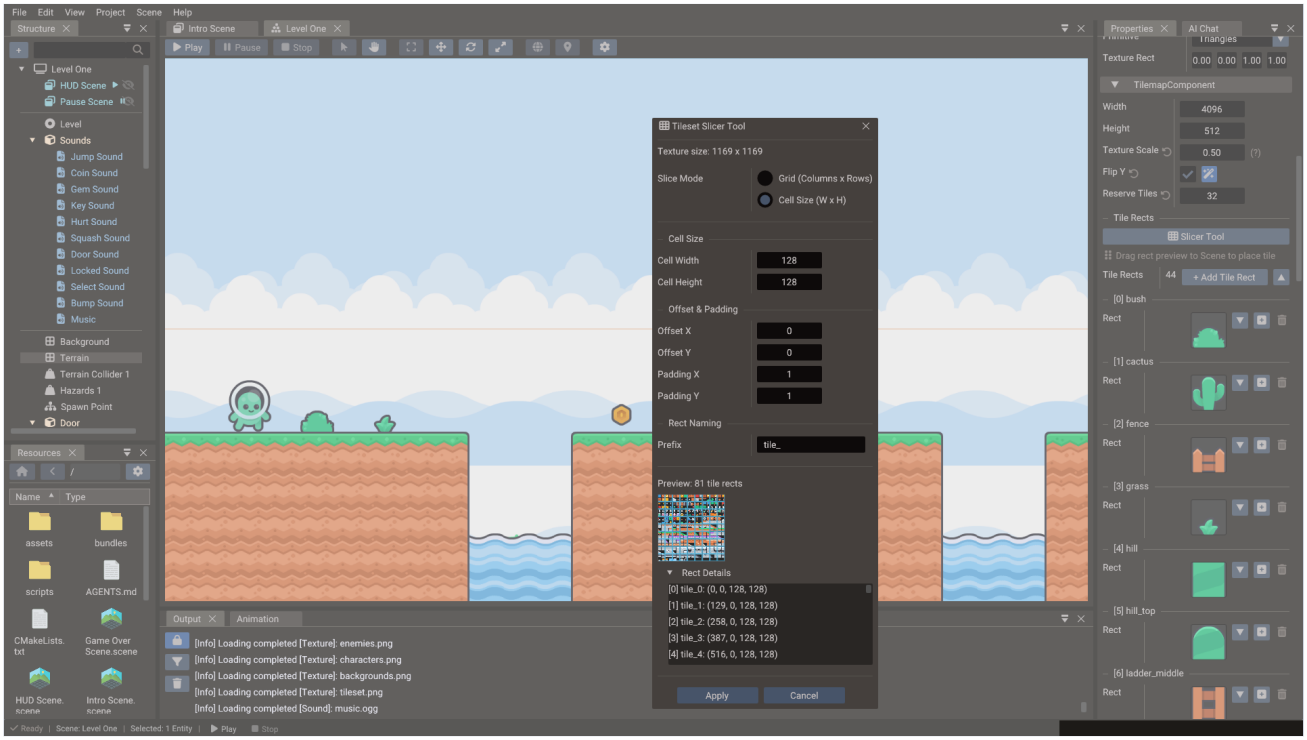
- Keep sprite sheets power-of-two in size for best GPU compatibility.
- Use consistent frame naming conventions (`walk_00` … `walk_07`) so animation ranges are easy to specify.
- A single texture can contain multiple animation sequences — just name them clearly.
- For tilemaps and tilesets, use the Tileset Slicer (p. 119) instead, which is designed for tile index workflows.

See also

- Tileset Slicer (p. 119)
- Sprite (p. 480)
- SpriteAnimation (p. 586)
- 2D Graphics (p. 243)

Tileset Slicer

The **Tileset Slicer** tool divides a tilemap's texture into uniformly-sized tile rects. Each rect gets a name and a numeric ID — its index in the tilemap's rect list. Those IDs are the values you store in `TileData` cells and reference from `Tilemap` objects in script.



Opening the Tileset Slicer

Select an entity with a **Tilemap** component. In the Properties window (p. 90), click **Slicer Tool** in the component's **Tile Rects** section. The slicer cuts the tilemap's texture; when the tilemap uses several textures, pick the one to slice at the top of the dialog.

Setting up the grid

Field	Purpose
Grid (Columns x Rows) / Cell Size (W x H)	Slice by tile count or by tile size in pixels
Columns / Rows	Number of tiles across and down; the tile size is derived from the texture size
Cell Width / Cell Height	Pixel size of one tile; the number of columns and rows is derived
Offset X / Y	Pixel inset of the first tile
Padding X / Y	Pixel gap between tiles, for tilesets with gutters
Prefix	Rect name, followed by its number: `tile_0`, `tile_1`, ...

The **Preview** shows how many rects the grid produces, and **Rect Details** lists each one in pixels. Click **Apply** to replace the rects of the sliced texture; rects that belong to the tilemap's other textures are kept, and the new ones follow them, numbered left to right, top to bottom. The change is one undoable step.

Tile IDs

Tile IDs are plain integers. In a tilemap with a single texture, the tile at column 0, row 0 is ID `0`; column 1, row 0 is ID `1`; and so on. Use these IDs when filling `TileData` in script.

```
-- Create a tilemap, assign a tileset texture, and place tiles
tilemap = Tilemap(scene)
tilemap:setTexture("world/tileset.png")

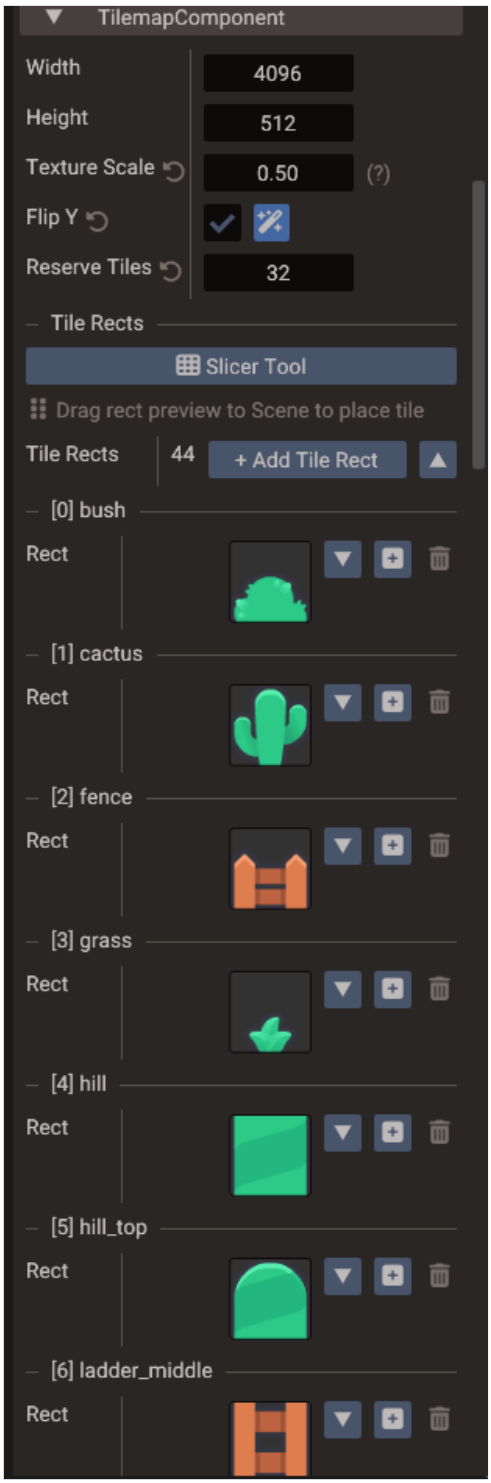
-- Define texture regions (rects), then place tiles that reference them
tilemap:addRect(Rect( 0, 0, 32, 32))                -- rect ID 0
tilemap:addRect(Rect(32, 0, 32, 32))                -- rect ID 1

tilemap:addTile(0, Vector2( 0, 0), 32, 32)          -- rect ID, position, width, height
tilemap:addTile(1, Vector2(32, 0), 32, 32)
-- ... or paint with the rect IDs generated by the slicer directly
```

Painting tiles in the editor

After slicing, the TilemapComponent's **Tile Rects** list shows a preview of each rect. To place tiles:

1. Select the **Tilemap** entity.
2. In the **Properties window**, find the rect in the **Tile Rects** list.
3. Drag its preview into the viewport to place a tile there.



When arranging placed tiles in the Scene View (p. 81), enable **Snap tile** in the viewport gear menu so drops and gizmo moves snap to each tile's width and height — useful for packing tiles flush without tuning the global grid.

Adding collision

Tiles are only drawn, so collision comes from a **2D Physics Body** on the Tilemap entity:

1. Select the Tilemap entity and add a **2D Physics Body**. Leave its type on **Static**.
2. Add a **Polygon** shape and set its **Source** to **Entity Tilemap**.

The shape now covers the tiles with boxes. Tiles that touch are joined, first into rows and then into blocks of rows with the same width, so a flat floor is a single box and characters don't catch on the seams between tiles. The boxes are drawn in the viewport while the body is selected, and the shape's **Boxes** row counts them.

To leave out tiles that shouldn't block anything, like grass, clouds, or background props, click the weight button on their rect's row in the **Tile Rects** list. It dims when Collision is off, and every tile placed from that rect is then skipped.

Every tile becomes a full square, so a ramp tile blocks like a wall. For slopes, and for one-way platforms that need their own filter or events, turn Collision off on those rects and add a **Polygon** or **Chain** shape that follows them.

The boxes are built from the tiles each time the body loads, so tiles you paint in the editor are included the next time you play. If a script adds or removes tiles during the game, set `needUpdateShapes` on the body's component afterwards. See createTilemapShape (p. 547).

If you want to change single boxes, for example to give a wall less friction or turn a ledge into a sensor, click **Fit to Tiles** in the body's **Shapes** section instead. It replaces the body's shapes with plain Polygon boxes over the same tiles, which you can then edit like any other shape. They don't follow later tile changes, so click **Fit to Tiles** again after painting. See Physics (p. 317).

Tips

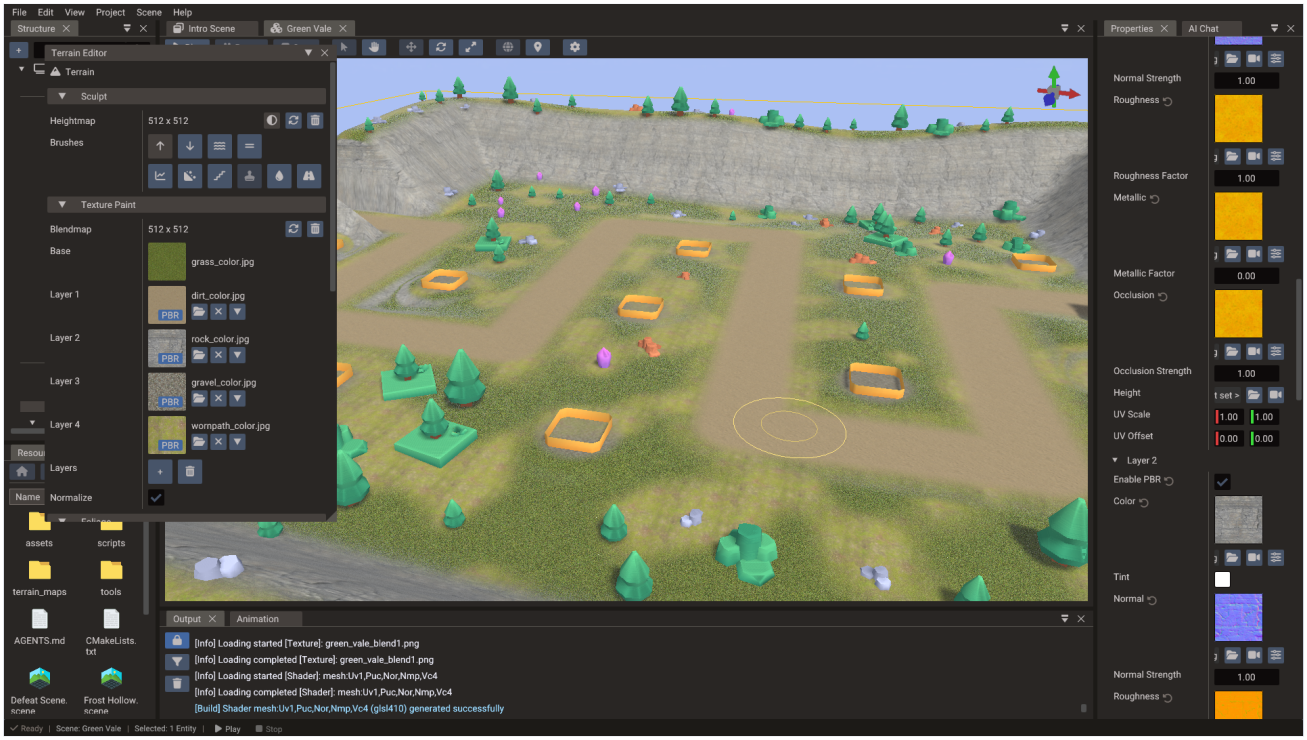
- Use power-of-two tile sizes (16×16, 32×32, 64×64) for consistent UV mapping.
- Keep the tileset image as one file even if the game uses multiple tile themes — tile ID ranges can be managed logically in data.
- For sprite animation and character frames, use the Sprite Slicer (p. 116) instead.
- Export-friendly naming: use lowercase filenames and avoid spaces so tile resource paths stay consistent across platforms.

See also

- Sprite Slicer (p. 116)
- Scene View (p. 86)
- Tilemap (p. 492)
- TileData (p. 719)
- TileRectData (p. 719)
- 2D Graphics (p. 243)

Terrain Editor

The **Terrain Editor** is the authoring window for entities with a Terrain (p. 309) component. It paints the terrain's maps — heightmap, blend maps, and foliage density maps — and places props on the surface, all with the same brush.



Opening the window

Select the terrain entity and click **Open Terrain Editor** in the Properties (p. 90) window, or choose **View → Terrain Editor**. The window can also stay open while you work: it follows the selection, and shows a hint instead of the tools when the selected entity is not a terrain or a scene is playing.

Terrain editing is disabled during play mode.

Maps

Every brush writes into a texture, and each map section has the same row:

Control	Action
Thumbnail / size	The current map and its pixel size
Resolution field	Size used when a new map is created
Create	Creates the map at the chosen resolution and assigns it to the terrain
Recreate	Replaces the map with a fresh one at its current resolution
Remove	Clears the map from the terrain
Start at middle height (heightmap only)	New heightmaps start at half height so the first stroke can lower as well as raise

Painted maps are written as PNG files under `terrain_maps/` inside the project's assets directory, so they are referenced, exported, and version-controlled like any other texture. Heightmaps are stored 16-bit to keep gentle slopes free of terracing.

Files are written in the background while you paint, and every scene save flushes the pending writes and deletes the maps no scene, bundle, or material still references. If a write fails the editor says so and keeps the scene marked unsaved, so the next save retries it.

Sculpt

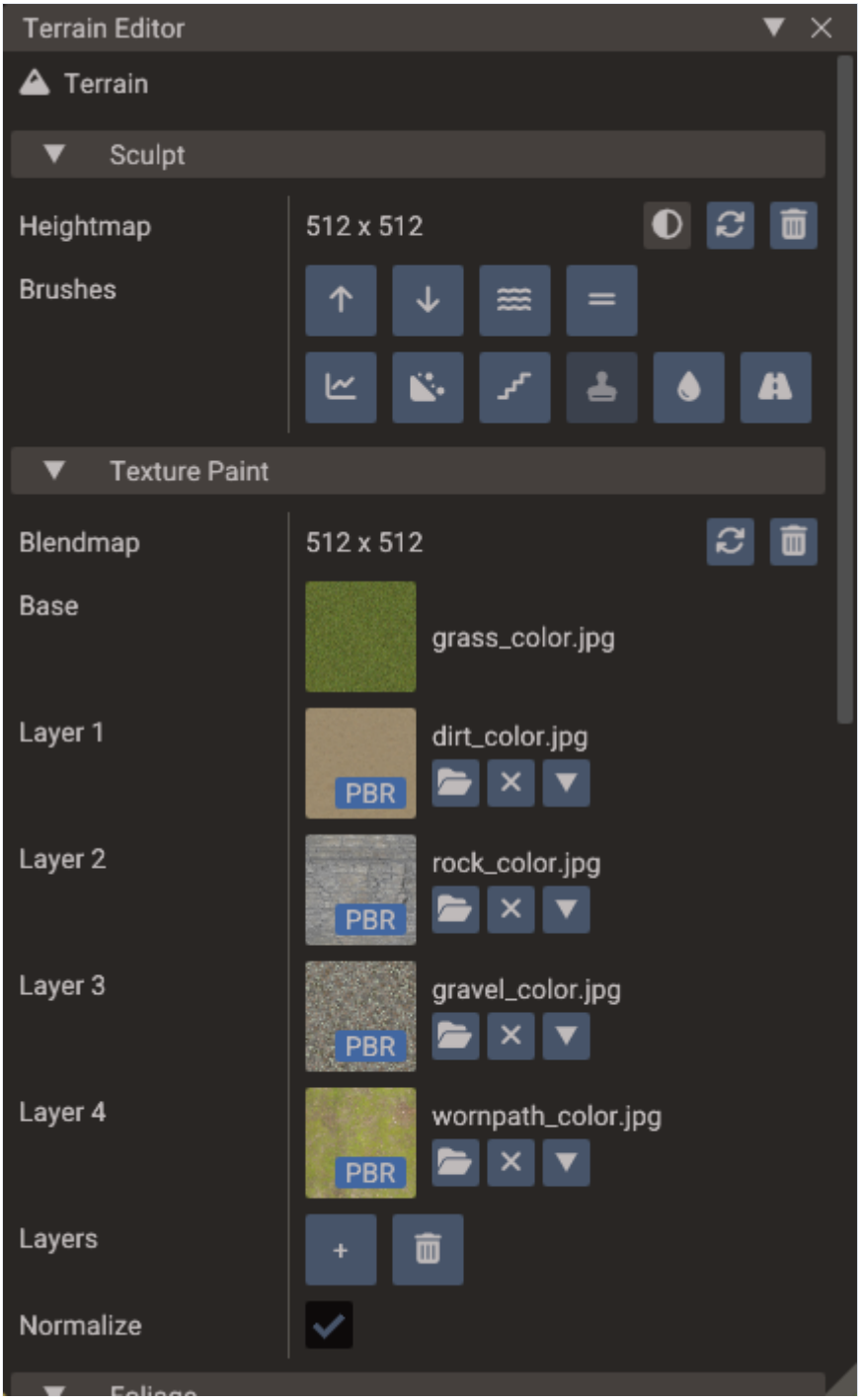
Sculpt brushes write the heightmap. They need a heightmap to exist first — the buttons stay disabled until one is created.

Brush	Effect
Raise	Pushes the surface up (Ctrl lowers, Shift smooths)
Lower	Pulls the surface down (Ctrl raises, Shift smooths)
Smooth	Averages each texel with its neighbours
Flatten	Levels toward a target height (Shift smooths)
Sharpen	The smooth kernel pushed the other way — pulls ridges back out of an over-smoothed surface
Noise	Adds fractal value noise to break up flat ground
Terrace	Snaps heights to evenly spaced flat levels
Stamp	Stamps the brush mask into the surface as relief (needs a brush mask)
Erode	Hydraulic droplets cut channels and drop sediment; thermal talus settles slopes steeper than the angle of repose
Ramp	Drag from one point to another to lay a ramp between the heights the stroke started and ended at

Flatten takes its target from **Sample height** (picked from the terrain where each stroke begins) or from an explicit **Flatten height**. **Noise** exposes a **Noise size** — the width of one noise feature in world units, anchored to the terrain so the field stays put as the brush moves. **Terrace** exposes **Terrace steps**, the number of flat levels the height range is cut into.

Texture Paint

The terrain's ground is a base texture plus up to nine detail **layers**. Layers are grouped in threes: each group is weighted by the RGB channels of one blend map, and the **Blendmap** row above the list creates the map for the group the selected layer belongs to.



Row	Meaning
Base	The material's base color texture. Painting it clears the blend map the selected layer sits on, letting the base show through
Layer 1…9	A painted surface. Click the thumbnail to paint that layer; use the folder/clear buttons or drop an image from the Resources Browser (p. 106) to assign its color
▾ (per layer)	Opens that layer's settings under the row
Layers	Add or remove the last layer, up to nine (three per blend map)
Normalize	Fades the other layers sharing the same blend map as you paint. Layers on other blend maps keep their weight

Painting is a blend-map write, so the layer textures themselves are never modified.

See Detail layers (p. 309) for how the layers are blended at runtime.

Layer settings

The arrow beside a layer's clear button opens its settings under the row. A layer with **Enable PBR** off shows only what it can use — the rest appears once it is on, and the layer's preview is marked **PBR** in the list.

Setting	Meaning
Enable PBR	Off blends only the layer's color and leaves the rest to the terrain material. On gives the layer its own surface
Copy from material	Fills the layer from a `.material` file. Dropping one on the layer does the same
Tint	Multiplies the layer color
Normal + **Normal Strength**	Tangent-space normal map and how far it tilts the surface
Roughness + **Roughness Factor**	Map read from green (or the only channel of a grayscale map), multiplied by the factor
Metallic + **Metallic Factor**	Map read from blue, same fallback and multiply
Occlusion + **Occlusion Strength**	Map read from red, and how far it darkens ambient light
Height	Biases the blend toward this layer where it is taller — it never moves geometry
UV Scale / **UV Offset**	Tiling for this layer alone, multiplying the shared detail tiling

Copying a material takes its base color and tint, normal, metallic/roughness, occlusion and factors, and turns the layer into a PBR one. Emission and transparency are not part of a terrain layer and are left out; the layer keeps the tiling it already had.

Slope and height mask

While a paint brush is selected the Brush section grows a **Mask** toggle that restricts painting to a range of ground **slope** (degrees) and **height** (normalized against the terrain's max height). Texels outside the range keep what they had — the way to paint "rock only above 40 degrees" without tracing the cliffs by hand.

Foliage

A foliage layer scatters a mesh over the terrain from a painted density map. Instances are resolved from that map at runtime rather than stored in the scene, so a dense field costs a density map and a set of parameters.

Add a layer, pick its **mesh** (a model file), create its **Density map**, then paint with **Paint density** (Ctrl erases) or **Erase density** (Ctrl paints).

Setting	Meaning
Density	Instances per square world unit where the map is fully painted
Scale range	Minimum and maximum random scale per instance
Slope range	Allowed ground slope in degrees, 0 (flat) to 90 (vertical)
Height range	Allowed ground height, 0 (terrain base) to 1 (max height)
Rotation	Random yaw as a share of a full turn
Normal alignment	Blends from upright (0%) to lying along the surface normal (100%)
Draw distance	Visibility distance in world units; instances fade in over the last quarter of it
Min distance	Instances nearer than this are skipped. A copy of the layer with a simpler mesh, a min distance near this layer's draw distance and a longer draw distance becomes its far detail level
Cast shadows	Put this layer into the shadow cascades. Turn off for grass-scale scatter so it still draws in the colour pass without the extra shadow geometry
Seed	Change it to reshuffle instance positions

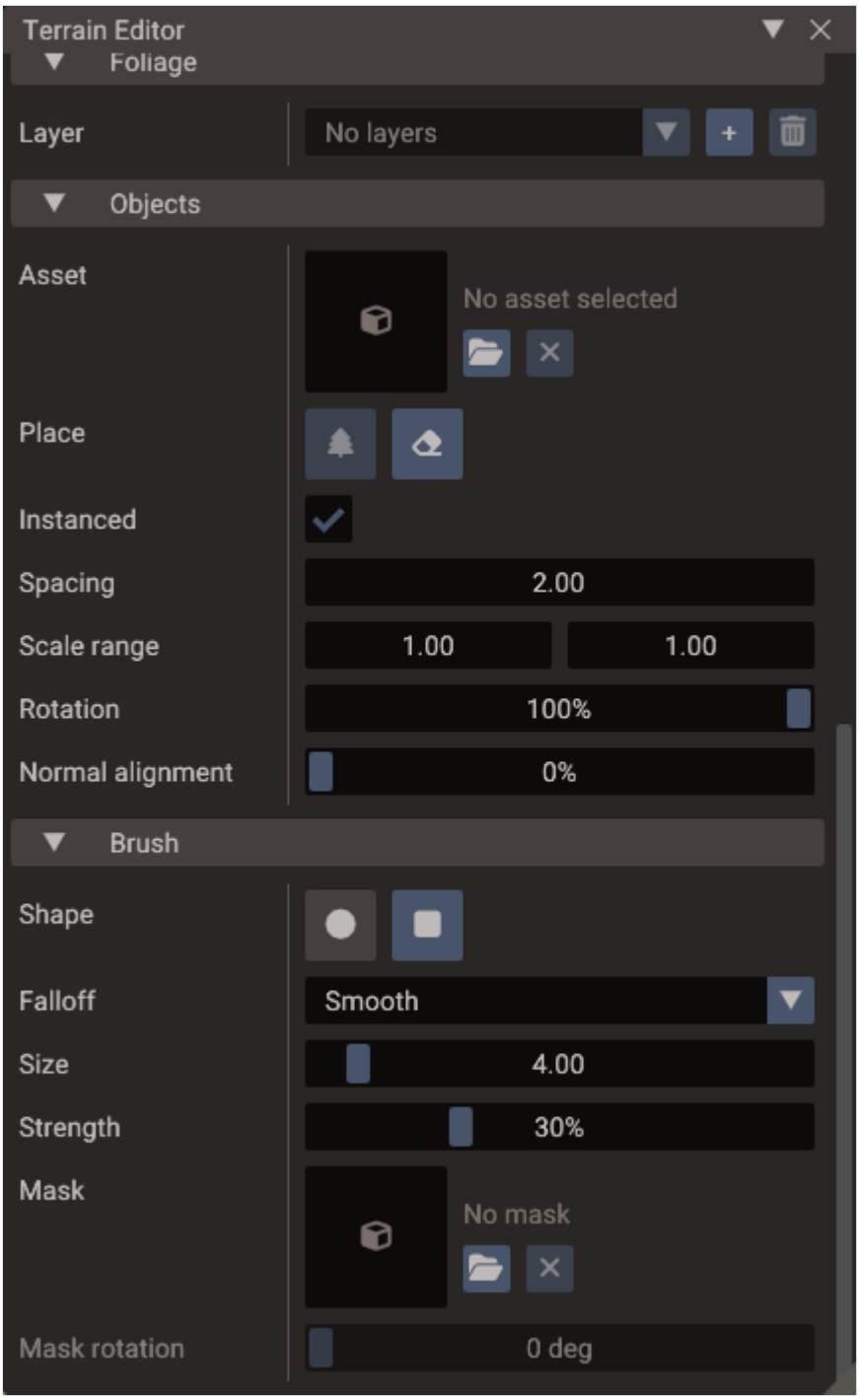
Foliage entities are created by the engine, not authored: they never appear in the Structure panel (p. 73), and clicking one in the viewport selects its terrain.

Draw distance while editing

While the Terrain Editor is open on a terrain, that terrain's foliage is scattered over its whole surface so you can see what you painted. **Draw distance** and **Min distance** apply again when the window closes, the selection changes, or play mode starts.

Objects

The **Objects** section places real props — models or entity bundles (p. 111) — along the drag, as children of the terrain entity.



Setting	Meaning
Asset	The model or `.bundle` to place; browse or drop one from the Resources Browser

Setting	Meaning
Instanced	Batches every object of this asset into a single instanced draw. Off is one entity per object
Spacing	Closest two placed objects are allowed to get, in world units
Scale range	Random scale per object; left at 1 a model keeps the scale its file authored
Rotation	Random yaw as a share of a full turn
Normal alignment	Upright (0%) to aligned with the terrain normal (100%)

Place objects scatters within the brush radius (Ctrl erases); **Erase objects** removes the terrain's child props under the brush (Ctrl places). Only meshes and bundle roots count as props — a light or a camera parented to the terrain is neither erased nor counted for spacing.

Instanced placement keeps one host entity per asset under the terrain and adds an instance per object. Instances can be moved, but carry no components of their own, so turn **Instanced** off for props that need collision, scripts, or animation. Bundles are hierarchies of entities and are always placed un-instanced.

A whole drag is one undo step, whichever mode it used.

Brush settings

Setting	Meaning
Shape	Circle or square
Falloff	Smooth, Linear, or Constant — how strength fades from the center to the edge
Size	Brush radius in world units; for placement, how far objects scatter from the cursor
Strength	Flow per second while the button is held (not applicable to placement)
Mask	A grayscale image that multiplies the falloff — and the relief the **Stamp** brush lays down
Mask rotation	Turns the mask under the brush

Because strength is a flow rate, holding the brush in place keeps building up the effect, and stroke speed does not change how much a slow frame applies.

Brush settings, the selected brush, map resolutions and the placement palette are stored per project, so they come back the way you left them.

Shortcuts

Shortcut	Action
[/ **]**	Shrink / grow the brush
Shift+[/ **Shift+]**	Lower / raise brush strength
Ctrl (held)	Inverts the brush: raise↔lower, paint↔erase density, place↔erase objects
Shift (held)	Smooths, for the height brushes that support it

Undo

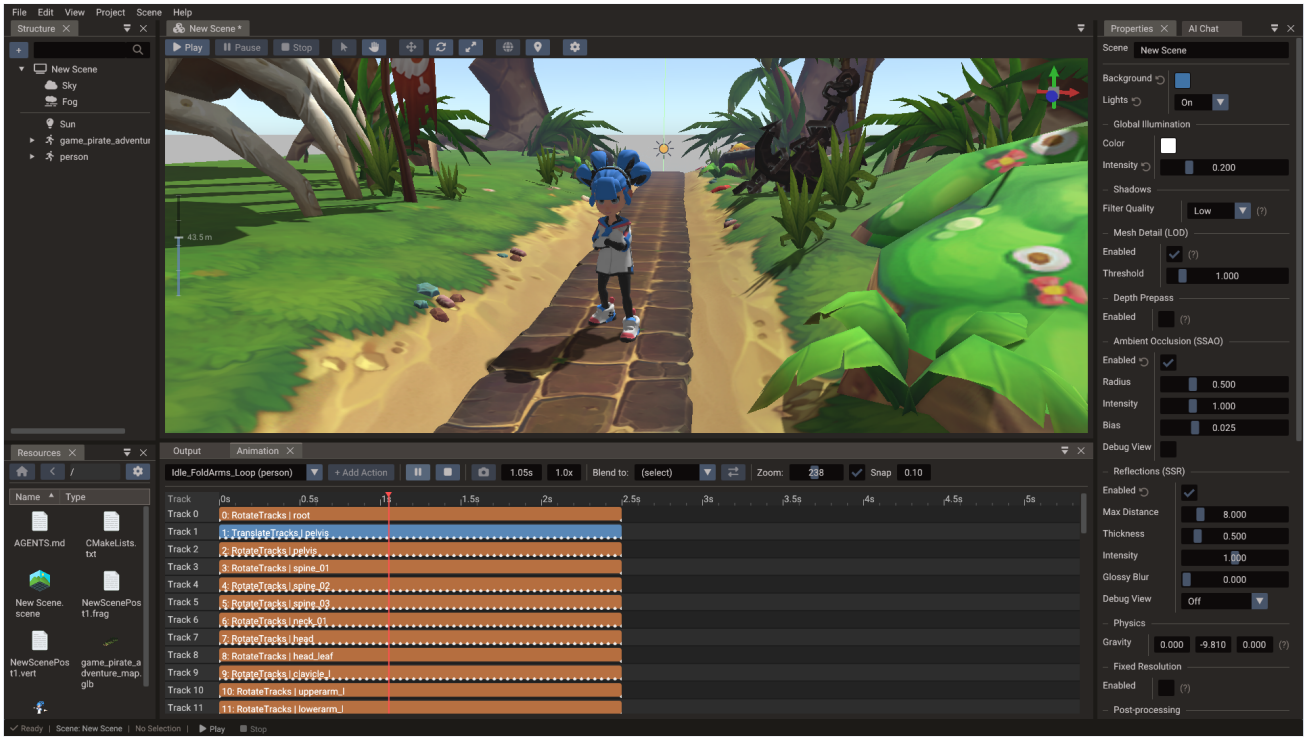
Each stroke is one undo entry. Map strokes record only the rectangle of texels they changed rather than a copy of the whole map, so a long sculpting session stays cheap to undo.

See also

- Terrain (p. 309) — terrain concepts, LOD tuning, and the runtime API
- Terrain class reference (p. 486)
- Properties & Components (p. 90) — the Terrain component rows

Animation Timeline

The **Animation Timeline** edits time-based changes and previews them in the scene. It works with entity transforms, sprite frame sequences, skeletal animation clips, morph targets, and runtime action components — all driven by the `ActionSystem` at runtime.



Timeline concepts

Concept	Meaning
Playhead	Current time or frame being previewed
Track	A single animated property channel (e.g. position X, sprite frame, alpha)
Keyframe	A value recorded at a specific time on a track
Clip	A named time range containing one or more tracks
Interpolation	How values are calculated between keyframes (linear, step, ease, etc.)
Snap	Grid interval used when placing or moving keyframes
Loop	Whether the clip repeats when it reaches the end

Supported track types

Track type	What it controls
Transform	Position, rotation, scale (translate, rotate, scale tracks)
Color / Alpha	Color tint and opacity
Sprite frame	Active frame index for a `Sprite` atlas
Skeletal clip	Bone transforms from an imported GLTF animation
Morph weights	Blend shape weights for facial or deformation animation
Particle playback	Play/stop state of a `Particles` action

Action frames

An animation is a list of **action frames**. Each frame schedules an action entity to run at a **start time** on a **track** (a timeline lane, used only for visual organization — overlapping frames are re-laned automatically).

Add frames with the **+ Add Action** button (which creates a new action entity of the chosen type; one undo removes both the entity and its frame), or **drag an action entity from the Structure panel onto the tracks area** — a ghost block previews the target track, the grid-snapped start time, and the frame length before you drop. Invalid drops are rejected with an explanatory tooltip: entities without an `ActionComponent`, entities from another scene, and anything that would make an animation contain itself, directly or through nested animations.

Operation	How
Select a frame	Click the block (details appear in the toolbar)
Move a frame	Drag the block; dragging vertically changes track
Resize a frame	Drag the block's left or right edge
Scroll tracks vertically	When the track stack exceeds the viewport, use its vertical scrollbar or the mouse wheel
Zoom the timeline	Use the mouse wheel over the visible time ruler
Scroll a long timeline	When blocks, the authored duration, or the playhead exceed the viewport, use the horizontal scrollbar; its range includes trailing space
Open the action	Double-click the block to select the action entity
Remove a frame	**Properties → AnimationComponent → Actions** → trash button

Frame moving and resizing remain available while a timeline preview is paused. After the edit, the preview is evaluated again at the current playhead time. Timeline editing remains locked during active animation playback and scene play mode. If playback or preview shutdown interrupts an active timeline drag, the pending edit is finalized as its normal undo step before the timeline locks.

Automatic duration

A frame whose duration is `0` is **auto**: it follows the action's own length — a `SpriteAnimation` lasts as long as its frame sequence, a `TimedAction` uses its duration, a nested `Animation` its own total length. Frames created in the editor default to auto, so configuring the action later updates the timeline block automatically.

Resizing a block in the timeline converts it to an **explicit** duration. To switch back and forth, use the checkbox next to **Duration** in the Properties window (**AnimationComponent → Actions**): checking it stores `0` (auto) and displays the resolved duration read-only; unchecking freezes the current resolved value as an editable number.

Creating a clip

1. In the **Structure panel**, create an **Animation** entity (**Create entity → Animation → Animation**). Creating it as a child of the entity to animate sets that entity as the animation's target; otherwise set **Target** in the Properties window (**ActionComponent → Target**).
2. Open the **Animation Timeline** and pick the animation in the clip selector.
3. Right-click empty track space at the first key time and choose **Key Position**, **Key Rotation**, **Key Scale**, or **Key Transform**. Missing transform tracks are created automatically as auto-duration action frames.
4. Scrub to another time, pose the target in the scene view or Properties, and click the camera **Snapshot** button. Repeat for each pose.
5. Press **Play** in the timeline to preview the clip.

You can also create action frames manually with **+ Add Action** or by dragging action entities from the Structure panel.

Keying transforms

Transform keying has two complementary controls: **Snapshot** for the complete pose at the playhead, and right-click keying for one block or selected channels.

A track with **Relative** checked stores offsets, not the pose: its keys hold the target's offset from where it was when the preview started (see Relative tracks (p. 304)). Keyed outside a preview, the offset is zero.

Snapshot

The camera **Snapshot** button next to **Stop** stores the current position, rotation, and scale values of every transform keyframe track reached by the playhead. All writes are grouped into **one undo step**.

A track is included when its action frame starts at or before the playhead:

- An **auto-duration** frame (`duration <= 0`) remains keyable after its current last key. Adding the key extends the action and grows its block.
- An **explicit-duration** frame is included only while the playhead is inside its fixed span, including the end. Snapshot does not extend it.
- Keys use time local to their own frame (`playhead time - frame start`), so moving a block along the timeline does not change where its keys belong.

Snapshot only keys tracks already present in the animation. If no eligible transform track is under the playhead, the button is disabled and its tooltip says **No keyframe tracks under the playhead to snapshot**. Use empty-area right-click keying when you want to create missing tracks.

Scrubbing starts a paused preview, and Snapshot remains available in that state. It is disabled during active animation playback and scene play mode. The button tooltip shows the exact snapshot time or explains why the action is disabled. Snapshot does not reset the displayed pose, and its keys remain after you stop previewing.

Right-click keying

Right-click while the scene is stopped and animation playback is paused:

Where	Result
A transform-track block	Inserts that block's position, rotation, or scale value at the clicked block-local time
Empty track space	Keys the animation target's position, rotation, scale, or complete transform

Block keying always targets the **clicked action**, even when same-channel blocks overlap. Its clicked time must be inside an explicit-duration block's real span; the menu disables insertion when the block's minimum visual width extends beyond that span.

Empty-area keying reuses the latest matching block active at the clicked time. It can extend auto-duration blocks, but it will not write past an explicit block's end. When the requested channel does not exist yet, it creates the track and an auto-duration action frame. **Key Transform**, including any track creation, is one undo step; single-channel keying is one undo step as well.

Authoring past the current end

The playhead and time field can move beyond the current clip duration (up to one hour), so auto-duration tracks can grow. Preview evaluation still stops at the clip's current end, however: until you extend it, the object holds its **end pose** at the later authoring time. The Snapshot tooltip and right-click menu warn when this happens.

This held pose is useful for pose-to-pose authoring, but it is not a newly evaluated future pose. After scrubbing past the end, adjust the target before keying if you want different values there.

Keyframes appear as small **diamonds** along the bottom of track blocks.

Editing keyframes

Keyframes of track actions show as diamonds on their timeline blocks. Click a diamond to select it; the selected key is highlighted. Drag it horizontally to change its track-local time using the current timeline snap interval. A key cannot cross its neighbors or move outside an explicit-duration block, and the completed drag is one undo step. Auto-duration blocks grow when their last key moves later.

Values, interpolation, and deletion are edited in the **Properties** window with the track entity selected (double-click its block to select it):

Operation	How
Move a keyframe in time	Drag its timeline diamond horizontally, or edit its **KeyframeTracks** time entry
Change a keyed value	Edit the entry in the track component's values list
Change interpolation	Under **KeyframeTracks → Easing**, click **Add Ease**, expand the list, and pick a curve (`Ease 0 - 1`, ...)
Remove explicit easing	Expand **Easing** and use the trash button beside the entry; later entries shift to the preceding segments
Delete a keyframe	Trash button next to the time entry (easings and cubic tangents stay aligned)
Move position keys visually	Drag the path handles in the scene view (p. 87)

Sprite animation

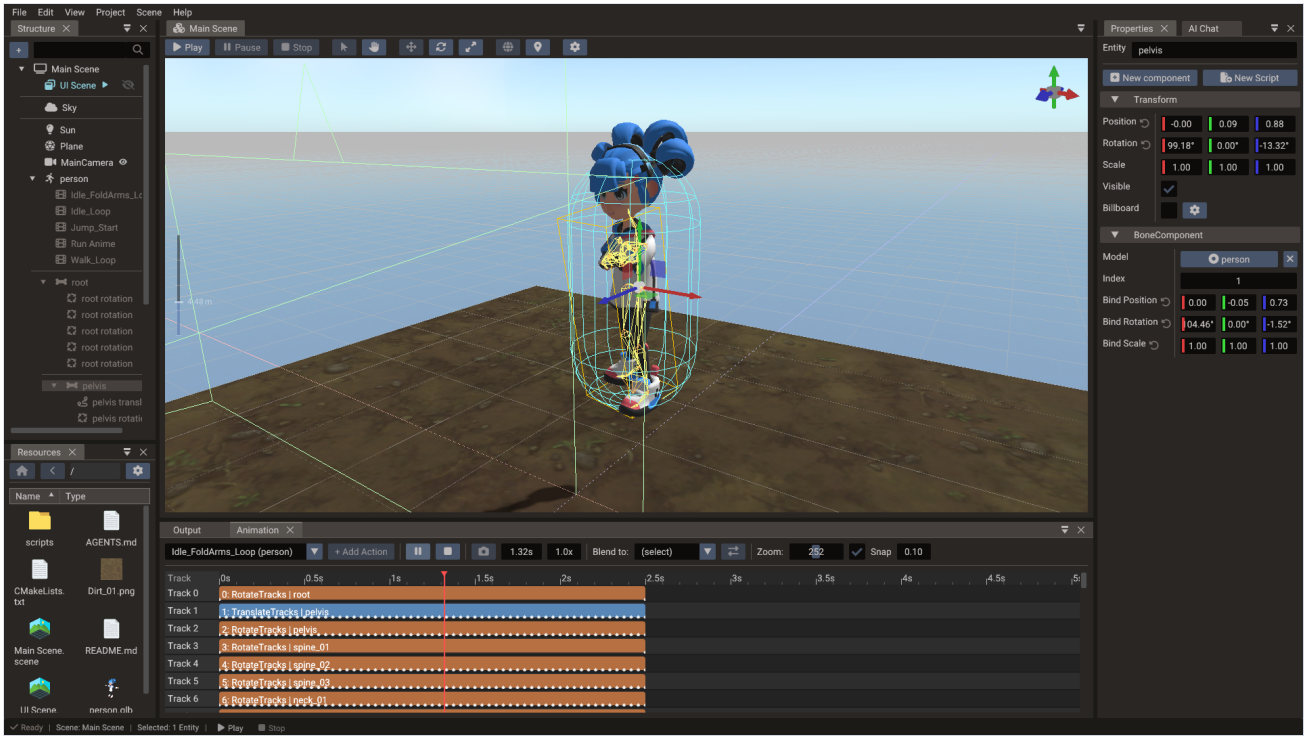
For 2D frame animation, add a **Sprite Frame** track and set keyframes to specific frame names or indices from the sliced sprite sheet. Keep frame intervals consistent for smooth playback.

The runtime equivalent is `SpriteAnimation`. Identify a sprite animation by its **entity name** (Structure panel) — `SpriteAnimationComponent` has no separate Name field. See the Sprite Slicer (p. 116) page for how to prepare a sprite sheet.

Skeletal animation

GLTF models can include one or more named skeletal animation clips. Files with clips, or more than one skin, import the **full node tree** under the Model, so transform-only nodes animate as well as joints. The timeline editor lets you preview and blend these clips on the **Model** entity. Use the **Bone** view to inspect and edit individual joint transforms.

The animation selector uses each animation entity's name. Imported clips start with the name authored in the GLTF; you can rename the entity in the Structure panel. Runtime `Model:findAnimation` and `Model:playAnimation` string lookups use that same entity name.



At runtime, look up a clip by name on the `Model` object:

Lua

```lua
model = Model(scene)
model:loadGLTF("characters/hero.gltf")
local walk = model:findAnimation("Walk")
walk:start()
```

C++

```cpp
Model hero(&scene);
hero.loadGLTF("characters/hero.gltf");
Animation walk = hero.findAnimation("Walk");
walk.start();
```

Previewing crossfades between clips

At runtime you usually switch clips with a **crossfade** so the character eases from one motion into the next instead of snapping (see Smooth transitions (p. 303)). You can preview that blend directly in the timeline:

1. Select an animation clip and press **Play** to preview it.
2. In the toolbar, pick a clip in the **Blend to** dropdown.
3. Click the ⇄ button. The playing clip fades out while the chosen clip fades in, using the target clip's **Fade time**, and the viewport shows the blended result.

Each clip carries a **Fade time** (`defaultFadeTime`) — the default crossfade duration used when `Model:playAnimation` (p. 452) is called without an explicit time. Edit it in the **Properties** window under **AnimationComponent → Fade time** (~0.2–0.3s reads well for locomotion, ~0.1s for a quick hit or death). Scrubbing the playhead exits the transition and returns to the single selected clip.

> **Use looping clips to see the blend**
>
> The blend reads best when you crossfade *from* a looping clip (idle, run). Blending from a one-shot clip that has already finished will snap, because a finished clip no longer contributes to the pose.

Runtime action system

For scripted one-shot and looping motion that does not need a full authored timeline, the runtime action system provides lightweight action types:

Action	Purpose
`PositionAction`	Animate entity position
`RotationAction`	Animate entity rotation
`ScaleAction`	Animate entity scale
`ColorAction`	Animate material or UI color
`AlphaAction`	Animate opacity
`TimedAction`	Trigger a callback after a delay or on each loop iteration
`SpriteAnimation`	Cycle through sprite frames
`Animation`	Play a skeletal or keyframe clip
`Particles`	Drive particle playback
`TranslateTracks` / `RotateTracks` / `ScaleTracks` / `MorphTracks`	Multi-keyframe tracks with per-segment easing

All actions support easing curves. See TimedAction (p. 588) for the full list of `EaseType` values.

In the **Properties** window, **Play on start** starts an action or animation with the scene (checking it sets its **State** to Running), and **Start offset** or **Random start** keep copies out of sync. Timed actions have **Loop** and **Ping-pong**, a rotation action can **Spin** by any **Angle**, and keyframe tracks have **Loop** and **Relative**, so each of them can also run on its own, without an animation. See Runtime actions (p. 298).

Easing curves

Timed actions (`PositionAction`, `RotationAction`, ...) have a single **Ease** property in the **Properties** window (`setFunctionType()` in code). Keyframe tracks ease **per segment** through a collapsible, sparse **Easing** list. **Add Ease** appends the next explicit segment entry, up to one entry per pair of consecutive keys; expand the list to edit or remove its `Ease 0 - 1`, `Ease 1 - 2`, ... rows. Missing trailing entries are linear. Removing an entry shifts later entries to the preceding segments. GLTF-imported clips match their authored sampler mode: `LINEAR` clips leave easing unset, `STEP` clips import with every segment set to `Step`, and `CUBICSPLINE` clips interpolate from imported tangents instead — see Per-segment easing (p. 304) and Interpolation modes (p. 302). Common choices:

Curve	Best for
`LINEAR`	Constant-speed motion, debug
`STEP`	Held poses — stop-motion and stepped, pose-to-pose motion
`QUAD_IN_OUT`	UI transitions and camera moves
`BOUNCE_OUT`	Playful jumps and pop-in effects
`BACK_IN`	Anticipation before a jump
`ELASTIC_OUT`	Springing UI elements

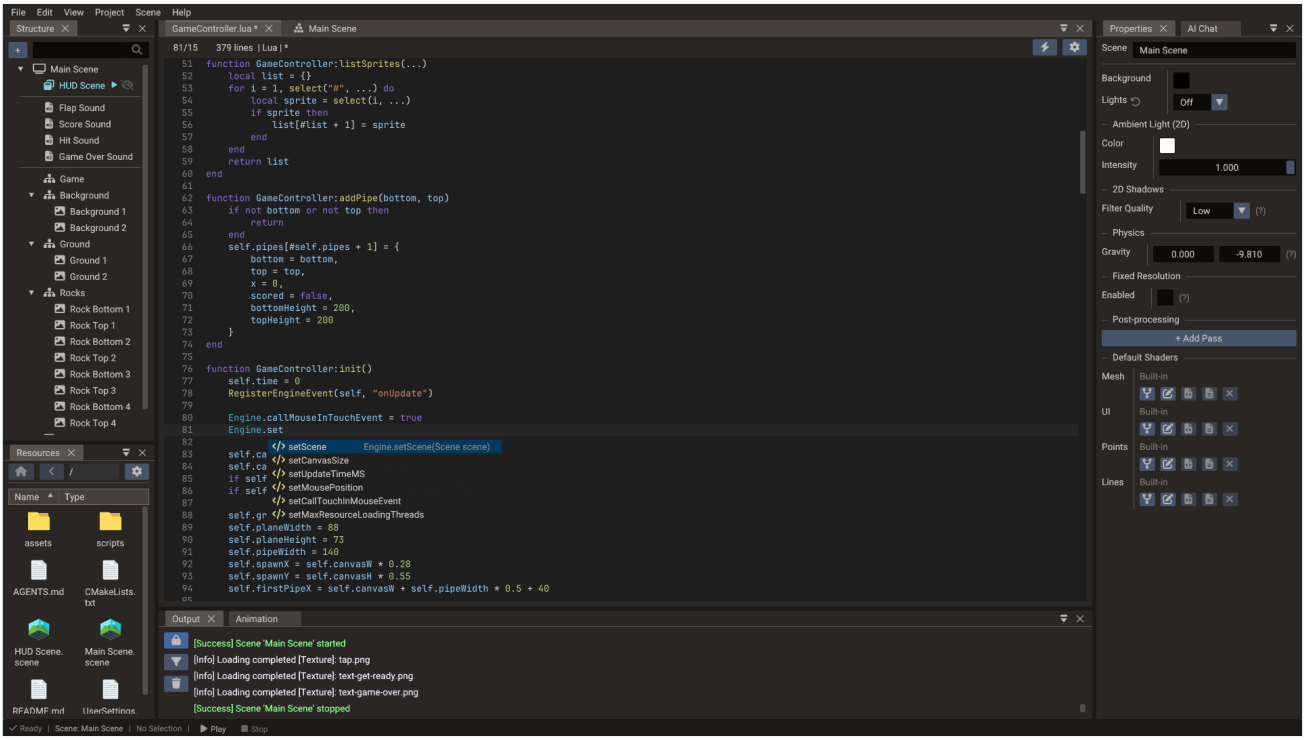
Best practices

- Name clips clearly and consistently (`idle`, `walk`, `run`, `jump`, `attack`).
- Keep looping and one-shot animations in separate clips.
- Use consistent frame intervals for sprite animations.
- Prefer skeletal animation for character movement; use keyframe tracks for camera and scene-wide effects.

- Use morph targets sparingly on performance-sensitive targets.
- Test in play mode after changing script-driven animation timing.

Code Editor

The integrated code editor lets you write and edit Lua and C++ scripts without leaving the Doriax editor. It provides syntax highlighting, API completion, script creation dialogs, and a live output panel for compile and export messages.



Supported workflows

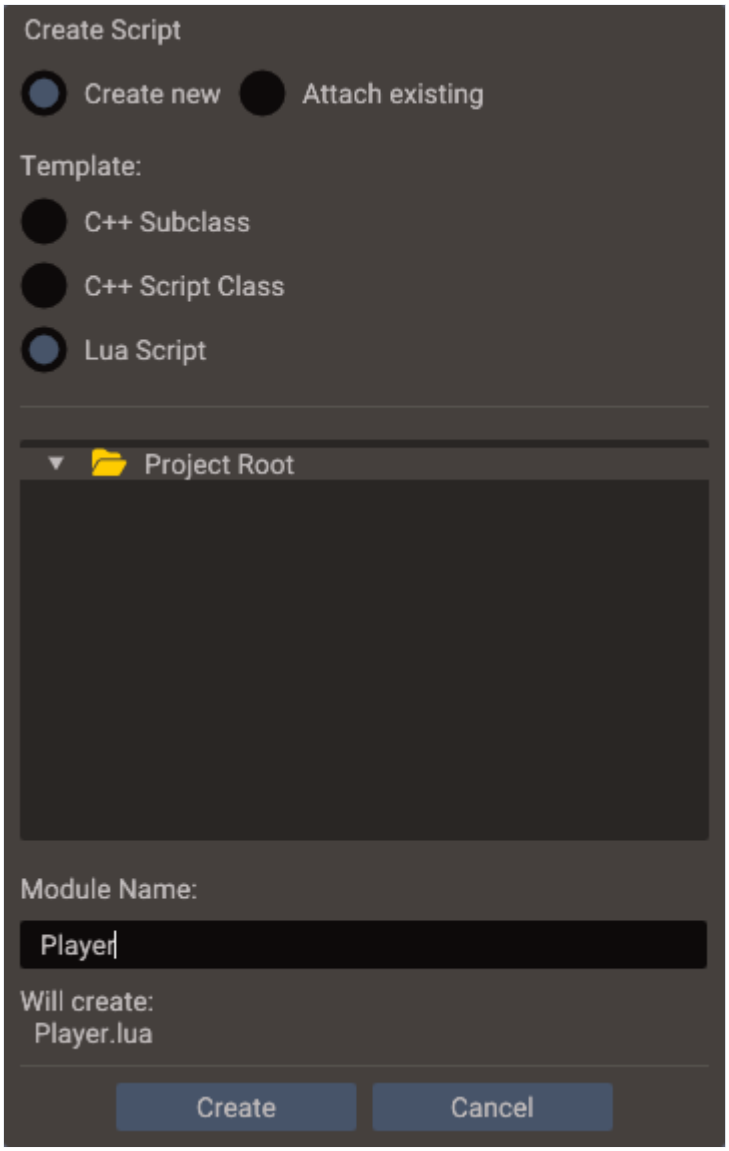
Workflow	Notes
Lua scripts	Fast iteration — script-only changes take effect immediately in play mode without rebuilding
C++ scripts	Compiled at export/build time for native performance; require a rebuild to take effect
GLSL shaders	Edit forked shader sources (`.vert` / `.frag` / `.glsl`); saving recompiles and refreshes the viewport. See Custom Shaders (p. 143)
Script templates	Create new Lua or C++ script files from boilerplate using the **New Script** dialog
Event menu	Add an engine, input, component, physics, or monetization event handler to a script in one click. See Add events
API completion	Engine API suggestions generated from Lua bindings; covers classes, methods, properties, and constants. Properties that return a copy (p. 329) are marked `(copy)`
Build output	Compiler errors, warnings, and export messages stream into the Output panel

Creating a new script

1. Select an entity and click **New Script** at the top of the **Properties** window.
2. Choose **Lua Script**, **C++ Subclass**, or **C++ Script Class**.
3. Enter the class/module name and confirm.

The editor generates the files from templates and attaches a `ScriptComponent` entry to the entity; open them here with the header and source buttons on that entry. To link *existing* files instead, choose **Attach existing**

in the same dialog, or add an empty entry with **Add Script** inside the ScriptComponent and pick the files in its edit dialog — see Creating Scripts (p. 199).



Drag entities into your code

Drag an entity from the **Structure panel** onto a script open in the Code Editor to create an entity reference property:

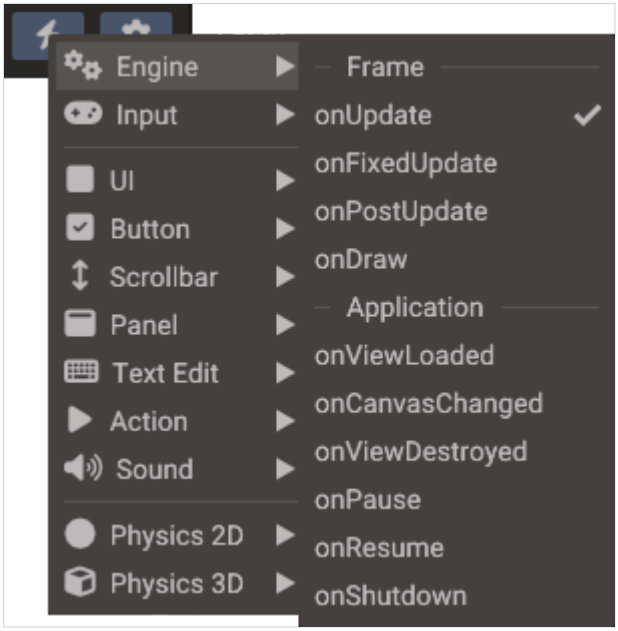
- **Lua file** — a typed entry is appended to the `properties` table.
- **C++ header** — a `DPROPERTY` line and pointer member are inserted (with the `#include` when the type is another script class). Dropping on a `.cpp` shows a reminder to use the header.

The property type comes from the dropped entity (its script class, or wrapper type such as `Object`/`Mesh`/`Camera`), and the entity is assigned as the property value automatically. See Script Properties (p. 227).

Add events

The **lightning** button (*Add event*) at the top of a Lua or C++ script window, next to the **gear**, opens the menu of events a script can subscribe to:

Menu	Events
Engine	Frame updates (`onUpdate`, `onFixedUpdate`, `onPostUpdate`, `onDraw`), application events (`onPause`, `onResume`, `onShutdown`, ...) and `onSceneLoaded`
Input	Keyboard, mouse, touch, and gamepad events
UI, **Button**, **Scrollbar**, **Panel**, **Text Edit**, **Action**, **Sound**	Events of that component, which the script's entity needs to have
Physics 2D, **Physics 3D**	Contacts, sensors, body activation, and collision filters, for every body in the scene
AdMob, **In-App Purchase**, **Web Portal**	Results of the monetization services (p. 337): ads loaded, shown and rewarded, purchases, portal ads and cloud saves. They belong to the class, so any script can add them



The **Touch** section of **Input** includes `onTouchCancel(pointer, x, y)` for releasing an interrupted finger's action without a click. A full cancellation calls the handler once per tracked finger. See Touch cancellation (p. 216) for notification order and how to handle fingers that remain down.

Each entry shows the handler parameters. Picking one writes the code the event needs and puts the caret inside the new handler:

C++

The `REGISTER_*` macro goes in the constructor and its `UNREGISTER_*` pair in the destructor. The handler is declared in the header and defined in the source file, and the event's header (such as `ButtonComponent.h` or `AdMob.h`) is included when missing. It works from either file of the script; the other one, with the same name next to it, is opened when the handler goes there. Monetization events register with `REGISTER_EVENT(AdMob::onAdLoaded, onAdLoaded)`.

```
Menu::Menu(Scene* scene, Entity entity): ScriptBase(scene, entity) {
    REGISTER_ENGINE_EVENT(onUpdate);
    REGISTER_BUTTON_EVENT(onPress, onPress);
}

Menu::~Menu() {
    UNREGISTER_BUTTON_EVENT(onPress, onPress);
}

void Menu::onPress() {

}
```

Lua

The registration goes in `init()`, which is created when missing, and the handler before `return Name`. Monetization events register with `RegisterEvent(self, AdMob.onAdLoaded, "onAdLoaded")`.

```
function Menu:init()
    RegisterEngineEvent(self, "onUpdate")
    RegisterEvent(self, Button(self.scene, self.entity):getButtonComponent().onPress, "onPress")
end

function Menu:onPress()

end
```

An event the script already registers shows a check mark, and picking it moves the caret to its handler. Engine and input handlers are named like the event, as `REGISTER_ENGINE_EVENT` and `RegisterEngineEvent` require. Other handlers are named after the event too (`onPress`, `onBeginContact2D`), with the component added when that name is taken: **Sound > onPause** becomes `onSoundPause`, since `onPause` belongs to the engine event. Monetization events add their class instead: the second `onInitialized` a script handles becomes `onAdMobInitialized`, `onInAppPurchaseInitialized` or `onWebPortalInitialized`. Collision filters (`preSolve2D`, `shouldCollide2D`, `shouldCollide3D`) start with `return true`, so every contact is kept until you add a condition.

Each change is one undo step in the files open in the Code Editor. A closed header that only needs the declaration is saved directly.

Changed lines

When the project is kept in Git with Version Control Files (p. 66) on, the gutter marks every line that differs from the staged version of the file, as VS Code does:

Mark	Meaning
Green bar	Added lines
Striped blue bar	Changed lines
Red triangle	Lines were removed here

The same colors mark the changes on the left of the scrollbar, to find them in a long file. Rest the mouse on a mark to see the change: the removed lines in red, the lines that replaced them in green, and the characters that changed highlighted. The arrow at the top of the popup reverts the change to the staged lines, as one undo step.

The marks follow your typing and compare with what is staged, so staging a change clears its marks. Untracked, ignored and conflicted files have none, and neither has a file opened through a symbolic link.

Script entry point

Scripts subscribe to the engine events they need (`onUpdate`, `onFixedUpdate`, `onPause`, `onResume`, `onShutdown`, and the input events). Lua scripts register in `init()`; C++ scripts register in the constructor and unregister in the destructor:

Lua

```lua
local Player = {}

function Player:init()
    RegisterEngineEvent(self, "onUpdate")
end

function Player:onUpdate()
    -- called every frame
end

return Player
```

C++

```cpp
#include "ScriptBase.h"

class Player : public doriax::ScriptBase {
public:
    Player(doriax::Scene* scene, doriax::Entity entity);
    ~Player();

    void onUpdate();
};
```

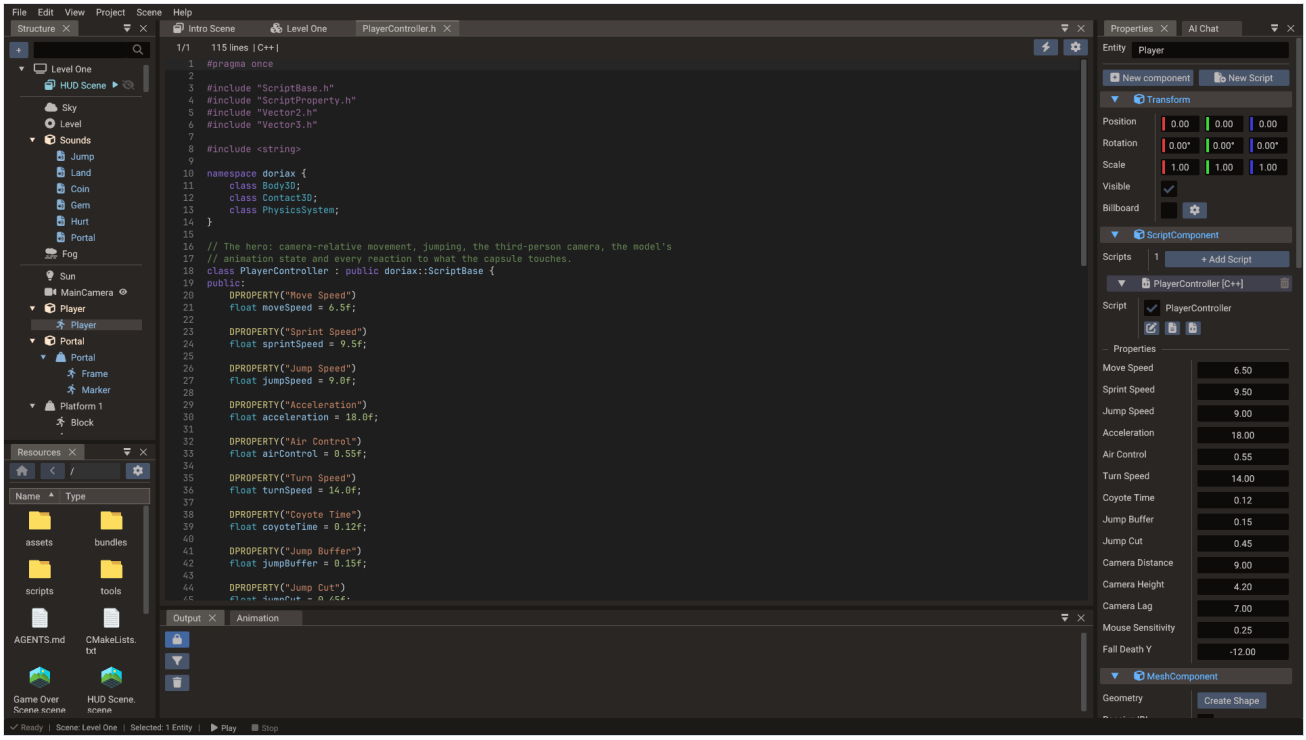
```cpp
Player::Player(Scene* scene, Entity entity) : ScriptBase(scene, entity) {
    REGISTER_ENGINE_EVENT(onUpdate);
}

Player::~Player() {
    UNREGISTER_ENGINE_EVENT(onUpdate);
}
```

The event menu writes this code for any event. See Creating Scripts (p. 199) for the full lifecycle and event system documentation.

DPROPERTY and script properties

Properties declared with `DPROPERTY` (C++) or in the `properties` table (Lua) appear as editable fields in the **Properties window** automatically.




```
DPROPERTY("Speed")
float speed = 5.0f;

DPROPERTY("Jump Force")
float jumpForce = 12.0f;
```

The editor parser reads `DPROPERTY` from C++ headers and populates the Properties window without requiring a build. See Script Properties (p. 223) for syntax and type mapping details.

Project entry points

A project can have both Lua and C++ startup paths. Set which one is active in project settings, or keep both and use `NO_CPP_INIT` / `NO_LUA_INIT` to disable one at compile time.

C++ startup

```
#include "Doriax.h"
using namespace doriax;

Scene scene;

DORIAX_INIT void init() {
    Engine::setScene(&scene);
}
```

Lua startup

```
scene = Scene()
Engine.setScene(scene)
```

Practical tips

- Keep startup code small — move gameplay behavior into scripts or systems.
- Use `DPROPERTY` for values that designers should tune; avoid hard-coding magic numbers inside script logic.
- Watch the Output panel after every C++ edit or export; it shows the first compiler error clearly.

- Lua scripts iterate faster during development — prototype in Lua, port to C++ for performance-critical paths.
- Use the built-in Log (p. 387) API for runtime diagnostics rather than relying on `print`.

Custom Shaders

Doriax ships built-in shaders for each renderable type (Mesh, UI, Points, Lines, Sky, Water). The editor lets you **fork** any of them into your project, edit the GLSL in the Code Editor (p. 136), and see the result in the viewport — without leaving the editor. Forked shaders are compiled and shipped with your project just like the built-in ones.

The same fork mechanism drives post-process passes — fullscreen shaders a scene runs over the finished image.

How customization works

A custom shader is a **fork** of a built-in shader. The editor copies the built-in entry-point files (for example `mesh.vert` and `mesh.frag`) into a folder you pick in your project; you edit the shading code while the rest of the rendering contract stays intact:

- The **variant system** is unchanged — the engine still injects the same feature defines (skinning, normal maps, shadows, IBL, fog, instancing, ...) based on the mesh, so your edited shader keeps working across all of those cases.
- Skinned forks that `#include "includes/skinning.glsl"` bind bone matrices through a storage buffer, or through a bone texture when the editor compiles for OpenGL / OpenGL ES (`SKINNING_TEXTURE`). Re-fork that include if you copied an older `skinning.glsl` that still declared `mat4 bonesMatrix[MAX_BONES]` in a uniform block.
- The **shadow, depth, and G-buffer passes** keep using the built-in shaders, so shadow casting and screen-space effects continue to work. A Mesh can fork its **depth shader** as well — the one behind its shadow maps and the depth pre-pass — for a colour fork that moves vertices or discards fragments (see The Depth Shader row). To only move vertices, a vertex offset is simpler: one function that every pass applies.
- `#include` directives still resolve against the engine's shader library, so you only need the top-level `.vert` / `.frag` — not the whole include tree. If you want to edit the includes as well, the fork dialog can copy them for you (see Includes).

This means you customize the *surface/shading logic* while the engine continues to drive lighting, passes, and built-in inputs. Values of your own — a wave speed, a dissolve amount, the current time — reach the fork through its shader parameters.

The Shader row

Open the **Properties** window for a Mesh, UI, Points, Lines, Sky, or Water entity. Each of these components has a **Shader** row:

Control	Action
Built-in label	Shown when the component uses the default engine shader.
Fork	Opens the Fork Shader dialog to choose where the fork is created and what it is called. On confirm it writes the files, assigns them to the component, and opens them in the Code Editor.
Edit shader files	Picks the `.vert` and `.frag` independently, so entry points that do not share a base name can be assigned. Changes apply as soon as both slots are set (or both cleared).
Open vertex / **Open fragment**	Re-opens the assigned file in the Code Editor.
Reset to Built-in	Clears the custom shader and returns to the engine default.
Drag-and-drop	Drop an existing `.vert` / `.frag` from the Resources Browser (p. 106) onto the row to assign it.



Fork is available only while the row shows **Built-in** — reset the shader first to fork it again, so an existing fork is never orphaned by accident.

The Shader row is shown for a single selected entity (the path is per-entity). Assigning, editing, or resetting a custom shader is undoable.

The Depth Shader row

A Mesh component has a second row, **Depth Shader**, with the same controls. It forks `depth.vert` and `depth.frag` — the shader the engine draws the mesh with into **shadow maps**, into the optional scene depth prepass (p. 276), and into the **depth pre-pass** that feeds SSAO and post-process depth while SSR is off (with SSR on, those read the G-buffer, which keeps the built-in shader).

The colour fork never reaches those passes, so a mesh whose fork displaces vertices or discards fragments keeps casting the shadow of its undisplaced, undiscarded geometry — and self-shadows against it once the displacement grows. Fork the depth shader too, apply the same displacement or `discard` there, and the shadow follows. The variant defines (skinning, morph targets, terrain, instancing, alpha mask) are the ones `depth.vert` and `depth.frag` already handle, so the fork keeps working across those cases.

A depth fork reads the same shader parameters as the colour fork, so one value drives both: declare the same `#pragma param` names in it. It is compiled as soon as it is set — even in a scene with no shadow-casting light — so build errors show up right away; a fork that fails to build is reported as **Depth shader failed to build** under the row while the built-in depth shader keeps drawing.

The depth shader is per mesh — there is no scene default — and its fork name is pre-filled with a `_depth` suffix. Like the colour fork it is undoable, saved with the scene, exported with the project, and scriptable through `customDepthShader` (see Setting values from scripts).

> **Resolution in a depth fork**
>
> `Engine.resolution` is the size of the target being written: the shadow slot of a cascade or cube face, or the depth pre-pass target.

Depth forks and the depth prepass

With the scene depth prepass (p. 276) on, the depth fork also writes the depth the colour pass tests against, so both must land on the same pixel. `u_vs_depthParams` carries `mvpMatrix` — the light (or camera) view-projection already multiplied by the model matrix — and `depth.vert` sets `gl_Position` from it under `invariant`, matching `mesh.vert` bit for bit. Keep that in a fork: multiplying `lightVPMatrix * modelMatrix` yourself still compiles (older forks keep working) but can round differently and z-fight in the prepass.

A mesh whose **colour** shader is forked — or covered by a scene default mesh shader — without a depth fork is left out of the prepass and drawn once in the colour pass, since the built-in depth shader cannot know where the fork moves its vertices. Add a depth fork with the same displacement to bring it back in.

The Vertex Offset and Coverage rows

A Mesh component also has a **Vertex Offset** row and a **Coverage** row, for meshes that only need their vertices moved or part of their surface removed, without forking any shader. Each takes one `.glsl` file with a single function:

```
vec3 getVertexOffset(vec3 localPosition, vec3 worldPosition);
float getCoverage(CoverageContext context);
```

The engine runs them in the color, depth and G-buffer passes, so shadows, SSAO and screen-space reflections follow without a depth fork, and the mesh keeps the built-in shading. Their values are declared with `#pragma` lines, such as `#pragma param float waveHeight = 0.2`, and edited in the effect's own **Parameters** section under the table. See Shader Effects (p. 282) for the functions, the parameters and the engine's effects.

Control	Action
None label	Shown while the mesh has no effect of that kind.
New	Writes a starting file to `shaders/<entity>_offset.glsl` or `shaders/<entity>_coverage.glsl`, assigns it, and opens it in the Code Editor.
Engine effects	Assigns one of the engine's own, such as `offsets/wind.glsl` or `coverage/see_through.glsl`, shown as *built-in*.
Pick	Assigns an existing `.glsl` from the project.
Copy	Copies the assigned built-in effect into the project, assigns the copy and opens it, to change it.
Open	Re-opens the assigned file.
Remove	Clears the effect.
Drag-and-drop	Drop a `.glsl` from the Resources Browser (p. 106) onto the row.

Max Offset, shown under a vertex offset, is the longest offset in world units: longer ones are clamped, and culling widens the mesh bounds by it. **Cut Shadows**, shown under a coverage function, makes the shadow maps follow the coverage too.

An effect file finds its includes like a fork, starting in its own folder. Those files stay its own: a fork on the same mesh with a file of the same name does not replace them. Each combination of fork and effect files builds its own shaders, compiled and exported with the project like a fork. Terrain ignores both rows. Scripts set them through `vertexOffsetShader`, `vertexOffsetMax`, `coverageShader` and `coverageShadows`.

Water shaders

A Water (p. 496) component has two rows. **Shader** forks `water.vert` and `water.frag`, the surface itself. **Underwater Shader** forks `fullscreen.vert` and `underwater.frag`, the fullscreen fade drawn over the scene while the main camera is below the surface (it needs **Underwater** on). Its fork name is pre-filled with an `_underwater` suffix, and a fork that fails to build is reported as **Underwater shader failed to build** under the row while the built-in fade keeps drawing.

Both forks read the water's shader parameters by name, so one value can drive both. The surface shader can also be a scene default; the underwater shader is per water only.

The water fork keeps the engine's variant defines: `HAS_FOG`, `USE_SCENE_DEPTH` (Depth Effects), `USE_PLANAR_REFLECTION`, `USE_SHADOWS`, and `USE_REFRACTION` — see Shader Variants — water (p. 280).

> **Keep the engine blocks and the wave math**
>
> The engine fills `u_vs_waterParams`, `u_fs_waterParams`, and `u_fs_underwaterParams` by layout, not by name, so keep them as the fork copied them. New engine versions only append members at the end of `u_fs_waterParams`: a fork made on an older version keeps working without the newer effects, and logs a warning to fork it again.
>
> Water.getHeight (p. 502), body buoyancy (p. 319), and the sorting of transparent objects around the water use the built-in wave sum. If a fork displaces the surface differently in `water.vert`, they no longer match the waves it draws.

The Fork Shader dialog

Forking asks where the new shader goes before writing anything:

Field	Meaning
Destination folder	A tree of the folders in your project. Defaults to `shaders` when it exists, otherwise the project root.
Shader Name	The file name stem, without an extension. Letters, digits, and underscores only; other characters become `_` as you type.
Also fork shader includes	Copies the engine `.glsl` files this shader actually uses so you can edit them too. See Includes.
Will create	The exact files that will be written, project-relative.

The name is pre-filled from the entity (or the scene and shader type, for a scene default) and gets a numeric suffix if that name is already taken. **Create** stays disabled while a target already exists — forks never overwrite files, because undoing one deletes what it wrote.

The layout depends on whether you fork the includes:

Option	Files created
Off	`<folder>/<name>.vert` and `<folder>/<name>.frag`
On	`<folder>/<name>/` containing `<name>.vert`, `<name>.frag`, and an `includes/` copy

Forking includes gives the shader a folder of its own so your edited copies apply to that shader alone. Creating the fork is a single undoable step — undo removes the files it wrote and restores the previous shader.

Scene default shaders

Besides per-component shaders, each scene can set a **default custom shader per type**. Select the scene (no entity) in the **Structure panel** and look for the **Default Shaders** section at the bottom of the scene settings in the Properties window. It shows one row per applicable type — 3D scenes list Mesh, Sky, Water, UI, Points, and Lines; 2D scenes omit Sky and Water; UI scenes list only UI.

Each row offers the same controls as the component Shader row (fork, edit files, open `.vert` / `.frag`, reset, drag-and-drop), and every change is undoable.

The shader used by a component is resolved with this priority:

1. **Component shader** — a custom shader assigned on the component always wins.
2. **Scene default shader** — used by every component of that type with no shader of its own. Its Shader row shows the default's name followed by *(scene default)*.
3. **Engine built-in** — used when neither is set.

A component with no custom shader *inherits* the scene default; resetting a component's shader returns it to the scene default (or the built-in if the scene has none). To keep a single component on stock shading while a scene default is active, fork the built-in shader for that component and leave it unedited.

A default that declares shader parameters lists them under its row, in a section named after the fork (for example **toon Parameters**). Values set there belong to the scene, and every component that inherits the shader reads them. A component can still set its own value for a parameter, and that value wins (see Editing values).

Scene default shaders and their values are saved with the scene, applied in play mode, and exported like any other custom shader — components that inherit them compile and ship the right shader variants automatically. They are also scriptable via the `Scene` properties `defaultMeshShader`, `defaultUIShader`, `defaultSkyShader`, `defaultPointsShader`, `defaultLinesShader`, and `defaultWaterShader` (see the Scene

reference (p. 390)), and the values through `setDefaultShaderUniform` (p. 405), which takes the shader type (`ShaderType.MESH`, `ShaderType.UI`, and so on).

Shader parameters

A fork declares its own values with `#pragma` lines, and the engine fills them from fields in the Properties window, from scripts, and from values the scene and its cameras share. This is how a water, wave, fire, or dissolve effect gets its parameters.

Write one `#pragma param` line per value, in the vertex file, the fragment file, or an include, and read it as `params.name`. Every built-in shader calls `loadShaderParameters()` at the top of `main()`, which fills `params`, so a fork can read them anywhere after that:

```
// wave.vert
#pragma param float amplitude = 0.2 [0, 1] "Height of the waves"
#pragma param float speed = 3 [0, 10]

// ... in main():
pos.y += sin(Engine.time * params.speed + pos.x * 4.0) * params.amplitude;
```

Keep that call in a fork: without it the values read zero, and the build warns about it.

> **Vertex displacement is colour-pass only**
>
> A fork only replaces the colour pass, so a vertex moved like this swims while its shadow stays where the mesh data is. Fork the depth shader as well and displace there too, and the shadow (and SSAO, while SSR is off) follows. Screen-space reflections read the built-in G-buffer and keep the undisplaced surface. A vertex offset moves every pass at once.

```
// wave.frag
#pragma param vec4 tint = 0.2, 0.5, 1, 1 color

// ... at the end of main():
g_finalColor.rgb = mix(g_finalColor.rgb, params.tint.rgb, 0.5 + 0.5 * sin(Engine.time));
```

A name declared in both files is one value. The types are `float`, `vec2`, `vec3`, `vec4`, `int`, and `ivec2` to `ivec4`, and a declaration can give a default, a range, a color picker, and a tooltip. `#pragma scene` and `#pragma view` lines read values that the whole scene or one camera shares, and `Engine.time` and `Engine.resolution` need no declaration. The syntax and where each value comes from are in Shader Effects — Declaring parameters (p. 282).

Every shader type takes them: Mesh, depth, UI, Points, Lines, Sky, Water, underwater, and post-process forks. The shader builder packs the values of each stage into one uniform block, so a fork declares no uniform block of its own: the engine never fills a block it does not know. Changing a value never rebuilds the shader.

Forks from v0.8.4 and earlier

Older forks declared `u_vs_customParams`, `u_fs_customParams`, or `u_fs_postParams` blocks. The engine no longer fills them, and a fork that still declares one fails to build with an error naming the block.

When a project with such files opens, the editor lists them and asks whether to rewrite them; answering No leaves them alone, and it asks again the next time. AI agents have `port_shader_fork` for one fork. The command line never rewrites them. The port changes each file in place: every block member becomes a `#pragma param` line read as `params.name`, with the comment after the member as its tooltip, `time` becomes `Engine.time` (it now follows the scene's clock), `resolution` becomes `Engine.resolution`, and the parameter include and the `loadShaderParameters()` call are added. The values saved by name on the component or the pass are kept. A file whose block has a member that is not a `float`, a `vec2` to `vec4`, an `int` or an `ivec2` to `ivec4` is left as it was, with the reason in the Output window.

The port keeps the rest of the fork as it was, so a ported Mesh fork still lacks the hooks of vertex offsets and coverage. To get them, fork the current built-in again and copy your changes over. The port will be removed in a later version, so port old forks while it is there.

Editing values

Each shader file of the component gets a **Parameters** section below the Shaders table, named after the file (for example **wave Parameters**), with one field per `#pragma param`. Saving the file updates the fields. Values are stored **on the component**, by name:

- A Mesh keeps one set of values for all of its submeshes, and its depth fork and effect files read the same values.
- A component that inherits a scene default shader shows the scene's value for any parameter it has not set. Editing the field gives the component its own value, which wins, so two meshes on the same default shader can still be tinted differently. A field with its own value has a reset arrow that goes back to the scene's.
- A parameter nobody set reads its declared default. Renaming or deleting a parameter leaves its old value unused.

The fields follow the declaration: a range limits the drag, `color` shows a color picker, `int` types take whole numbers, and the description is the tooltip. Every edit is undoable, and the values are saved with the scene and exported with it. The section also lists the scene and view parameters the file reads, which are edited on the scene and on each camera.

glTF models

A multi-node glTF `Model` renders through its part entities (one Mesh component each); the root's own Mesh is empty unless the model is merged or single-node. Fork and edit the shader, and set the values, on the parts — like colour and material — not on the Model root.

`Engine.resolution` is what screen-space effects need (scanlines, pixel snapping, dithering against `gl_FragCoord`), and it follows the target: the same fork drawn by a render-to-texture camera sees that camera's size. See Engine values (p. 283) for both engine values.

Setting values from scripts

The same values are scriptable, so a parameter can follow gameplay or animate every frame. `Mesh` (and everything derived from it — `Shape`, `Model`, `Sprite`, `Terrain`, `Tilemap`), `Image`, `Text`, `Polygon`, `Points`, `Lines`, `SkyBox`, and `Water` all expose:

Method	Meaning
`setShaderUniform(name, value)`	Sets the value of a parameter. `value` is a `float`, `Vector2`, `Vector3`, or `Vector4`; the components a narrower value leaves out read zero.

Method	Meaning
`getShaderUniform(name)`	The component's own value as a `Vector4`, zero when unset. Declared defaults and values inherited from a scene default are not there.
`removeShaderUniform(name)`	Drops the value, so the parameter reads the inherited or declared default again.
`customShader`	The fork base path, so a script can assign or reset the fork too.
`customDepthShader`	A Mesh's depth fork base path; it reads the same values.
`vertexOffsetShader`	A Mesh's vertex offset file; it reads the same values.
`vertexOffsetMax`	The longest offset in world units, which also widens the culling bounds; 0 = no limit.
`customUnderwaterShader`	A Water's underwater fork base path; it reads the same values.

Setting a value only rewrites the uploaded block, with no shader reload, so calling it every frame is fine:

C++

```
Shape water(&scene);
water.createPlane(20, 20);
water.setCustomShader("shaders/wave");
water.setShaderUniform("amplitude", 0.3f);
water.setShaderUniform("tint", Vector4(0.2f, 0.5f, 1.0f, 0.5f));
```

Lua

```
function WaveScript:init()
    self.water = Shape(self.scene)
    self.water:createPlane(20, 20)
    self.water.customShader = "shaders/wave"
    self.water:setShaderUniform("amplitude", 0.3)
    self.water:setShaderUniform("tint", Vector4(0.2, 0.5, 1.0, 0.5))
    RegisterEngineEvent(self, "onUpdate")
end

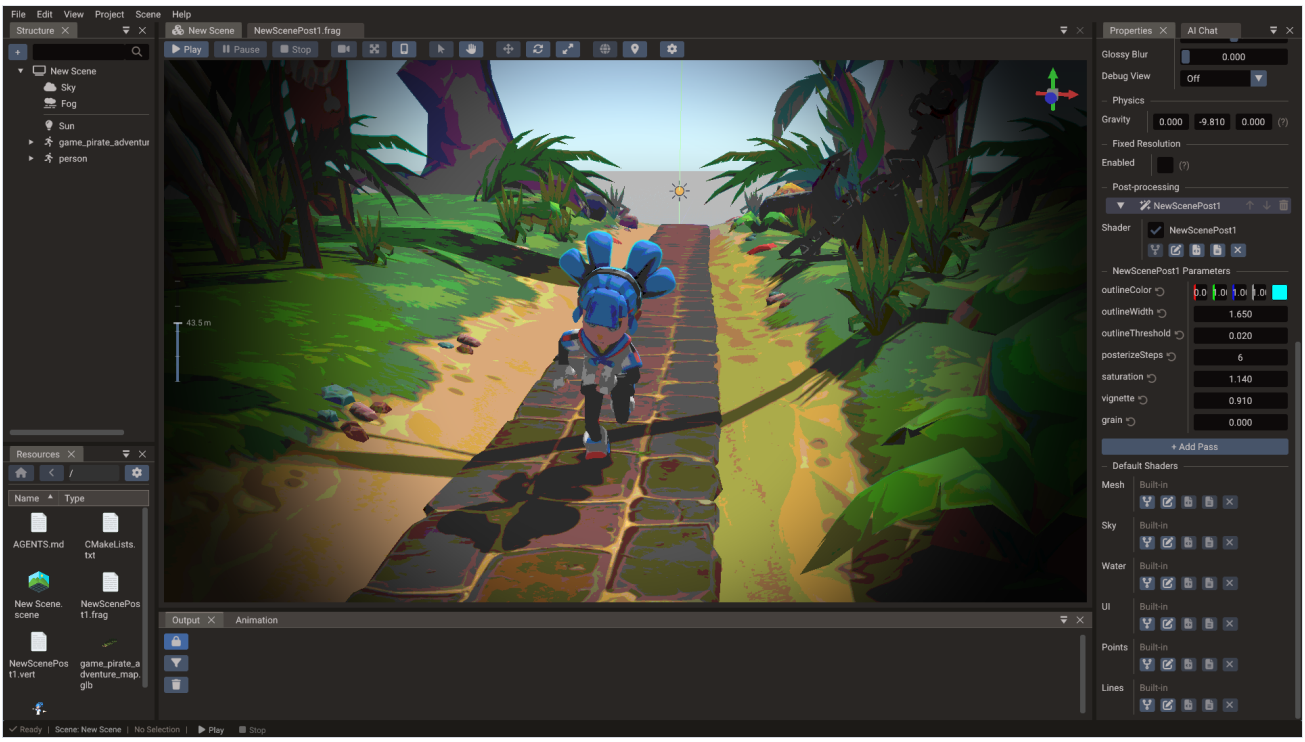
function WaveScript:onUpdate()
    -- Engine.time needs no call: the engine writes it. Gameplay-driven values
    -- go through the same method every frame.
    self.water:setShaderUniform("amplitude", 0.3 * self.stormLevel)
end
```

For a glTF model, call these on the part entities (`Mesh(scene, scene:findEntity("Body"))`), as the note above explains. See the Mesh reference (p. 445) for the full signatures.

> **The G-buffer pass**
>
> A Mesh's values reach its colour fork, its depth fork, and its effect files. The SSR G-buffer pass keeps the built-in shader and runs only the effect files, so a displacement written in a fork never moves the object's screen-space reflection, while a vertex offset does.

Post-process passes

A scene can run an ordered chain of **fullscreen passes** over its finished image: sharpen, tint, vignette, colour grading, and so on. Each pass is a forked shader, and unlike the component shaders above the chain belongs to the scene, not to any entity.

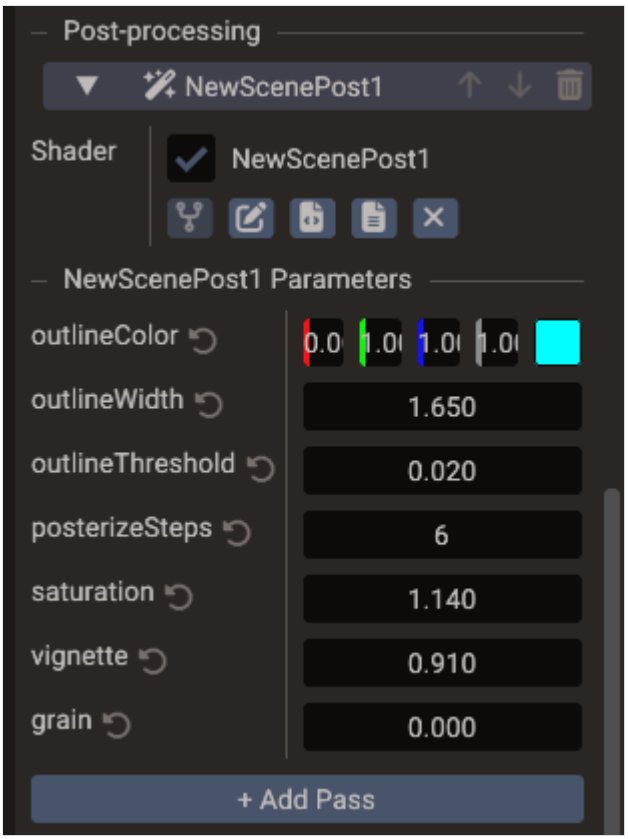


Select the scene (no entity) in the **Structure panel** and open the **Post-processing** section of the scene settings in the Properties window.

Control	Action
Add Pass	Appends a pass. A new pass starts on the built-in passthrough, which copies the image unchanged.
Pass header	Names the pass after its shader, followed by its place in the chain when the panel is wide enough (**NewScenePost1 [Pass 1]**). Click it to fold the pass. It is dimmed while the pass is disabled.
↑ / ↓	On the header. Moves the pass in the chain. Passes run top to bottom.
🗑	On the header. Removes the pass. Right-clicking the header offers the same actions.
Checkbox	Before the shader name in the **Shader** row. Enables or disables the pass without removing it.
Shader row	The same controls as any other shader row — fork, pick files, open, reset, drag-and-drop.

Every change is undoable, and the chain is saved with the scene.

An agent connected through the MCP server (p. 172), or the AI Chat, edits the same chain: `list_post_process_passes` reads it, `set_post_process_pass` adds, removes, moves, enables, or repoints a pass, and `fork_shader` with `shader_type` set to `postprocess` adds a pass with a new fork of the passthrough.



The chain runs on the scene's **active camera**, after the opaque, transparent, UI, and SSR passes, and before the fixed-resolution upscale. Secondary cameras — minimaps, mirrors, portals — do not run it; if the active camera itself renders to a texture, the chain writes into that texture. A scene drawn as a layer skips its chain, because a fullscreen pass would overwrite the scene below it.

Writing a pass

Forking a post-process pass copies `fullscreen.vert` and `postprocess.frag` into your project. The vertex shader draws a fullscreen triangle and normally needs no edits — all the work happens in the fragment shader.

A pass may declare any subset of the inputs below; only the ones it actually uses are bound:

Sampler	Contents
`u_sceneColorTexture`	The previous pass's output, or the scene colour for the first pass.
`u_depthTexture`	Camera-space packed depth. The engine runs a depth pre-pass for the chain when SSR and SSAO are both off, so this is always available.
`u_gbufferTexture`	View-space normal, roughness, and metallic. Only produced while **SSR** is enabled; reads black otherwise.
`u_ssaoTexture`	Blurred ambient occlusion. Reads white while SSAO is off.

> **Depth orientation**
> Depth and the G-buffer keep the logical orientation of the depth pre-pass, so flip `v_texcoord.y` when sampling them on the OpenGL backends. The built-in `ssr.frag` shows the pattern.

Pass parameters

A pass declares its values like any fork, with `#pragma param` lines, and each one becomes a field of the pass in a section named after the fork:

```
#pragma param float amount = 0.5 [0, 1]
#pragma param float threshold = 0.8 [0, 1]
#pragma param vec4 tintColor = 1, 0.9, 0.8, 1 color

// ... in main(), one texel of the pass target:
vec2 texel = Engine.resolution.zw;
```

The values are stored by name on the pass. `Engine.resolution` holds the size of the pass target and its inverse: use it for anything that depends on target size, since a texel offset written as a literal breaks the moment the window resizes. `#pragma scene` and `#pragma view` values work in a pass too, so it can follow the same `Weather.wetness` as the scene's materials, and the pass's fullscreen vertex shader can declare parameters as well.

The values are scriptable as well. `scene:setPostProcessUniform(index, name, value)` rewrites one value of the pass at `index` (top of the list is 0, also from Lua) without rebuilding the chain, so it can run every frame; `scene:getPostProcessUniform(index, name)` reads it back, `scene:removePostProcessUniform(index, name)` drops it, and `scene:setPostProcessPassEnabled(index, enabled)` toggles a pass. The whole chain is available as `scene.postProcessPasses` — a list of `PostProcessPass` values with `shader`, `enabled`, and `setUniform`/`getUniform` — for scripts that build it themselves (in Lua a 1-based sequence, so `scene.postProcessPasses[1]` is pass index 0):

C++

```
scene.setPostProcessUniform(0, "amount", 0.8f);
```

Lua

```
scene:setPostProcessUniform(0, "amount", 0.8)
```

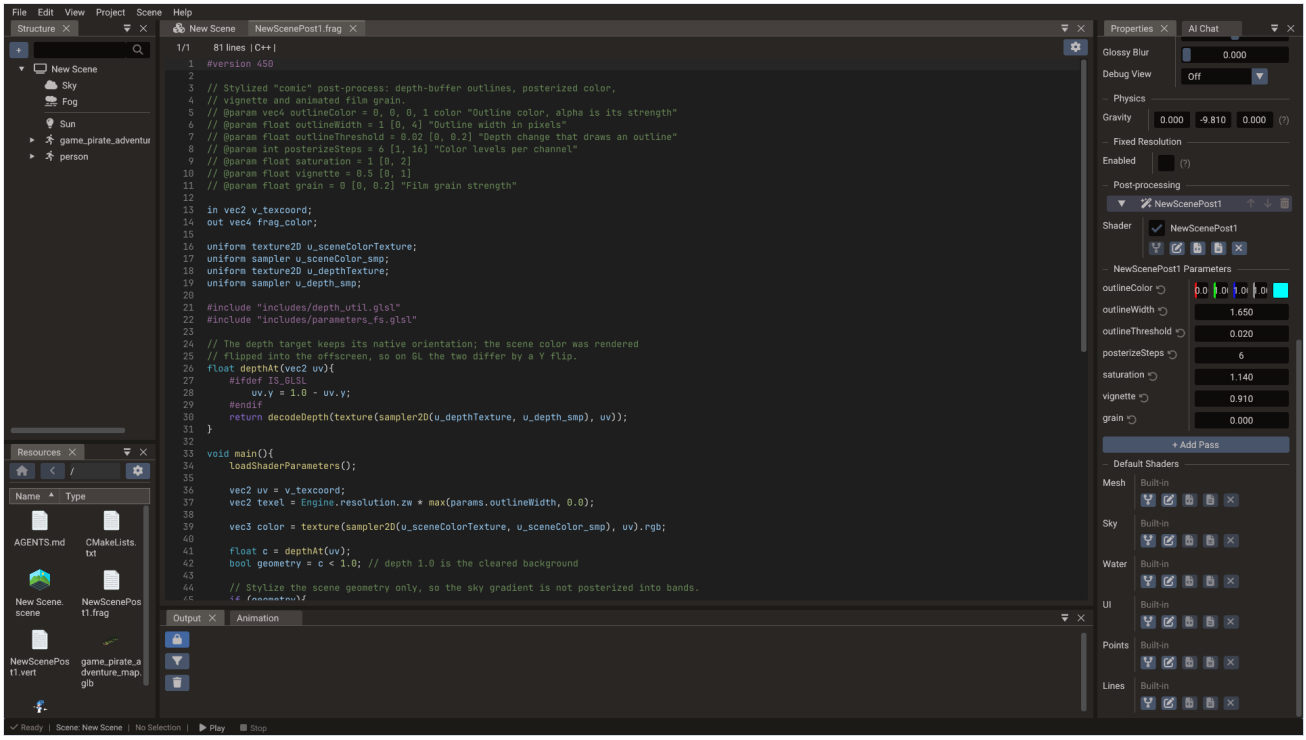
See the Scene reference (p. 405).

Export

Every pass ships its forked shader, including disabled ones — the generated scene carries the whole chain, and the enable flag is a runtime value. A pass whose shader fails to compile falls back to the passthrough, so the scene keeps rendering; the list marks it **Shader failed to build** and the compiler output goes to the Output panel.

Editing and live updates

Forked shaders open in the Code Editor (p. 136) with GLSL syntax highlighting (`.vert`, `.frag`, and `.glsl` files). When you **save** a shader source file, the editor recompiles the affected forked shaders and refreshes the viewport — no rebuild or replay needed.



Only shaders that actually reach the edited file through `#include` are rebuilt: the editor tracks each fork's real dependency graph rather than assuming everything in a folder is related. Editing a shader in an external editor works too — the change is picked up when the Code Editor reloads the file.

> **Iterate fast**
>
> Keep the file open, tweak, **Ctrl+S**, and watch the viewport update.

Vertex attribute limit

A vertex stage can declare at most **16 inputs**, and each one has to sit at a `layout(location = N)` below 16 — the location indexes the engine's attribute semantic table, which is what the HLSL and Metal backends bind against. A shader that goes past either limit fails to compile with the offending input named in the Output panel, and the component keeps the shader it had. Reuse a free location or pack values into a wider vector instead of adding another input.

Fragment inputs and outputs are not bound by that table, so varyings between stages are free of the limit.

Includes

Shaders include by a fixed key — `#include "includes/pbr.glsl"` — and a fork resolves each key in this order, taking the first that exists:

Order	Location	Scope
1	The fork's **own folder**	That fork only
2	The **project root**	Every fork in the project
3	The **engine shader library**	The default for everything

A fork's folder mirrors the engine shader library, so a key keeps its shape: `includes/pbr.glsl` is looked up at `<fork folder>/includes/pbr.glsl`.

- **Use an engine include** — `#include "includes/pbr.glsl"` resolves to the engine's copy automatically; you do not need to copy anything.

- **Override an include for one shader** — tick **Also fork shader includes** when you fork. The copies land in the fork's own folder and win for that shader alone.
- **Override an include for the whole project** — put the file at the same key under the project root, for example `<project>/includes/pbr.glsl`.
- **Add your own include** — drop a `.glsl` next to the fork and include it by its path relative to the fork folder (`#include "lib/noise.glsl"`), or put it anywhere in the project and include it relative to the project root (`#include "shaders/lib/noise.glsl"`).

Overrides never change the engine's built-in rendering — a built-in shader always uses the engine library. A fork stored loose in a folder shares whatever `includes/` that folder holds, which is why forking includes moves the shader into a folder of its own.

> **Moving a fork**
> Nothing about the fork is recorded anywhere but its file paths, so you can move, rename, or copy it freely — including by hand — and its includes follow. Moving a fork into a folder with a different `includes/` changes what it resolves to, which is what makes overrides opt-in per location.

Shader locations

Shaders have no directory settings. Sources live wherever you fork them — each fork chooses its own location in the Fork Shader dialog — and the engine never reads them anyway. It only consumes compiled `.sdat`.

Compiled `.sdat` output goes to the project's `shaders` folder, the fixed location the engine reads through `System::getShaderPath()`.

Export and runtime

Forked shaders flow through the same export pipeline as the built-in ones (see Export Window — Shader compilation (p. 164)):

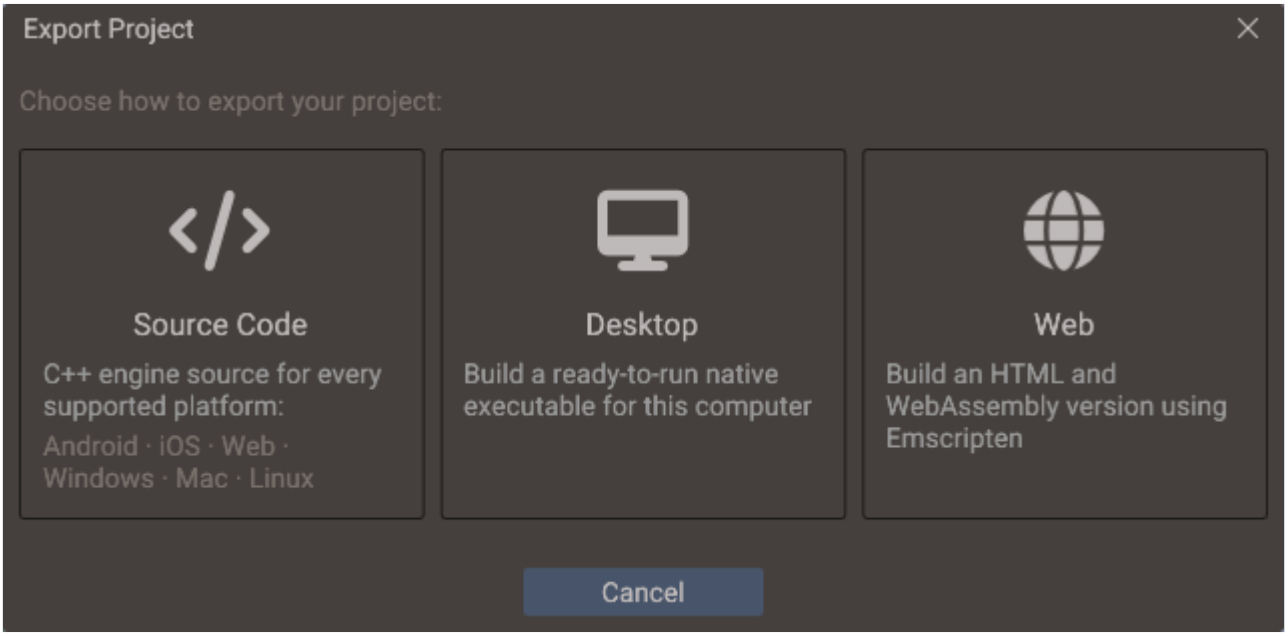
- Every used shader — including your forks — is compiled and embedded into the build for the target backends. The shader *sources* are not needed at runtime.
- To get loose `.sdat` or JSON files instead, use the `shaders` CLI command (p. 170), which writes them straight into its `--out` directory.

At runtime the engine identifies a component's shader (or a post-process pass's shader) from the value saved on it and loads the matching compiled `.sdat` — it never reads the source files. Because the engine only cares about the compiled output, your shader sources can be organized however you like.

> **Compiled vs. source location**
> The standalone runtime loads compiled shaders from the assets' `shaders` folder, and never reads shader sources. An export embeds them into the build instead, so it reads no shader directory at runtime.

Export Window

The **Export Window** turns a Doriax project into something you can ship. It collects generated scene and bundle C++, resources, scripts, engine runtime files, and compiled shaders — and, depending on the mode, also compiles the result into a ready-to-run build.



From editor data to runtime code

The engine runtime is independent of the editor. Export generates C++ that uses the public runtime API; the resulting game builds and runs without the editor or its internal project model.

Project input	Exported form
`.scene` files	C++ factory functions that create entities and configure their components
`.bundle` files included in the build	C++ factory functions for reusable entity hierarchies
User C++ scripts	Source files copied unchanged and compiled alongside generated code, with separate generated script bindings
Lua scripts	Runtime resources loaded by the engine's Lua binding
Textures, models, audio, and other assets	Runtime resources copied or packaged for the target
FBX models	Their glTF conversion, shipped as `<name>.fbx.glb` beside the FBX's path (see FBX models)

Scene and bundle setup therefore runs as compiled code, without parsing the editor's YAML scene and bundle files at runtime. Lua and external assets remain resources; export does not convert them all into C++.

Source Code export lets you inspect and build this code with standard toolchains. **Desktop** and **Web** compile it into the final application. Editor Play mode has a separate build workflow; use export for distribution.

> **Asset directory contents**
>
> Files inside the assets directory ship as they are, except FBX models, which ship as their glTF conversion. When that directory is left at the project root, that includes the `.scene`, `.bundle`, and `project.yaml` files. The game never reads them, so point the assets directory at a dedicated folder to keep authoring files out of the build.

Opening the Export Window

Choose **File → Export Project...** from the menu bar, or press the **Export** button in the toolbar.

Export modes

The window opens with three modes to choose from:

Mode	Output	Use it when
Source Code	A self-contained buildable C++ project (engine source included) for Android, iOS, Web, Windows, macOS, and Linux	You want full control of the build, target mobile platforms, or integrate with CI
Desktop	A ready-to-run native executable for the machine you are on (Linux, Windows, or macOS)	You want a playable desktop build without touching a compiler
Web	HTML + JavaScript + WebAssembly via Emscripten[5]	You want a browser build without running `emcmake` yourself

Selecting a mode opens its settings screen; the **Back** button returns to the mode selection.

Desktop and **Web** run the Source Code pipeline internally: they generate the full project into a build cache inside your project (`.doriax/export/desktop` or `.doriax/export/web`), compile it there, and copy only the final artifacts to your chosen destination folder.

Common settings

All modes share these settings:

Setting	Meaning
Target / Destination Directory	Where the output goes. Source Code requires an empty directory; Desktop and Web overwrite existing files in the destination
Start Scene	Scene loaded when the exported game launches
Shaders	The shader variants compiled into the build (pre-filled from your scenes; add or remove entries as needed)

The output folder is remembered **per project and per export mode**, so switching from Desktop to Source Code brings back the folder you last used for that mode rather than overwriting one with the other. A project that has never been exported starts from the **Default Export Directory** in Editor Settings (p. 68), when one is set. Both are machine-local: they live in the project's `.doriax/user/build.yaml`, which is not committed, not in `project.yaml`.

Start Scene is the same setting as Project Settings → General (p. 55); changing it here changes it for the project.

The assets and Lua folders that ship with the build come from Project Settings (p. 52), not from this window: stored references are relative to those directories, so exporting a different one would break every path in the scenes.

The experimental **Native Resource Pack** choice also comes from **Project Settings → Directories**. It is off by default; see Native resource pack for its output and runtime restrictions.

Application identity and platform settings

The Export Window does not ask for the application's name, identifier, version or icons: those come from Project Settings (p. 55), so every export mode and the command line produce the same identity. What the exporter writes with them depends on the target:

5. https://emscripten.org/

Written into	From	Applies to
Web page title, favicon, HTML shell, `<head>` include, and the game portal SDK	Web settings (p. 60)	Web mode and Source Code
`.desktop` launcher (`Name`, `Comment`, `Categories`) and `app_icon.png`	Linux settings (p. 61) and the window icon	Linux builds from any mode
Executable icon and `VERSIONINFO` resource (`app_icon.rc`)	Windows settings (p. 61) and the window icon	Windows builds from any mode
`Info.plist`, bundle identifier, `AppIcon` asset set, and on iOS the Google Mobile Ads frameworks and the StoreKit adapter	macOS (p. 61) and iOS (p. 62) settings	Source Code exports (the generated Xcode workspaces)
`build.gradle` (including the AdMob and Play Billing libraries), `AndroidManifest.xml`, `strings.xml`, `styles.xml`, launcher icons	Android settings (p. 62)	Source Code exports (the generated Android Studio workspace)

The executable and CMake target are still named after the **project name**, not the application name, so renaming the application does not rename the binary.

Shaders

The shader list is populated automatically from every saved scene, including a live scan of the currently open scenes — so a component added since the last save (for example a shadow-casting 2D light) is already accounted for.

The editor records a scene's variants while drawing it and saves them in the scene file. Closed scenes, and every scene in a command-line export (p. 168), use the saved ones. If a scene file, or a bundle it uses, was changed outside the editor, the export warns and names the scene, since a missing variant makes the game exit at startup. Open and save the scene in the editor to update it. Changes to model or material files are not detected.

Each entry shows where it came from:

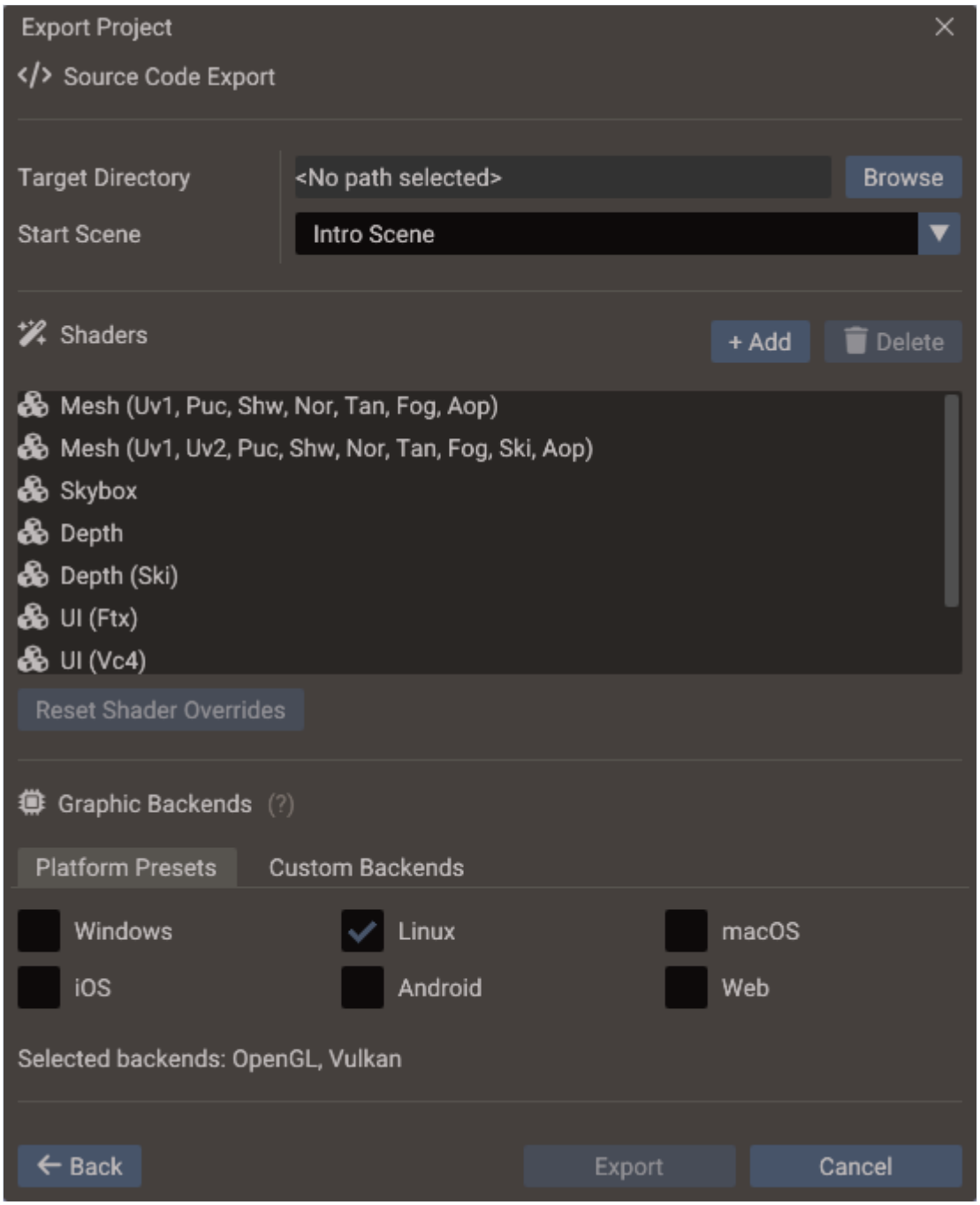
Icon	Entry	Meaning
Cubes	Normal	Collected from the project scenes
Plus	Addition	Added by hand with **Add**
Trash	*(Excluded)*, greyed out	Removed by hand, kept in the list so it can be restored

Use **Add** if your game enables a feature only at runtime from scripts (say, turning on SSR) so its shaders ship too. **Delete** on a collected shader does not remove the row — it marks it *(Excluded)* and greys it out, so you can always see what a build is leaving out and why the list is shorter than the scenes suggest. Selecting an excluded entry turns the button into **Restore**, and **Reset Shader Overrides** clears every manual addition and exclusion at once.

Additions and exclusions are saved in `project.yaml` under `export`, and they are **shared by all three export modes** — a shader you exclude for a Desktop test build is excluded from the Source Code export too. A command-line export (p. 168) applies them as well, unless `--shader` lists the shaders itself, and AI agents edit them with the `set_export_shaders` tool. Forked shaders are stored by their project-relative path rather than by a session id, so the list survives reopening the project. Effect functions follow `#`, as `fork#vertexOffset#coverage`: `mesh:8@#offsets/wind.glsl` is the built-in mesh with the engine's wind, and `mesh:8@##coverage/see_through.glsl` the built-in mesh with see-through.

Source Code mode

Exactly the classic export: choose the **graphic backends** to include (OpenGL, OpenGL ES 3, Direct3D 11, Metal for macOS and iOS, Vulkan) — this decides which shader formats are compiled in — and get a buildable CMake project in the target directory.



The source builds for every supported operating system regardless of this choice; only the shaders are affected, so include every backend the project may eventually be built with. See Shader compilation for which target needs which backend.

Choosing the backends

The **Graphic Backends** section has two tabs over the same selection:

Tab	Use it to
Platform Presets	Tick the platforms you intend to ship — Windows, Linux, macOS, iOS, Android, Web — and let the editor select the backends each one needs
Custom Backends	Tick individual backends, with the shader language of each shown in its tooltip

Switching tabs keeps the current selection, and the line under the tabs always spells out which backends the export will include. The two work together rather than overriding each other: a backend you tick by hand is kept even when a preset also covers it, and turning a platform off only clears the backends that preset added.

A project that has never configured this opens on **Platform Presets** with the platform you are running on already ticked; once a selection is saved, the window reopens on **Custom Backends** showing it. The selection is stored in `project.yaml` under `export.sourceCode.graphicBackends`, so it travels with the project.

Preset	Selects
Windows	OpenGL, Vulkan, Direct3D 11
Linux	OpenGL, Vulkan
macOS	Metal (macOS), OpenGL
iOS	Metal (iOS)
Android	OpenGL ES 3
Web	OpenGL ES 3

Generated output structure

```
output/
├── CMakeLists.txt       ← build system entry point
├── doriax.py            ← configure and build script
├── core/  libs/  platform/  renders/  workspaces/
│                        ← engine runtime source (copied from the editor's SDK)
├── shaders/             ← compiled shader headers for the selected backends
└── project/
    ├── main.cpp         ← generated startup entry point
    ├── *.cpp            ← generated scene and bundle factory sources
    ├── assets/          ← contents of the project's assets directory
    │                    ← (or resources.pak only, when packing is enabled)
    ├── lua/             ← contents of the project's Lua directory
    │                    ← (empty directory kept when packing is enabled)
    └── scripts/         ← your registered C++ scripts
```

Those three trees are filtered rather than mirrored, which matters when the assets and Lua directories are left at their default `<Project root>` and all three come from the same folder:

- **C++ sources and headers** only ever reach `scripts/`, at their path relative to the project root — a script kept in `scripts/CharacterController.cpp` therefore exports as `project/scripts/scripts/CharacterController.cpp`, which is the path the generated `scene_scripts.cpp` includes it by.
- **Lua sources** only reach `lua/`, the root `lua://` and `require` resolve against; they are not copied to `assets/` as well. A Lua file outside the Lua directory has no other home, so it still ships as an asset.
- **Data files** a script may read through either prefix (`.json`, `.txt`, `.csv`, `.dat` and similar) are copied to both trees.

A folder appears in the export only when a file lands in it, so a folder holding nothing but scripts leaves no empty directory behind. The exception is a packed Source Code export, which keeps an empty `project/lua/` directory for the generated Xcode workspace.

Export folders inside the project, such as `<project>/export`, are left out of everything an export copies, so an export never copies itself or an earlier one. Besides its own folder, that is any folder holding a Source Code export (`doriax.py`), a Desktop build (`MyProject` or `MyProject.exe`) or a Web build (`MyProject.wasm`). The target can't be the project directory itself, or its assets or Lua directory.

Build it afterwards with the appropriate platform toolchain, or run `python3 doriax.py` in the export folder to configure and build it in one step. See Build Script (p. 748).

Native resource pack

Enable **Project → Project Settings → Directories → Native Resource Pack** to combine the exported `assets/` and `lua/` trees into one `resources.pak`. This experimental, project-wide option is intended for ready-to-run **Desktop** exports and Android builds made from **Source Code** output. It has no effect on **Web** mode, which continues to produce Emscripten's `.data` bundle.

The resulting layout depends on the export mode:

Mode	Resource output when packing is enabled
Desktop	`resources.pak` beside the executable; loose `assets/` and `lua/` directories are removed
Source Code	`project/assets/resources.pak`; the loose contents are removed, while the empty `project/lua/` directory is retained for generated Xcode workspace compatibility
Web	Unchanged; Emscripten emits its normal `.data` bundle

The pack preserves the usual logical names. High-level loaders, Lua `require()`, script entries, and `Data::open()` can read plain asset-relative paths, `asset://` paths, and `lua://` paths transparently; external buffers and images referenced by GLTF models work the same way. Absolute paths, `data://`, and other schemes continue to resolve outside the pack. Editor Play mode is unaffected: it always reads the project's loose source files.

> **Experimental limitations**
>
> - `File` opens real filesystem files and cannot open a packed entry. Use `Data` (or an engine resource loader) for read-only packaged resources. This also means each opened entry is read into memory rather than exposed as a streaming file handle. `FileData.newFile()` follows the same rule: a packed path returns an in-memory `Data` object even when a file handle was requested.
> - A pack must remain below **2 GiB**. Export stops with an error if the combined header and resource data exceed that limit.
> - Packing only obfuscates the bytes; it does **not** encrypt or securely protect shipped assets.
> - `resources.pak` and `resources.pak.tmp` are reserved, case-insensitive names at the top level of the configured assets directory. Rename any file or directory with either name before exporting, even when packing is disabled.

If you disable the option and export again, the exporter removes a stale pack and ships the loose resource directories again. Command-line exports honor the saved project setting; there is no separate CLI switch.

FBX models

A game never reads FBX. Export ships the glTF conversion of every FBX in the assets directory beside it, as `<name>.fbx.glb`, and the engine loads that file wherever the `.fbx` path is used. The `.fbx` itself is not exported. See 3D Graphics — FBX models (p. 251).

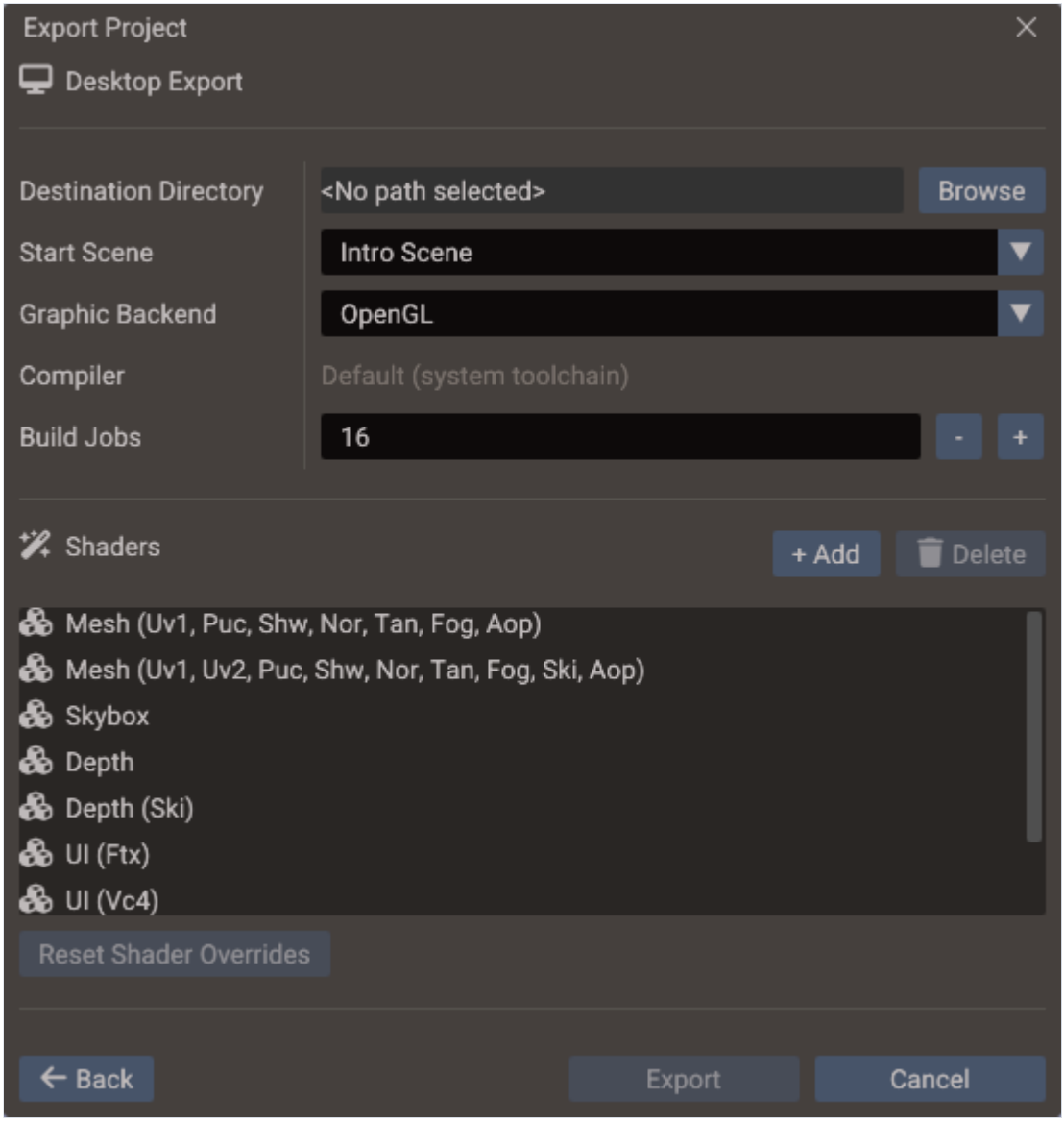
Conversions that are missing or out of date, such as on a fresh clone or after an FBX or one of its textures changed, are made during the export, so a clean checkout exports without opening the editor first. Models in open scenes are reloaded before their scenes are saved for the export.

A scene that is not open keeps the model parts it was saved with, and only the editor can rebuild them for a changed file. Each scene therefore records which version of an FBX its models came from, and export stops when that no longer matches the file on disk: open the scene, let its models reload, save it, and export again. A changed texture does not count.

Export also stops when an FBX cannot be converted, or when an asset already has the path a conversion ships as (`hero.fbx.glb` beside `hero.fbx`). A texture that a conversion links but that is now missing is reported as a warning, and the model is exported without it.

Desktop mode

Builds a native executable for the operating system the editor is running on, using CMake and the **compiler kit configured in Editor Settings** (p. 68) (or the system default toolchain when none is selected).



Setting	Meaning
Compiler	Read-only. Shows the kit this project builds with; change it in **Edit → Editor Settings → Desktop**
Build Jobs	How many compiler processes run in parallel. Shared with Play-mode script builds — see Parallel builds (p. 211)
Graphic Backend	The API the executable is built against, and the only shader format compiled

Both the compiler and the job count are machine-local: they are stored per project in `.doriax/user/build.yaml`, not in `project.yaml`, so a teammate on another machine keeps their own toolchain.

Choose the **Graphic Backend** before exporting. The available choices follow the host operating system:

Host	Backends
Linux	OpenGL *(default)*, Vulkan
Windows	OpenGL *(default)*, Vulkan, Direct3D 11
macOS	Metal *(default)*, OpenGL

The exporter passes the selection to CMake and compiles only the shader format needed by that backend. Vulkan builds require the Vulkan development files or SDK to be available to the compiler. macOS has no Vulkan option because its only driver, MoltenVK, is missing an extension the renderer needs — see Vulkan backend (p. 755).

The destination folder receives:

```
destination/
├── MyProject            ← the executable (MyProject.exe on Windows)
├── assets/              ← runtime resources (default)
├── lua/                 ← runtime Lua scripts (default)
├── resources.pak        ← replaces assets/ and lua/ when packing is enabled
├── app_icon.png         ← (Linux, with a project icon set)
└── MyProject.desktop    ← (Linux launcher entry)
```

Run the executable from that folder — it resolves the loose `assets/` and `lua/` directories, or `resources.pak`, relative to its working directory.

Application icon

Set a **project icon** in **Project Settings → Window → Icon** (square PNG, 256×256 or larger recommended) and desktop exports use it automatically:

- **Windows** — embedded into the executable (Explorer, taskbar, and window icon), alongside the version resource built from the Windows settings (p. 61).
- **Linux (X11)** — window and taskbar icon at runtime. Executable files on Linux cannot carry icons, so the export also ships `app_icon.png` and a ready-made `.desktop` launcher entry, whose name, comment and categories come from the Linux settings (p. 61); install it with `cp MyProject.desktop ~/.local/share/applications/` to get the icon in menus and docks — and on **Wayland**, where windows only receive icons through that entry.
- **macOS** — dock icon at runtime.

Apple bundles and Android launchers use their own icon settings on the Platforms tab (p. 60) instead, because they need asset catalogs and adaptive layers rather than a single PNG.

> **Requirements**
>
> Desktop mode needs **CMake** and a C++ compiler installed. The window warns you with install instructions when either is missing. Point the window at your installs in **Edit → Editor Settings → Desktop** if they are not on `PATH`. On macOS the **Xcode generator** is not supported for this mode (it produces an app bundle instead of a plain executable); use the default toolchain.

Web mode

Builds the project with Emscripten and copies `MyProject.html`, `.js`, `.wasm`, and `.data` (the packed assets) to the destination. The Web graphic backend is fixed to **WebGL 2 (OpenGL ES 3)**.

The editor locates the Emscripten SDK automatically from the `EMSDK` environment variable or `emcmake` on `PATH`. If neither is set, point at your `emsdk` folder once in **Edit → Editor Settings** (p. 70) **→ Web** — the path is remembered across projects, and the dialog reports whether the SDK was found. The Export Window shows the same status and warns before starting a build it cannot run.

On Windows the build runs with Ninja, taken from `PATH` or from the **C++ CMake tools for Windows** component of Visual Studio. Without either, Emscripten falls back to MinGW's `mingw32-make` on `PATH`, and the configuration step fails when that is missing too; see Building for HTML5 (p. 760).

The generated page is customized from the Web settings (p. 60): page title, favicon, an optional custom HTML shell, extra `<head>` markup, canvas resizing, and hiding Emscripten's default UI. The **Game Portal** setting

builds in the SDK of one web game portal, for the `WebPortal` (p. 626) class; export once per portal. See Monetization → Web portals (p. 346).

> **Testing the build**
>
> Browsers do not load WebAssembly from `file://`. Serve the destination folder with any local web server, for example:
>
> ```
> python3 -m http.server --directory /path/to/destination
> ```

See Building for HTML5 (p. 760) for Emscripten installation and advanced options such as thread support.

Build progress and cancellation

While exporting, the window shows each phase, a progress bar (driven by the compiler's own progress output during Desktop/Web builds), and the last build log line. The full log streams to the **Output** panel. **Cancel** stops the export and terminates the running compiler processes.

The build cache

Desktop and Web modes keep their generated project and CMake build tree inside your project under `.doriax/export/desktop` and `.doriax/export/web`. The first export compiles the whole engine and takes a while; subsequent exports only rebuild what changed — typically just your scenes and scripts, finishing in seconds.

- Changing the graphics backend, compiler kit, generator, or Emscripten SDK wipes the cache's build tree automatically (CMake cannot switch these settings in place).
- The cache is safe to delete at any time; the next export simply starts from scratch.
- Keep `.doriax/` out of version control.

Export phases

Phase	Output
Scene factory generation	Generates a C++ factory function for each scene from its YAML
Bundle factory generation	Generates C++ factory functions for each bundle
Asset packaging	Copies and organizes resource files, ships FBX models as their glTF conversion, then optionally combines native assets and Lua files into `resources.pak`
Startup code generation	Generates `main.cpp` / entry point with scene registration
Engine template copy	Copies the runtime engine source and CMake/build files
Shader compilation	Translates shaders for the selected graphics backends
Configure & build *(Desktop/Web)*	Runs CMake and the compiler on the build cache
Collect artifacts *(Desktop/Web)*	Copies the executable or web files to the destination

If a scene cannot be saved or a file the export needs cannot be copied, the export stops and the Output panel names it. Folders the editor cannot read are left out instead.

Generated scene data

Meshes backed by a model file keep their geometry in the exported asset and load it at runtime. Terrain meshes with a configured heightmap also rebuild their runtime buffers from `TerrainComponent` settings and the exported heightmap. Other meshes without a runtime geometry source, including geometry created directly in the editor, embed their vertex and index buffers in the generated scene or bundle C++ as read-only static data. The scene factory copies those bytes into the runtime buffers without placing the full geometry blob on the thread stack.

Generated factories also construct large fixed-capacity components, such as mesh, tilemap, and 3D physics body data, in temporary heap storage before inserting them into the ECS. This prevents those payloads from inflating the scene factory's stack frame.

Scene stacks

`main.cpp` gives every scene two functions and registers them together, as `SceneManager::registerScene(id, name, load_<file>, add_<file>, {start-active scene ids}, resources_<file>)`, where `<file>` is the name of the scene's generated source, `scene_<name>_<id>`. The id keeps two scenes apart even when their names make the same C++ identifier, like `A-B` and `A B`. Both functions begin the same way: for the scene and every child scene it involves, a `new Scene()` registered with `SceneManager::setScenePtr`, the scene factory, then `initScripts` — skipping any that already exist. `load_<file>()` goes on to call `Engine::setScene` and `Engine::addSceneLayer` for the start-active scenes and to tear down every scene the new stack does not involve; `add_<file>()` stops after creating, and `SceneManager::addChildScene` (p. 604) puts the scenes on screen. A scene is not torn down while a stack that involves it is running, which is what keeps a stack a script adds from its `init` alive across the load.

Bundle instances

Each bundle becomes `bundle_<name>.h` and `bundle_<name>.cpp` with two entry points: `create_bundle_<name>(scene, root)`, the factory `main.cpp` registers with `BundleManager`, and `build_bundle_<name>(scene, root, members)`, the same factory appending every entity it creates to `members` — its own entities first, then each nested bundle's. A scene factory calls `create_bundle_<name>` for an instance nothing else refers to, and `build_bundle_<name>` when it needs the members: per-instance overrides, scene entities kept under a member, and component or script references into the instance are emitted against that vector instead of an entity id, because member ids are only assigned when the factory runs. Scene entities keep the ids they have in the editor; bundle members do not.

Large inline meshes still increase the generated source size, executable size, and C++ compile time. Prefer a GLTF or OBJ asset for multi-megabyte geometry that does not need to be stored directly in the scene.

Shader compilation

The shader builder translates shader source per graphics API, never per operating system. Desktop mode compiles only the format its **Graphic Backend** selection needs, Web uses WebGL 2, and Source Code mode compiles one format per backend you ticked:

Backend	Shader language	File suffix
OpenGL	GLSL 4.10	`glsl410`
OpenGL ES 3	GLSL ES 3.00 (also WebGL 2)	`glsl300es`
Direct3D 11	HLSL 5	`hlsl5`
Metal (macOS)	MSL 2.1	`msl21macos`
Metal (iOS)	MSL 2.1	`msl21ios`
Vulkan	SPIR-V 1.0	`spirv10`

Metal is the one backend split by operating system, because the cross-compiler emits different MSL for macOS and iOS.

On Windows, Direct3D 11 shaders are also compiled to bytecode, so the game does not compile them each time it starts. Exports made on Linux or macOS keep the HLSL source.

Choosing backends

The runtime loads only the format matching the backend it was compiled with, so pick the backends by how the exported source will be built:

Target	Backends to include
Linux	OpenGL, Vulkan
Windows	OpenGL, Direct3D 11, Vulkan
macOS	Metal (macOS), OpenGL
iOS	Metal (iOS)
Android	OpenGL ES 3
Web	OpenGL ES 3

This is exactly the mapping the Platform Presets tab applies, so ticking your target platforms there is the same as reading this table.

Including a backend you never build with only costs export time and file size. Leaving one out that you do build with makes the game report the shaders as missing at startup, with the `doriax-editor shaders` command needed to generate them.

If shader compilation fails, the Output panel reports the shader name, stage, backend, and the compiler error. Fix the shader source and re-export.

Custom (forked) shaders are compiled and shipped through this same pipeline, embedded into the build alongside the built-in ones. See Custom Shaders — Export and runtime (p. 154).

Platform toolchains

Source Code exports are built afterwards with the appropriate native toolchain:

Target platform	Toolchain	Notes
Windows	CMake + MSVC or MinGW	Requires CMake 3.16+, Windows SDK
Linux	CMake + GCC or Clang	Requires X11 or Wayland dev libraries
macOS	CMake + Xcode or CLang	Requires Xcode command-line tools
Android	Gradle + Android NDK	Requires Android Studio or SDK command-line tools
iOS	Xcode workspace	Requires a Mac with Xcode
HTML5 / Web	Emscripten	Requires `emcmake cmake`

Desktop and Web modes run the Linux/Windows/macOS and Emscripten toolchains for you.

VSync in desktop exports

The project-level **VSync** option is written into exported desktop CMake projects. With VSync disabled, Linux GLFW, Windows/Linux Sokol OpenGL, and Windows Direct3D 11 builds use swap interval `0`. Vulkan builds prefer Immediate presentation, use Mailbox when Immediate is unavailable, and fall back to the always-supported FIFO mode when required by the driver or surface.

macOS Metal currently has no uncapped application-loop path in Doriax. Metal exports therefore remain synchronized and emit a CMake warning when project VSync is disabled. See Project Workflow — VSync (p. 53) for the full behavior table.

A minimized Windows, Linux, or macOS build stops drawing but keeps updating 60 times a second, so scripts, physics, and sounds go on. On macOS this also applies to a window that is fully covered or on another Space, and App Nap can make a silent game update less often. Until the window comes back, `onDraw` (p. 372) is skipped and gamepads are released and not read.

Window settings in desktop exports

The project-level **Window** settings (mode, size, resizable, title) are written into exported desktop CMake projects the same way. Linux GLFW builds honor all of them. Windows and macOS Sokol builds honor the size, title, and fullscreen mode, but have no maximized or non-resizable window support. Web, Android, and iOS exports ignore window settings entirely — the game always fills the browser canvas or the device screen. See Project Workflow — Window (p. 54) for the full behavior table.

Exporting from the command line

The Source Code pipeline is available headlessly through the `doriax-editor` CLI, which is ideal for build servers and CI/CD:

```
doriax-editor export --project ./MyGame --out ./build/MyGame --backend "vulkan,opengles"
```

See Command-Line Tools (p. 167) for the full `export` and `shaders` reference.

Build options

See Build Options (p. 357) for a full list of CMake flags and compile-time defines you can set to control engine features in the export. Source Code export also writes engine capacity macros (`MAX_SUBMESHES`, `MAX_BONES`, and others) from the largest values used in your scenes and bundles — see Engine capacity macros (p. 359).

Tips

- Keep export destinations and the `.doriax/` folder out of version control.
- Test exported builds on actual devices — some graphics, input, and memory behaviors differ between the editor's desktop preview and the target platform.
- If a shader is reported missing at runtime, re-export (the scene shader list refreshes automatically) or add the named shader manually with **Add** in the Shaders section.

Command-Line Tools

The `doriax-editor` executable doubles as a **command-line tool**. `export` and `shaders` are headless: they run and exit without opening a window, which is what build servers and CI want. `benchmark` opens a window, times a scene, prints JSON, and exits.

```
doriax-editor export    --project <path> --out <dir> [options]
doriax-editor shaders   --out <dir> --shader <spec> [options]
doriax-editor benchmark --project <path> [--scene <scene>] [options]
```

Running `doriax-editor` with **no arguments** launches the normal graphical editor. `--help`, `-h` or `help` print the top-level usage. Anything else is reported as an unrecognized argument and exits with code `2`, so a mistyped subcommand fails instead of opening a window.

> **Windows: two executables**
> Windows builds ship both. `doriax-editor.exe` is a GUI application — it re-attaches to the parent console so subcommand output still appears in your terminal, but `cmd` and PowerShell do not wait for a GUI application, so the prompt returns before the command finishes and its exit code never reaches you. `doriax-editor-cmd.exe` is the same program linked as a console application: use it for scripting and CI. Both accept the same subcommands, and both open the editor when given no arguments.

On Linux and macOS there is a single binary. `export` and `shaders` run before the window backend starts, so they work on headless machines. `benchmark` still opens a window.

> **macOS: the binary is inside the app bundle**
> The editor ships as `Doriax.app`, so the command-line entry point is `Doriax.app/Contents/MacOS/Doriax`:
>
> ```
> /Applications/Doriax.app/Contents/MacOS/Doriax export \
>   --project ./MyGame --out ./build/MyGame
> ```
>
> Alias it or add that directory to your `PATH` if you use it often. The loose-binary zip (p. 12) has no bundle, so there the entry point is just `./doriax-editor`.

`export` — build a project to a target

Serializes scenes, generates the C++ glue and bundle factories, compiles shaders, copies assets (FBX models as their glTF conversion) and the engine runtime, and writes a self-contained, buildable project directory — the same output as the Export Window (p. 155), but scriptable.

```
doriax-editor export --project ./MyGame --out ./build/MyGame --backend vulkan
```

Options

Option	Description
`-p`, `--project <path>`	**Required.** Project directory or a `project.yaml` file.
`-o`, `--out <path>`	**Required** (unless `--list-scenes`). Destination directory. Must be empty.
`--assets <path>`	Asset directory, relative to the project or absolute. Defaults to the project's asset directory.
`--lua <path>`	Lua directory, relative to the project or absolute. Defaults to the project's Lua directory.
`--start-scene <id|name>`	Override the start scene by numeric ID, scene name, or file stem.

Option	Description
`--backend <list>`	Graphic backend(s) to compile shaders for. Repeatable, or a list (see below). Defaults to **all** supported backends.
`--shader <spec>`	Shader(s) to compile, replacing what the scenes report and the project's shader additions and exclusions. Repeatable (see Shader specs). If omitted, shaders discovered while regenerating scenes are exported, with the project's additions and exclusions applied.
`--list-scenes`	Print the project's scenes (`id`, `name`, `path`) and exit.
`-h`, `--help`	Show usage for this subcommand.

Asset references are stored relative to the directories configured in the project, so `--assets` and `--lua` only make sense when they point at those same folders. Exporting a different root ships files the stored paths do not resolve against.

Project settings the command line uses

The export reads `project.yaml`, so the Project Settings (p. 55) that shape the output apply here exactly as they do in the editor — with one deliberate exception that a flag owns instead:

Project setting	In a CLI export
Application name, identifier, version, build	Applied
Per-platform settings (Web, Linux, Windows, macOS, iOS, Android)	Applied — the generated Xcode and Android Studio workspaces are configured from them
Window settings, VSync, canvas	Applied
2D and 3D Physics	Applied — a disabled backend is left out of the CMake, Xcode and Android Studio builds
Assets and Lua directories	Applied unless `--assets` / `--lua` override them
Start scene	Applied unless `--start-scene` overrides it
Native Resource Pack	Applied. There is no command-line override
Shader additions and exclusions	Applied unless `--shader` lists the shaders, which are then compiled exactly as given
Graphic backends saved by the Export Window	**Ignored.** `--backend` decides, and defaults to every supported backend

Ignoring the backends keeps a scripted build reproducible from its command line: a CI job compiles for what its flags say, not what someone last ticked in the Export Window. Pass the same backends you selected there when you want the two to match.

When Native Resource Pack is enabled, the source output contains `project/assets/resources.pak` in place of loose asset and Lua contents; see Native resource pack (p. 160) before using it in an automated Android export.

Shader variants from scenes

The command line draws nothing, so it exports the shader variants saved in each scene file (see Shaders (p. 157)). A scene changed outside the editor, or using a bundle that was, gets a warning, since a missing variant makes the game exit at startup. Open and save it in the editor, or list the variants with `--shader`.

List a project's scenes

Useful for discovering a start-scene ID before exporting:

```
doriax-editor export --project ./MyGame --list-scenes
# 1   MainMenu   scenes/main_menu.scene
# 2   Level01    scenes/level_01.scene
```

`benchmark` — measure scene FPS

Opens the editor, loads a project, and times the **scene camera** with VSync forced off and no frame cap, whether or not its window has focus. Unlike `export` and `shaders`, this is **not headless**: it needs a window and a GPU, waits until meshes, foliage, detail levels, and shaders have finished loading (90 seconds, then it measures anyway), warms up, then records FPS and draw counters. The JSON result is printed to stdout; `--out` writes the same text to a file. The process then exits.

Use it to compare scene settings — mesh LOD, depth prepass, foliage shadows — on the same machine without clicking through Play.

```
doriax-editor benchmark --project ./MyGame --scene Level01 --out ./bench.json
```

On Windows call `doriax-editor-cmd` so the shell waits for the exit code. On macOS the entry point is `Doriax.app/Contents/MacOS/Doriax`, same as the other subcommands.

Options

Option	Description
`-p, --project <path>`	**Required.** Project directory or a `project.yaml` file.
`-s, --scene <scene>`	Scene name, scene file name, or path. Defaults to the project's start scene.
`--frames <n>`	Measured frames after warmup. Default `180`.
`--warmup <n>`	Warmup frames after resources finish loading. Default `90`.
`--mesh-lod <on|off>`	Override the scene's mesh LOD (p. 275) setting for this run.
`--depth-prepass <on|off>`	Override the scene's depth prepass (p. 276) setting for this run.
`--name <label>`	Label written into the JSON `name` field. Defaults to the scene name.
`--out <path>`	Write the JSON to this file as well as stdout.
`-h, --help`	Show usage for this subcommand.

The capture uses the scene's game camera, hides the editor grid, origin, gizmos, and selection outline, and does not enter play mode — scripts that start in `onStart` are not run. The viewport is the scene view inside the editor's default window; the JSON reports its size.

Result JSON

```
{
  "name": "Level01",
  "scene": "Level01",
  "meshLod": true,
  "depthPrepass": false,
  "viewport": [1280, 720],
  "frames": 180,
  "warmupFrames": 90,
  "elapsedSeconds": 3.12,
  "wallFps": 57.69,
  "avgFps": 58.20,
  "minFps": 54.10,
  "maxFps": 60.00,
  "avgMs": 17.33,
  "avgDrawCalls": 142.0,
  "avgInstances": 8100.0,
  "avgTriangles": 1850000.0
}
```

`wallFps` is measured frames divided by wall time; `avgFps` averages `Engine::getFramerate()` per frame. Draw, instance, and triangle averages come from `Engine::getFrameStats()` (p. 366).

`shaders` — generate shader files standalone

Compiles shader variants directly to an output directory, without exporting a whole project. Use it to pre-bake a shader cache or produce shader data for custom tooling.

```
doriax-editor shaders --out ./shaders --shader mesh:Uv1,Nor --backend opengles
```

Options

Option	Description
`-o`, `--out <path>`	**Required.** Destination directory for the generated shader files.
`--format <format>`	Output format: `binary` (default, also `sdat`), `header`, or `json`.
`--backend <list>`	Graphic backend(s) to compile for. Repeatable or a list. Defaults to **all** supported backends.
`--shader <spec>`	**Required.** Shader(s) to generate. Repeatable (see Shader specs).
`-h`, `--help`	Show usage for this subcommand.

Graphic backends

A shader is compiled for a **graphics API**, not for an operating system — a Windows build needs GLSL, HLSL, or SPIR-V depending only on how it was configured. Both subcommands accept these backend names (case-insensitive):

Value	Alias	Shader format
`opengl`	`glcore`	`glsl410`
`opengles`	`gles3`	`glsl300es`
`d3d11`		`hlsl5`
`metal-macos`	`metal`	`msl21macos`
`metal-ios`		`msl21ios`
`vulkan`		`spirv10`
`all`		selects every supported backend

One file is written per shader and backend, named `<shader>_<format>`. At startup the runtime loads the format matching the backend it was built with, so an export must include every backend the final binary may use — see Choosing backends (p. 165).

Pass `--backend` multiple times, or give a single list separated by `,`, `+`, `|`, or `;`:

```
doriax-editor export -p ./MyGame -o ./build --backend vulkan --backend opengl
doriax-editor export -p ./MyGame -o ./build --backend "vulkan,opengl,opengles"
```

Shader specs

A `--shader` value is a **shader type** with optional **properties**, separated by a colon, and, for `export`, an optional **custom shader** after `@`:

```
<type>[:<properties>][@<custom shader>]
```

- **Type** (case-insensitive): `points`, `lines`, `mesh`, `sky` (or `skybox`), `depth`, `ui`, `postprocess` (or `post`), `water`, `underwater`.
- **Properties** are either:
  - a comma-separated list of property names, e.g. `mesh:Uv1,Nor`, or
  - a numeric bitmask, e.g. `mesh:0x3` or `mesh:12`.

Property names come from the engine's shader system for that shader type; an unknown property or type is reported as an error. Omit the properties to generate the base variant of a type (e.g. `--shader depth`).

The custom shader is a fork's project-relative base path, followed by a vertex offset and a coverage function (p. 287) after `#`, as `fork#vertexOffset#coverage`. Leave the fork out for the built-in shader: `mesh:Uv1,Nor@#offsets/wind.glsl`, `depth:Uv1@##coverage/dissolve.glsl`, `mesh:Uv1@shaders/myMesh`. It needs the project's files, so the `shaders` command, which loads no project, builds built-in shaders only.

```
# Base mesh shader plus a UV+normal variant, for WebGL 2 and Direct3D 11
doriax-editor shaders --out ./shaders \
  --shader mesh \
  --shader mesh:Uv1,Nor \
  --backend opengles --backend d3d11
```

Exit codes

Code	Meaning
`0`	Success.
`1`	Runtime failure (project failed to load, export/shader build error). The error is printed to `stderr`.
`2`	Invalid or unrecognized arguments. The error is printed to `stderr`, usage to `stdout`.

Use in CI/CD

Because the commands are headless and return meaningful exit codes, they drop straight into a pipeline. On Windows runners call `doriax-editor-cmd`: the default PowerShell shell does not wait for the GUI binary, so the step can pass even when the export failed. Example GitHub Actions step:

```
- name: Export game (Vulkan + WebGL 2)
  run: |
    doriax-editor export \
      --project ./MyGame \
      --out ./build/MyGame \
      --backend "vulkan,opengles"
```

The exported directory still needs its native toolchain to produce final binaries — see Platform toolchains (p. 165) and Building from Source (p. 746).

A clean checkout has no FBX conversions yet; the export makes them. It fails when a committed scene was saved with an older version of an FBX it uses — see Export Window — FBX models (p. 160).

See also

- Export Window (p. 155) — the graphical equivalent and full export pipeline.
- Rendering Pipeline (p. 276) — mesh LOD, instance culling, and the depth prepass.
- Build Options (p. 357) — CMake flags for the generated project.

MCP Server

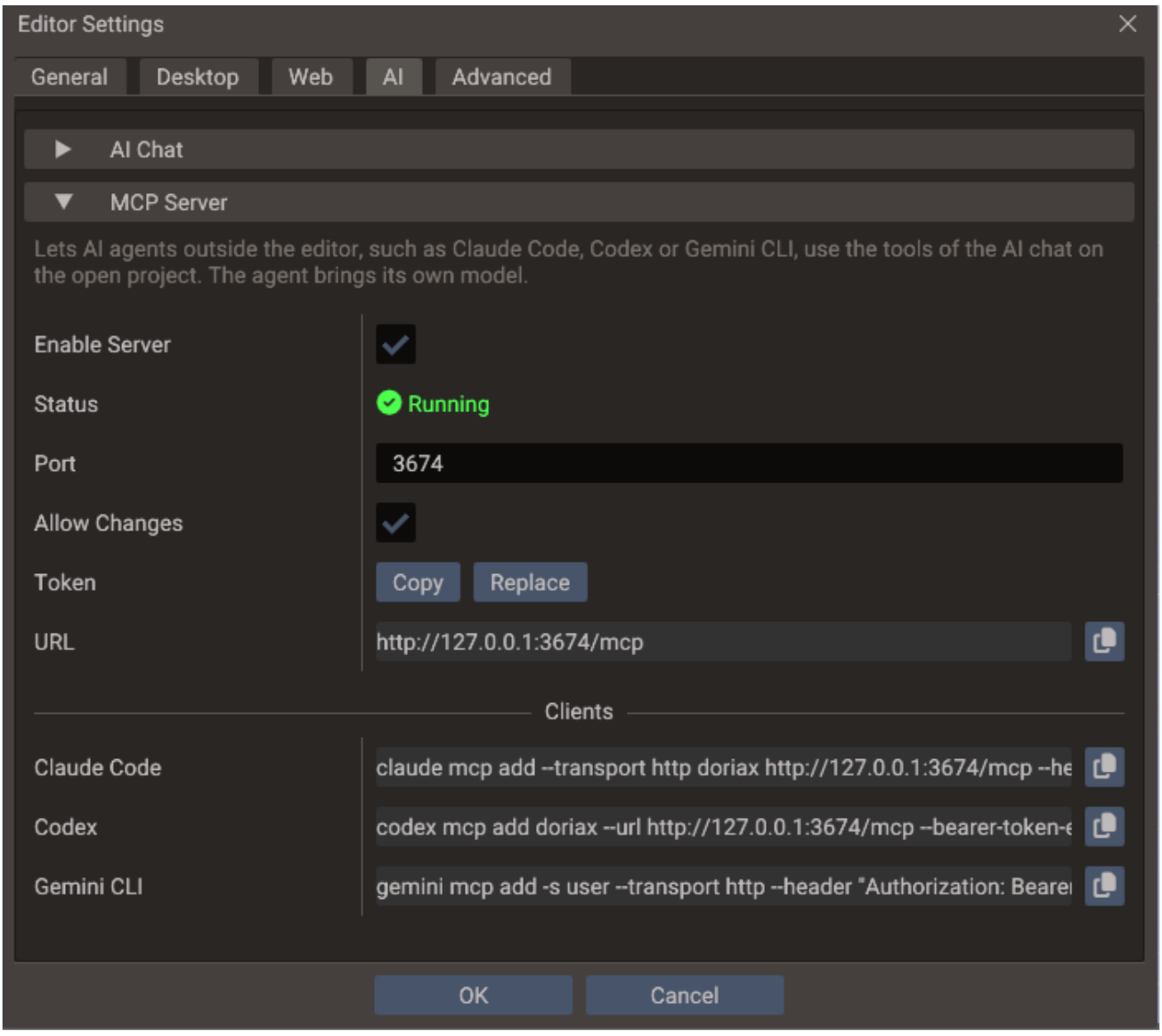
The editor can run a Model Context Protocol[6] (MCP) server, so AI agents outside it — Claude Code, Codex, Gemini CLI and other MCP clients — can work on the open project with the same tools as the built-in AI Chat: inspect scenes and entities, add components, write scripts, fork shaders, run a scene and read the output log.

The agent brings its own model and account. The editor only runs the tools, so no API key is needed here and the AI Chat can stay closed. The server works on whichever project the editor has open, so opening another project changes what agents see, and an agent can open or create one itself.

Turning it on

1. Open **Edit → Editor Settings** (p. 71) **→ AI** and expand **MCP Server**.
2. Tick **Enable Server** and press **OK**.

The Output panel reports `MCP server listening on http://127.0.0.1:3674/mcp`, and the server starts with the editor from then on.



6. https://modelcontextprotocol.io

Setting	Default	Effect
Enable Server	Off	Listens for MCP clients on `127.0.0.1` only, so nothing outside this computer can reach it
Status	—	**Running**, **Stopped**, or why the server could not start, such as a port another program holds
Port	`3674`	The local TCP port. Change it when another program uses it, or to run two editors with the server on
Allow Changes	On	Off leaves agents only the read-only tools, which inspect and search the project
Token	Generated	The secret every request must carry. **Copy** puts it on the clipboard and **Replace** makes a new one
URL	—	The endpoint, `http://127.0.0.1:<port>/mcp`
Clients	—	Ready commands that add the editor to Claude Code, Codex and Gemini CLI, each with a copy button

The token is created and kept the first time the section opens, and replacing it takes effect at once, so a command copied from here keeps working whether the dialog closes with **OK** or **Cancel**. The other fields apply on **OK**.

Connecting Claude Code

Copy the **Claude Code** row and run it in a terminal:

```
claude mcp add --transport http doriax http://127.0.0.1:3674/mcp \
  --header "Authorization: Bearer <token>"
```

The command adds the server for the folder you run it in, so run it in the project folder, where Claude Code can also read the project's files. Add `--scope user` after `--transport http` to have it in every folder. In a Claude Code session, `/mcp` then lists `doriax` as connected, with its tools.

Connecting Codex

Copy the **Codex** row and run it in a terminal:

```
codex mcp add doriax --url http://127.0.0.1:3674/mcp --bearer-token-env-var DORIAX_MCP_TOKEN
```

Codex keeps the token out of its configuration: it reads it from the `DORIAX_MCP_TOKEN` environment variable each time it connects. Set that variable to the token (**Copy** on the **Token** row) where Codex runs, for example in your shell profile:

```
export DORIAX_MCP_TOKEN=<token>
```

On Windows, `setx DORIAX_MCP_TOKEN <token>` sets it for terminals opened afterwards.

Connecting Gemini CLI

Copy the **Gemini CLI** row and run it in a terminal:

```
gemini mcp add -s user --transport http --header "Authorization: Bearer <token>" \
  doriax http://127.0.0.1:3674/mcp
```

`-s user` saves the server in your user settings. Without it, Gemini CLI writes it to `.gemini/settings.json` in the current folder, token included, where it could end up in version control.

Other clients

Any other client that speaks MCP over Streamable HTTP and can send a custom header works: give it the **URL** and the header `Authorization: Bearer <token>`. A client that can only start local (stdio) servers needs a stdio-to-HTTP bridge in between.

Keep the token out of version control

The token belongs to one machine and lets its holder change your project. When a client's configuration is committed, such as Claude Code's project-scoped `.mcp.json`, read the token from an environment variable instead of writing it there:

```
{
  "mcpServers": {
    "doriax": {
      "type": "http",
      "url": "http://127.0.0.1:3674/mcp",
      "headers": { "Authorization": "Bearer ${DORIAX_MCP_TOKEN}" }
    }
  }
}
```

What agents can do

The server offers every tool the AI Chat uses, with the same checks and the same results, plus `open_project` and `create_project`, which only MCP clients get: the chat runs inside the editor, and switching projects would end its own turn. Each tool is marked read-only or not (`readOnlyHint`), which clients can use when they ask you to confirm a call.

When a client connects, the server also sends the editor's main rules: build scenes in the editor and leave scripts to drive them, share repeated entities through bundles, give a HUD its own UI scene, and animate with native animations. The other rules the AI Chat's model has in its prompt, such as how scripts are written and how physics bodies get their shapes, come from `get_editor_guidance`, a read-only tool only MCP clients get. Agents are asked to call it once before their first change.

Some of those rules, which the AI Chat follows too, cover setting a project up. The agent gives the assets and Lua directories (p. 52) their own folders before adding files, so an export leaves the project's own files out, and asks you first when the project already has content at its root. It keeps the script language a project already uses. In a project without scripts it writes C++ once `check_cpp_build_tools` finds CMake and a compiler. It uses Lua instead when that check fails, when the build fails outside its code, or when its first C++ script still does not compile after two fixes, and tells you why.

How calls run

- Each call runs on the editor's main thread between frames, one at a time, exactly as if the AI Chat made it. A long action such as an export holds the editor until it ends.
- Scene edits go through the editor's undo history, so **Edit → Undo** reverts what an agent did.
- Agents see the files the Resources Browser (p. 106) shows: searches skip hidden files and folders, and the file tools refuse paths in `.trash`, where deleted files wait for undo.
- Each change an agent makes is logged in the Output panel as `MCP: <what it did>`, and a failed call as a warning or an error.
- While a project is loading, calls wait. While a scene is saving, calls that change the project wait as well, and read-only ones still run. A call still waiting after two minutes fails, and the agent is told to try again.
- A call the client gives up on before it starts, for example when you cancel it, is dropped. One already running is asked to stop early where the action supports it, as an export or an asset download does.

> **Scenes are edited through the tools, not on disk**
> The editor holds open scenes and `project.yaml` in memory and writes them back when it saves, so an agent that edits those files directly loses its change at the next save or overwrites yours. The server tells agents this; remind yours if it keeps editing the files. Scripts are ordinary files, but writing one with the `update_script_file` tool also refreshes the properties of every entity that uses it.

Saving an agent's work

Agents save with `save_scene`, `save_all_scenes` and `save_project`, like the AI Chat. Two saves would need a dialog that only you can answer, so the tools take a path instead:

- **A scene that was never saved.** `save_scene` with a project-relative `path`, such as `scenes/Level1.scene`, saves it there. Without one, and with `save_all_scenes`, which takes no paths, the editor opens the **Save Scene** dialog for you.
- **A temporary project** (p. 48)**.** `save_project` with an absolute `path` moves the project out of the system temp folder, as **Save Project** does, into a directory that is empty or does not exist yet. An optional `name` renames it too. A project that is not temporary is saved where it is, without `path`.

A scene saved before the editor drew its last changes, or while something in it still loads, lists only the shader variants (p. 278) drawn so far. `save_scene` then returns `shader_keys_stale`, and the agent selects the scene, lets it draw and saves it again, so an export does not warn about it.

Tools that would stop to ask you about unsaved work refuse instead, and tell the agent what to save first. `create_scene`, for example, closes the selected scene, so it refuses while that scene or one of its child scenes has unsaved changes.

Opening and creating projects

Two tools replace the project the editor has open, and only MCP clients have them:

- `open_project` opens an existing project directory, the one holding `project.yaml`, as **File → Open Project** does.
- `create_project` creates a project in an empty or new directory and opens it. It is named after the directory unless the agent passes `name`, and starts with an unsaved 3D scene, *New Scene*, which `save_scene` with a `path` keeps.

Both take an absolute path. The switch happens after the call returns, and the agent's next calls wait until the new project has loaded.

The editor does not drop work to switch, so both refuse where its menus would ask you first:

Refused while	Resolved by
A scene is playing, saving or loading	Stopping play mode (`control_play_mode`), or retrying once the save or load ends
A scene has unsaved changes	Saving it with `save_scene`, with a `path` if it has no file yet
A script in the Code Editor has unsaved edits	You saving or discarding them; the agent cannot
The project is temporary and holds work: a saved or changed scene	Moving it out of the temp folder with `save_project` and a `path`

The temporary project the editor opens at startup holds no work until you or an agent change it, so an agent can switch away from it at once.

Security

The server is meant for agents running on your own computer:

- It listens on `127.0.0.1` only.
- Every request must carry the token as `Authorization: Bearer <token>`. A missing or wrong token is answered with HTTP `401`.
- Requests from web pages are refused with HTTP `403`: the server checks the `Origin` header, so a site cannot reach it through your browser, not even by pointing its own domain at `127.0.0.1`.
- The token is stored obfuscated in `ai_keys.dat`, beside the editor's `settings.yaml`, and never in the project.

Treat the token like access to the editor itself. With **Allow Changes** on, an agent can write scripts and build settings, and those run on the next Play or export. A replaced token applies from the next request, so every connected client needs the new command.

Troubleshooting

Symptom	Cause and fix
Status says it could not listen on the port	Another program, or another editor with the server on, holds the port. Pick another **Port**, press **OK**, and add the server to the client again with the new URL
The client reports `401` or *Missing or wrong token*	The token was replaced, or the command came from another machine. Run the client's command from **Clients** again, after `claude mcp remove doriax` for Claude Code; for Codex, update `DORIAX_MCP_TOKEN` instead
The client cannot connect	The server only runs while the editor does, and only with **Enable Server** on. Start the editor, then reconnect from the client; in Claude Code, `/mcp` shows the server's status
Calls fail with *Changes are turned off*	**Allow Changes** is off. Turn it on, or ask the agent only for inspections
Calls fail with *The editor was busy*	A project load or a scene save took longer than two minutes. Try again once it finishes
`open_project` or `create_project` fails over unsaved work	The editor does not drop work to switch projects. Let the agent save it, or save or discard it yourself; see Opening and creating projects
`open_project` says the project is already open in another Doriax editor	A project opens in one editor at a time. Close it in that editor first, or connect the agent to that editor's server
A **Save Scene** dialog opens while an agent works	The agent saved a scene that has no file yet without a `path`. Answer the dialog, or ask the agent to pass `path` to `save_scene`
The agent still lists tools that **Allow Changes** removed, or misses ones it added	Clients keep the tool list they read when they connected. Reconnect the client

Protocol details

For client authors:

- **Transport:** Streamable HTTP on one endpoint, `POST /mcp`. Every response is a single JSON object, never an SSE stream. `GET` and `DELETE` return `405`.
- **Versions:** `2026-07-28` (stateless: `server/discover`, `_meta` on every request, and the mirrored `MCP-Protocol-Version`, `Mcp-Method` and `Mcp-Name` headers checked), plus the `initialize`-based `2025-11-25`, `2025-06-18` and `2025-03-26`. No sessions are created.

- **Capabilities:** tools only, with no resources, prompts or change notifications. The tool list only changes with **Allow Changes**, but that can happen at any time, so under `2026-07-28` the `server/discover` and `tools/list` results are marked never to be cached (`ttlMs: 0`, `cacheScope: "private"`).
- **Results:** text content holding the action's message followed by its data as JSON. A failed action is a result with `isError: true`, so the model can correct itself; an unknown tool or a malformed request is a JSON-RPC error.

Where the values are stored

File	Key	Holds
`settings.yaml`	`mcp_server`	`enabled`, `port` and `allow_changes`
`ai_keys.dat`	`mcp_server`	The token, obfuscated and bound to this machine

Both files are in the editor's settings folder, which the FAQ (p. 769) lists per platform.

See also

- Editor Settings (p. 71) — the MCP Server section, beside the AI Chat's keys
- Command-Line Tools (p. 167) — exports, shader builds and benchmarks without the editor window
- Model Context Protocol[7] — the specification

7. https://modelcontextprotocol.io

PART 4

Manual

Engine Architecture 180
Scenes & Entities 182
Entity Component System 190
Scripting 195
Creating Scripts 199
C++ Build Setup 209
Events 215
Script Properties 223
Input 229
Resources & Assets 238
2D Graphics 243
3D Graphics 248
Rendering Pipeline 258
Shader Variants 278
Shader Effects 282
Multiple Resolutions 288
User Interface 292
Animation 298
Particles 306
Terrain 309
Physics 315
Audio 326
Math & Utilities 329
Threading & Async Loading 333
Monetization 337

description: Runtime architecture of Doriax: Engine, Scene, ECS, systems, managers, rendering, physics, and scripting.

Engine Architecture

Doriax is organized around a small runtime core, an ECS, scene-scoped subsystems, and high-level object wrappers. The editor uses this runtime API, while the runtime itself has no dependency on editor internals.

The editor saves authoring data as `.scene` and `.bundle` files; export turns them into C++ factory functions built on the same API as handwritten game code. See From editor data to runtime code (p. 155) for the full boundary.

Runtime layers

Layer	Responsibility
`Engine`	Global runtime state: scenes, canvas, scaling, timing, platform, events, async loading
`Scene`	Entity registry, active camera, lighting state, UI events, and registered subsystems
ECS	Entity IDs, component arrays, signatures, and component lifecycle notifications
Subsystems	Rendering, mesh processing, physics, audio, actions, UI
Object wrappers	Friendly APIs such as `Sprite`, `Model`, `Camera`, `Light`, `Body2D`, `Button`
Managers	Scene and bundle factories for runtime loading
Platform/render backends	Window, input, graphics, filesystem, and target-specific integration

Engine singleton

`Engine` is a static API. It stores the main scene stack, canvas size, scaling mode, timing values, platform/backend information, event subscriptions, and async-loading state.

Common responsibilities:

- Set and layer scenes.
- Configure canvas size and scaling.
- Track delta time, framerate, draw-call / triangle stats, and physics interpolation.
- Forward input and lifecycle events.
- Manage render-to-texture framebuffers.
- Coordinate async loading and queued work committed to the main thread.

Scenes

`Scene` extends `EntityRegistry`. It owns entities and components, then registers systems that process those components. Component add/remove notifications are forwarded to every registered subsystem so render, physics, audio, and UI state can stay in sync.

Systems

Systems are scene-scoped, not global. This keeps scene state isolated and lets editor play mode snapshot and restore scene data safely.

System	Runtime role
`RenderSystem`	Draws cameras, meshes, sprites, lights, UI, sky, fog, particles, lines, and points
`MeshSystem`	Generates and loads mesh data, including GLTF, OBJ, terrain, tilemaps, sprites, instancing, and mesh detail levels
`PhysicsSystem`	Steps 2D/3D physics, contacts, sensors, joints, filters, and body transforms
`AudioSystem`	Plays, pauses, seeks, and spatializes sounds through SoLoud
`ActionSystem`	Updates actions, animation playback, sprite animation, and keyframe tracks
`UISystem`	Handles UI layout, events, text input, and UI rendering support

Data flow

1. User or editor code creates entities and components.
2. The scene registry stores component data and updates entity signatures.
3. Subsystems receive lifecycle notifications.
4. Every frame, `Engine` updates scenes and systems.
5. Fixed updates run physics-sensitive behavior.
6. Render systems draw visible scene layers.

Managers

`SceneManager` and `BundleManager` register factory functions. Use them when runtime code needs to load scenes or instantiate prefab-like entity hierarchies by ID or name.

Scripting

Lua and C++ both drive the same runtime. Lua bindings expose engine classes, object wrappers, components, math types, actions, I/O helpers, and enums. C++ code includes `Doriax.h` and registers startup code with `DORIAX_INIT`.

Scenes & Entities

A **scene** is the container for everything that exists in a part of your game: entities, cameras, lights, physics, scripts, and the systems that update them each frame. A game is typically made of one or more scenes that you create, configure, and switch between at runtime.

Scenes

A scene owns a collection of entities and drives their updates. Internally, `Scene` extends `EntityRegistry`, so it owns entity allocation, component storage, and subsystem registration. You create a scene, populate it with entities, and tell the engine to run it.

Lua

```lua
scene = Scene()
-- ... add entities ...
Engine.setScene(scene)
```

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Scene scene;
// ... add entities ...
Engine::setScene(&scene);
```

You can have several scenes loaded at the same time. The most common pattern is one main scene for gameplay with additional scene layers for overlays, menus, or transition effects.

Operation	C++ API
Set the main scene	`Engine::setScene(scene)`
Add an overlay/layer	`Engine::addSceneLayer(scene)`
Execute once and discard	`Engine::executeSceneOnce(scene)`
Remove one layer	`Engine::removeScene(scene)`
Clear all scenes	`Engine::removeAllScenes()`

For project-level scene switching, use `SceneManager` to register scene factories and load scenes by name or numeric ID:

C++

```cpp
SceneManager::registerScene(1, "Level1", []() {
    static Scene level;
    Engine::setScene(&level);
});

SceneManager::loadScene("Level1");
```

That form registers a stack that can be **loaded**. To also overlay it with `addChildScene`, pass a second factory that only creates the scenes — see registerScene (p. 602). Scenes registered by the editor always get both.

See Switching scenes with SceneManager for full scene transitions and on-demand overlays at runtime.

Scene types

All scenes use the same `Scene` class and ECS foundation at runtime. The **scene type** is an authoring concept that tells the editor which defaults, camera setup, and tools to use.

Type	Editor defaults	Typical use
3D	Perspective camera, 3D gizmos, lighting setup	Open-world levels, first/third-person games, 3D platformers
2D	Orthographic camera, canvas gizmos, 2D tools	Top-down games, side-scrollers, puzzle games, tilemaps
UI	Screen-space canvas, anchor gizmos	Menus, HUDs, settings screens, overlays

A 3D scene can contain UI entities, and a UI scene can reference 2D or 3D child scenes. This compositing model is the foundation for in-game HUDs: create a separate **UI scene** and add it as a scene layer on top of the gameplay scene.

```
-- Load a HUD scene on top of the main gameplay scene
Engine.setScene(gameplayScene)
Engine.addSceneLayer(hudScene)
```

Entities

An **entity** is a lightweight integer identifier that lives in a scene. By itself it holds no behavior or data; it gains both by having **components** attached to it. This is the core of the Entity Component System (p. 190).

Higher-level objects in Doriax — `Sprite`, `Model`, `Camera`, `Button`, etc. — are convenience wrappers. They create or reference an entity and manipulate the components it needs for you.

```
scene = Scene()
triangle = Polygon(scene)
triangle.position = Vector3(300, 300, 0)
triangle:setColor(0.6, 0.2, 0.6, 1)
```

Transforms

Most visible entities have a **Transform** component that defines their **position**, **rotation**, and **scale** in space. Transforms can be parented to build hierarchies — moving a parent moves all its children with it.

```
entity.position = Vector3(300, 300, 0)
entity.rotation = Quaternion(0, 0, 0)
entity.scale    = Vector3(1, 1, 1)
```

When an entity has a parent, its world transform is derived from the full parent chain. Local transforms are relative to the parent; world transforms are absolute.

Entities without transforms

Not every entity needs a Transform. Non-transform entities are useful for:

- Logic-only scripts and global state
- Non-spatial (2D panned) audio
- Scene environment components like Fog or Skybox
- Physics joints that link two body entities
- Actions and animations targeting another entity

In the editor's Structure Panel (p. 73), these entities appear in a separate area above the transform hierarchy. They can be selected and edited but cannot be parented until a Transform is added.

Scene settings

Every scene owns rendering and interaction state that affects all entities within it:

Setting	Purpose
Active camera	The `Camera` entity used for rendering
Background color	Clear color for the scene
Shadow quality	PCF filtering level for smoother shadow edges (`NONE` to `HIGH`)
Light state	`OFF`, `ON`, or `AUTO` (auto-detects lights in the scene)
Global illumination	Ambient light color and intensity
Ambient occlusion (SSAO)	Screen-space ambient occlusion — enable plus radius, intensity, and bias — see Rendering Pipeline (p. 265)
Reflections (SSR)	Screen-space reflections — enable plus distance, thickness, intensity, and glossy blur — see Rendering Pipeline (p. 266)
Ambient light (2D)	Flat ambient color and intensity for 2D scenes, with its own 2D shadow quality — see 2D lighting (p. 244)
Physics gravity	Per-world gravity in m/s^2 — 2D (Box2D) and 3D (Jolt) are independent — see Physics (p. 316)
Fixed resolution	Render the main scene at a fixed internal size and upscale it to the window — see Multiple Resolutions (p. 289)
UI events	Enables/disables UI input processing for the scene
Default shaders	Per-type custom shaders (Mesh, UI, Sky, Points, Lines) used by every component of that type without its own custom shader — see Custom Shaders (p. 146)
Canvas size	Logical resolution for 2D and UI scenes

Edit these in the editor by selecting the scene root in the Structure panel (p. 73) with no entity selected; the settings appear in the Properties window (p. 90). Which sections are shown depends on the scene type — 2D and UI scenes hide 3D-only options such as SSAO and SSR. As with component fields, any setting changed from its default shows a small **reset arrow** next to its label that restores the default, and every change is saved with the scene and undoable.

Scene stacks

When the engine runs, every loaded scene lives in an ordered list of **layers**, and a group of scenes that are loaded together is called a **scene stack**:

- The **main scene** is always the bottom layer. There is exactly one per stack.
- Each additional scene is a **layer** drawn on top of the layers before it.
- The **last** layer is the topmost — it renders over everything below it.

In the editor, each scene file is the main scene of its own stack, and you attach other scenes to it as **child scenes**. At export, the editor turns each stack into a factory that calls `Engine::setScene()` for the main scene and `Engine::addSceneLayer()` for each child, then registers it with `SceneManager`.

Child scenes

A **child scene** reference lets one scene load and run another as part of the same stack. Child scenes share the same engine tick but keep their own entity and component storage, so they stay cleanly separated. Use them for:

- **Persistent UI** — a HUD or menu scene that stays loaded across gameplay level changes.
- **Shared setup** — a single lighting and skybox scene included in every level.
- **Additive worlds** — large open worlds assembled from independently-authored chunks. Bodies in different chunks don't collide (see the note below).
- **Scene streaming** — load and unload sections without a full scene transition.
- **Pop-ups and pause menus** — UI that appears on demand and is removed without reloading the level.

Each scene has its own physics world

Every scene runs its own 2D and 3D physics world. A body in one scene never collides with a body in another scene. A ray cast only tests the scene you pass it, and a joint connects only bodies of its own scene. Keep everything that must touch physically in one scene, and use child scenes for parts that don't, such as a HUD, lighting or menus. If chunks of an additive world have to interact, see Physics across scenes (p. 323).

Entity IDs are also numbered per scene. That is why the editor only lets you reference a child scene's entity from a script property (p. 105).

Adding a child scene in the editor

Select the parent scene in the Structure panel (p. 73) and either:

- right-click the scene root and choose **Add child scene → *SceneName***, or
- drag a `.scene` file from the Resources Browser (p. 106) onto the scene root.

A child scene can have child scenes of its own, but a scene can't contain itself, directly or through other child scenes. The editor refuses an add that would create such a cycle.

Child scenes are listed **above** the parent's entities. Click the eye icon next to a child scene to load it *inline*, so you can see and edit its entities in context — this is an editor convenience and does not change runtime behavior.

Start active

Each child scene reference has a **Start active** flag (right-click the child scene node → **Start active**). It controls whether the child scene is added to the engine automatically when its parent loads:

Start active	Behavior when the parent loads	Use it for
On (default)	The child scene is created **and** added as a layer. It runs and renders immediately.	HUDs, persistent overlays, shared lighting — anything always present.
Off	The child scene is still created and prepared (its entities, scripts, and `Scene*` are all ready), but it is **not** added to the engine.	Pause menus, dialogs, level sections — anything you reveal later.

A child scene that starts inactive is *ready but hidden*. Because it is already built, showing it later with `SceneManager.addChildScene` is instant — there is no loading cost at the moment it appears.

Start active is not what makes a scene addable at runtime; any registered scene can be overlaid, and one that is not a child scene is simply built the first time you add it. What the flag controls is whether the scene comes up *with its parent*, and inactive children are the ones already paid for. See Preloaded or built on demand.

Scene order

The order of the layers follows the order the child scenes are listed under their parent in the Structure panel, which is the order you added them. The main scene stays at the bottom; each child scene draws on top of the ones above it in the list. **Put a HUD or menu scene last so it renders over the gameplay below it.**

To change the order, remove the child scenes and add them back in the sequence you want — the one added last ends up on top. At runtime, `SceneManager.addChildScene` always adds a scene **on top** of the current layers, so re-adding is also how you bring a layer to the front.

Switching scenes with SceneManager

`SceneManager` (p. 601) is the runtime entry point for moving between scenes. The editor registers every scene stack for you at startup, so from a script you refer to scenes by their **name** (or numeric ID) and never touch the factory functions directly. There are two ways to change what is on screen.

Migrate to another scene

`SceneManager.loadScene` performs a full **transition**: it removes every currently loaded scene (`Engine::removeAllScenes()` — main *and* all layers), then builds the requested stack from scratch and runs it. Use it to move between levels, menus, and game-over screens.

Lua

```lua
-- Leave the current scene entirely and load another
SceneManager.loadScene("Level2")

-- Or by registration ID
SceneManager.loadScene(2)
```

C++

```cpp
SceneManager::loadScene("Level2");
SceneManager::loadScene(2);
```

> **One stack at a time**
> `loadScene` clears everything first, so you do **not** call it once per layer. Attach persistent layers (a HUD, shared lighting) as **start-active child scenes** of the target scene instead — they come up with it in a single `loadScene` call.

> **Called from a running scene**
> From a script, a physics contact or an input event the transition happens at the start of the **next frame**, so the calling scene and its scripts finish what they are doing intact; `SceneManager::isLoadPending()` is `true` until then. See SceneManager (p. 603).

Show a loading screen

Building a large scene takes a moment. Pick a UI scene as the **Loading Scene** in Project Settings → General → Loading (p. 56) and `loadScene` shows it over the old scene, replaces the old scene behind it, and removes it once the new scene has loaded. From code:

Lua

```lua
SceneManager.setLoadingScene("Loading")
SceneManager.loadingDelay = 0.3
```

C++

```cpp
SceneManager::setLoadingScene("Loading");
SceneManager::setLoadingDelay(0.3f);
```

`SceneManager.loading` and `SceneManager.loadingProgress` drive its animation. Turn on async loading (p. 335) so it keeps animating while textures and sounds stream in. A level starts its gameplay in `Engine.onSceneLoaded` (p. 373), which fires once it has loaded, still behind the loading scene.

Overlay a scene without leaving the current one

`SceneManager.addChildScene` and `removeChildScene` add or remove a scene **on top** of what is already running, with no transition. This is how you show a pause menu, dialog, or HUD toggle over live gameplay. Any registered scene can be overlaid — it does not have to belong to the loaded stack. If its scenes have not been built yet, `addChildScene` builds them before showing them.

Lua

```lua
-- Open a pause menu over the running game
function onPause()
    if SceneManager.addChildScene("PauseMenu") then
        Engine.pauseGameEvents(true)   -- freeze gameplay updates
    end
end

-- "Resume" button handler inside the PauseMenu scene
resumeBtn = Button(scene)
resumeBtn.label = "Resume"
local btnComp = resumeBtn:getButtonComponent()
btnComp.onPress = function()
    SceneManager.removeChildScene("PauseMenu")
    Engine.pauseGameEvents(false)
end
```

C++

```cpp
// Open a pause menu over the running game
if (SceneManager::addChildScene("PauseMenu")) {
    Engine::pauseGameEvents(true);
}

// "Resume" button handler inside the PauseMenu scene
resumeBtn.getComponent<ButtonComponent>().onPress = []() {
    SceneManager::removeChildScene("PauseMenu");
    Engine::pauseGameEvents(false);
};
```

Because `addChildScene` puts the scene at the top of the layer list, the overlay always draws above the gameplay below it. `removeChildScene` takes it back out and leaves the main scene and other layers untouched — it does not destroy the scene, so adding it again brings it back exactly as you left it, scroll position and all.

`addChildScene` is also safe from a script's `init` while that script's own stack is still loading — a level that opens its HUD from code. The stack it adds survives the load; only scenes no running stack involves are torn down by the transition.

Preloaded or built on demand

Because `addChildScene` can build a stack itself, you choose when a scene pays for its own setup:

How the scene is set up	Built	First time it is shown
Child scene, **Start active → On**	With its parent	Already on screen
Child scene, **Start active → Off**	With its parent	Instant — it is ready and waiting
Not a child scene at all	On the first `addChildScene`	Pauses to create entities and start scripts

Keep something you toggle often — a pause menu, a HUD panel — as an inactive child, so showing it costs nothing. Leave a heavy screen you may never open — a full inventory, a map, an end-of-game summary — out of the child list, so a session that never opens it never pays for it.

> **Cross-scene references need the scene to exist**
>
> If a script in one scene references entities in another (see SceneManager.setScenePtr (p. 608)), the referenced scene must be built when that script initializes. Keep it a child scene — active or inactive — so it comes up with its parent. A scene built on demand does not exist yet when the parent's scripts start.

Knowing where you are

`SceneManager` also reports the current state, which is handy for transition logic and save systems:

Query	Returns
`SceneManager.getCurrentSceneName()`	Name of the most recently loaded scene
`SceneManager.getCurrentSceneId()`	ID of the most recently loaded scene
`SceneManager.getSceneNames()`	All registered scene names
`SceneManager.getSceneId(name)`	Numeric ID for a name (`0` if unknown)
`SceneManager.loading`	`true` from `loadScene` until the new scene has loaded and its loading scene has closed

See the SceneManager reference (p. 601) for the complete API.

Creating UIs as scenes

User interfaces fit the child-scene model naturally. The recommended pattern is to author each interface — HUD, main menu, pause menu, settings — as its own **UI scene**, then bring it on screen as a layer:

1. Create a **UI scene** (p. 292) and add widgets to it (`Panel` (p. 518), `Button` (p. 506), `Text` (p. 528), and so on).
2. Add it as a **child scene** of the gameplay scene it belongs to.
3. Choose its **Start active** flag:
   - **On** for a permanent HUD that should appear with the level.
   - **Off** for a menu or dialog you open later with `SceneManager.addChildScene`.
4. Keep the UI scene **last** in the child list so its widgets draw above the game.

Because a UI scene is screen-space and self-contained, you can swap menus without touching gameplay data and load or unload interfaces independently. For widget construction, anchoring, events, and canvas scaling, see the User Interface (p. 292) manual and the First UI Scene (p. 35) tutorial.

Bundles

Bundles are reusable entity hierarchy templates — similar to prefabs in other engines. They let you define a configured group of entities once and place the same hierarchy in any number of scenes.

How bundles work

A bundle is authored visually in the editor (or defined in script) and saved as a YAML file. During export, the bundle is converted into a **factory function** registered with `BundleManager`. At runtime, calling `createBundle` runs the factory and returns the root entity of the new hierarchy.

Unlike simple duplication, bundles:

- Are a **single source of truth** — editing the bundle definition updates every scene that references it.
- Are **shared across scenes** — the same bundle can be instantiated in many scenes without duplicating authored data.
- **Can be created and destroyed at runtime** — useful for enemies, projectiles, UI cards, and any pooled or on-demand hierarchy.

Creating a bundle in the editor

1. Build the entity hierarchy you want to reuse.
2. In the Resources Browser, open the folder the bundle should be saved to.
3. Right-click the root entity in the Structure panel and choose **Save as Bundle** (dragging the entity onto the Resources Browser does the same thing).

The editor writes `<EntityName>.bundle` into that folder and replaces the hierarchy in the scene with an instance of it; rename the file in the Resources Browser to change the bundle name. Bundles can live in any project folder — `bundles/` is just a convention. See Bundles (p. 111) for the full editor workflow.

Using a bundle at runtime

Bundle **registration** is generated automatically by the export step (it emits a C++ `registerBundle` call that wires the factory to the bundle name/ID). At runtime you mainly call `createBundle` and `destroyBundle`. Note that `createBundle` takes the bundle **name first, then the scene**, and returns the root `Entity`. Entity IDs are scene-local: to parent the instance under an existing entity, look it up in the destination scene with `createBundle(name, scene, "parentName")`, or pass an object that already carries its scene. Every call creates its own root, so a bundle can be spawned as many times as you want. Member entities are created when the factory runs, so their ids differ between the editor and a build; reach them by name under the instance root with `findEntity(name, root)` (p. 379) rather than by id.

A spawned instance runs its scripts like the scene does: Lua `init()` and C++ constructors run before `createBundle` returns. `destroyBundle` stops them, then removes the instance along with anything parented under it since, bundles spawned there included. A C++ script that destroys its own bundle is deleted during that call, so it must return without touching its members.

> **There is no direct `.bundle` file loader**
>
> You don't load a `.bundle` file by path at runtime. Export turns each file into a registered factory, and you spawn it by the **bundle name** — the file's path with the `.bundle` extension removed and forward slashes (so `bundles/enemies/EnemyShip.bundle` is created as `"enemies/EnemyShip"`). Only the bundles a scene instantiates and the ones listed in **Project → Bundles** are registered, so a bundle spawned from script alone has to be listed there. Use `BundleManager.getBundleNames()` to list everything that is registered.

Lua

```lua
-- Create an instance (name first, then scene)
local root = BundleManager.createBundle("enemy_grunt", scene)

-- Attach under a named entity in that scene
BundleManager.createBundle("enemy_grunt", scene, "spawn")

-- Destroy it later (scene first, then root entity)
BundleManager.destroyBundle(scene, root)
```

C++

```cpp
// Registration is normally emitted by export, e.g.:
BundleManager::registerBundle(1, "enemy_grunt",
    [](Scene* scene, Entity parent) -> bool {
        // factory body
        return true;
    });

Entity root = BundleManager::createBundle("enemy_grunt", &scene);
BundleManager::createBundle("enemy_grunt", &scene, "spawn");

// Destroy when done
BundleManager::destroyBundle(&scene, root);
```

See BundleManager (p. 597) for the full API reference.

Next steps

- Entity Component System (p. 190) — how data and behavior are organized under the hood.
- Scripting (p. 195) — add gameplay logic to entities.
- Project Workflow (p. 48) — how to manage scenes and bundles in the editor.

Entity Component System

Doriax is built on an Entity Component System. This is not only an internal implementation detail; it is the mental model used by the editor, the runtime, and the API.

Entity

An entity is only an ID:

```
using Entity = unsigned;
#define NULL_ENTITY 0
```

The ID does not own position, rotation, rendering, physics, scripts, audio, hierarchy, or children. It is just a key used by the scene registry to find component data.

Component

A component is data attached to an entity. An entity behaves according to the components it has.

```
Entity player = scene.createUserEntity();

scene.addComponent<Transform>(player);
scene.addComponent<MeshComponent>(player);
scene.addComponent<Body2DComponent>(player);
scene.addComponent<ScriptComponent>(player);
```

This entity is now spatial, renderable, physical, and scriptable because those components exist. Remove one component and that part of the behavior goes away.

The one-argument form default-constructs the component directly in ECS storage. Prefer it over passing `{}`, especially for components with large fixed-capacity data. To attach an already configured component, pass the value instead; its type can be deduced:

```
Transform transform;
transform.position = Vector3(0.0f, 2.0f, 0.0f);
scene.addComponent(player, transform);
```

System

A system processes entities with the components it understands. `RenderSystem` cares about renderable components, `PhysicsSystem` cares about body and joint components, `AudioSystem` cares about sound components, and so on.

EntityRegistry and Scene

`Scene` inherits from `EntityRegistry`. The registry creates entities, stores component arrays, tracks each entity signature, and manages the transform hierarchy.

Important registry methods:

Method	Purpose
`createEntity()`	Create from the scene's default entity pool
`createUserEntity()`	Create a user entity, starting at `EntityManager::firstUserEntity()`
`createSystemEntity()`	Create a system entity from the reserved system range
`destroyEntity(entity)`	Destroy an entity and remove its components

Method	Purpose
`addComponent<T>(entity)`	Default-construct a component directly in ECS storage
`addComponent(entity, value)`	Attach configured component data; the type is deduced
`removeComponent<T>(entity)`	Remove component data
`findComponent<T>(entity)`	Return a nullable component pointer
`getComponent<T>(entity)`	Return a component reference
`getSignature(entity)`	Return the component bitset for the entity

`EntityManager` reserves IDs `1..1000` for system entities and starts user entities at `1001`. `NULL_ENTITY` is `0`.

Transform and hierarchy

The hierarchy is not owned by the entity ID. It is built from the `Transform` component:

```
struct Transform {
    Vector3 position;
    Quaternion rotation;
    Vector3 scale;
    Entity parent;
    bool visible;
    // world transforms and matrices are computed from the parent chain
};
```

Only entities with `Transform` can live in the spatial hierarchy. In the editor's Structure panel, entities without `Transform` are shown in a separate non-hierarchical area. Add `Transform` if an entity should be positioned or parented.

Hierarchy operations live on `EntityRegistry` and object wrappers:

API	Meaning
`addEntityChild(parent, child, changeTransform)`	Parent one transform entity under another
`moveChildToTop/Up/Down/Bottom(entity)`	Change sibling order
`findOldestParent(entity)`	Find the root of a transform chain
`isParentOf(parent, child)`	Test ancestry
`Object::addChild(entity)`	Wrapper convenience for parenting

Removing `Transform` from an entity removes it from this hierarchy. Its script, sound, action, or other non-spatial components can still exist, but it no longer has local or world transform data.

Wrapper classes

Object classes such as `Sprite`, `Model`, `Camera`, `Button`, and `Sound` are wrapper APIs around entities and components. They make gameplay code easier to write, but they do not change the ECS rule: the entity is still an ID, and the data lives in components.

```
Sprite sprite(&scene);
sprite.setSize(64, 64);
sprite.setPosition(100, 100);
```

This wrapper creates or references an entity and manipulates the components needed to represent a sprite.

Writing components directly

A wrapper setter usually does two things: it writes the field, and it marks the component dirty so the system picks the change up on the next frame.

```
void Sound::setVolume(double volume){
    SoundComponent& audio = getComponent<SoundComponent>();
    if (audio.volume != volume){
        audio.volume = volume;
        audio.needUpdate = true;   // without this, AudioSystem never applies it
    }
}
```

Writing the component yourself means doing both, for the fields a flag guards. Flags exist where the work is expensive — rebuilding geometry, pushing a voice's parameters to the audio backend, refreshing a button or a tilemap — not for every field. Plenty are read as written: `ActionComponent.speed` is used on every tick, `UIComponent.color` is uploaded on every draw, and `ParticlesComponent` has no flag at all. When in doubt, look at what the wrapper setter does.

Lua

```
local audio = sound:getSoundComponent()
audio.volume = 0.25
audio.needUpdate = true
```

C++

```
SoundComponent& audio = scene.getComponent<SoundComponent>(entity);
audio.volume = 0.25;
audio.needUpdate = true;
```

The flags are named after what they refresh: `needUpdate`, `needUpdateSizes`, `needUpdateButton`, `needUpdateTilemap`, `needReload`, and so on.

> **Changing a sprite's mesh directly**
>
> A sprite that nothing changes is drawn from a copy of its mesh that `RenderSystem` keeps (see Sprite batching (p. 273)). After writing a sprite's `MeshComponent` yourself in C++, set `needUpdateBatch` on its `SpriteComponent`, or the sprite keeps drawing its old texture region, color, or texture:
>
> ```
> MeshComponent& mesh = scene.getComponent<MeshComponent>(entity);
> mesh.submeshes[0].material.baseColorFactor = Vector4(1, 0, 0, 1);
> scene.getComponent<SpriteComponent>(entity).needUpdateBatch = true;
> ```
>
> Wrapper setters such as `Mesh::setColor` and `Sprite::setTextureRect` set it for you, and reading a component never counts as a change. Like the other flags, it is picked up on the next update.

Two kinds of field are read-only from Lua, because writing them can only desync the system that owns them: values a system produces (`playingTime`, `state`, `pressed`, `loaded`) and the child-entity handles of a widget (`label`, `bar`, `fill`, and the `TextEdit` `text` / `selection` / `cursor` entities).

> **Colours are linear in components**
>
> A wrapper setter such as `Image::setColor` or `Button::setColorNormal` converts from sRGB; the component field stores the linear value. Writing `color` directly stores exactly what you pass, so prefer the setter unless you already have linear values.

Reaching a component from Lua

Lua has an accessor for each component in the reference, on the wrapper that adds it: `object:getTransform()`, `camera:getCameraComponent()`, `text:getTextComponent()`. Each page under Components (p. 352) names its accessor. On an entity without that component the call raises an error, except `mesh:getInstancedMeshComponent()`, which returns `nil` until the mesh is instanced.

Field writes land on the component, but lists such as `points` or `times` come to Lua as copies: change the table, then assign it back.

```
local tracks = KeyframeTracks(self.scene, self.entity):getKeyframeTracksComponent()
local times = tracks.times
times[#times] = 2.0
tracks.times = times
```

> **Get the component where you use it**
> Components of one kind are packed in an array that moves when entities or components are added or removed, so a component kept in a script field can end up pointing at another entity's data. Call the accessor again instead, as you would re-fetch a `Component&` in C++. A component taken from a temporary such as `Object(scene):getTransform()` goes stale as soon as Lua collects that object, since it destroys its entity.

Built-in component groups

The engine includes a broad set of built-in components. Most are exposed through both editor inspectors and high-level object wrappers.

Group	Components
Core	`Transform`, `ScriptComponent`, `BundleComponent`
2D	`SpriteComponent`, `SpriteAnimationComponent`, `TilemapComponent`, `PolygonComponent`
3D	`MeshComponent`, `ModelComponent`, `InstancedMeshComponent`, `TerrainComponent`, `BoneComponent`, `SkinComponent`
Rendering	`CameraComponent`, `LightComponent`, `FogComponent`, `SkyComponent`, `WaterComponent`, `WaterExclusionComponent`, `ParticlesComponent`, `LinesComponent`, `PointsComponent`
Physics	`Body2DComponent`, `Body3DComponent`, `Joint2DComponent`, `Joint3DComponent`
UI	`UIComponent`, `UIContainerComponent`, `UILayoutComponent`, `ButtonComponent`, `TextComponent`, `ImageComponent`, `PanelComponent`, `ScrollbarComponent`, `ProgressbarComponent`, `TextEditComponent`
Animation/ actions	`ActionComponent`, `TimedActionComponent`, `PositionActionComponent`, `RotationActionComponent`, `ScaleActionComponent`, `ColorActionComponent`, `AlphaActionComponent`, keyframe track components
Audio	`SoundComponent`

Scene systems

Scenes register subsystem instances that react when relevant components are added or removed. The major runtime systems are:

System	Handles
`RenderSystem`	Mesh, sprite, UI, camera, light, sky, fog, line, point, and particle drawing
`MeshSystem`	Procedural meshes, model loading, sprite/tilemap geometry, terrain, and instancing
`PhysicsSystem`	Box2D 2D bodies, Jolt 3D bodies, joints, contacts, and collision filtering
`AudioSystem`	SoLoud-backed sound playback and 3D audio positioning
`ActionSystem`	Time-based actions, sprite animation, skeletal animation, and keyframe tracks
`UISystem`	UI layout, button interaction, text editing, and event dispatch

Object wrappers

The object classes in `engine/core/object/` are convenience wrappers around entities. For example, `Sprite` owns an entity and exposes sprite-specific methods, while still using the same ECS data as editor-created entities.

Use wrappers when writing gameplay code. Drop to raw ECS access when you are building engine extensions, custom tools, or systems that need to process many entities at once.

Next steps

See Creating Scripts (p. 199) to learn how Lua and C++ drive behavior on top of entities and components.

Scripting

Doriax supports two scripting languages: **Lua** for fast iteration and flexible scripting, and **C++** for full native performance. You can use either one or combine both in the same project.

Lua vs. C++

	Lua	C++
Iteration speed	Very fast — no recompile needed	Requires a build step
Runtime performance	High	Maximum (native)
Compilation	Interpreted at runtime	Compiled at **build time**
Best for	Gameplay logic, prototyping, glue code	Performance-critical systems

A common workflow is to prototype in Lua, then move hot paths to C++ when you need the extra performance.

Editor-era scripting model

Current Doriax projects usually attach behavior to entities with `ScriptComponent`. Each `ScriptComponent` can contain multiple script entries. A script entry stores the script type, paths, class/module name, enabled state, Properties window fields, and runtime instance pointer.

Script type	Use it for
`SUBCLASS`	C++ class derived from the entity's object wrapper (`Object`, `Mesh`, `Camera`, `Terrain`, ...)
`SCRIPT_CLASS`	C++ class derived from `ScriptBase`
`SCRIPT_LUA`	Lua module returning a script table

The difference between the two C++ types trips people up: a **Subclass** *is* the object (it inherits `Camera` / `Mesh` / `Object` /... and you call wrapper methods on `this`), while a **Script Class** only *references* the entity through `ScriptBase`. See C++ Subclass vs. C++ Script Class (p. 201) for the full breakdown and player-controller guidance.

See Creating Scripts (p. 199), Events (p. 215), and Script Properties (p. 223) for the full workflow.

Entry points

Each project has entry points for both languages:

- **C++** — the engine calls your `init()` function when the game starts.
- **Lua** — the main Lua script runs at startup and can require other Lua files.

Lua

```lua
-- main.lua
scene = Scene()
-- build your scene here
Engine.setScene(scene)
```

C++

```cpp
// main.cpp
#include "Doriax.h"
using namespace doriax;

Scene scene;

DORIAX_INIT void init() {
    // build your scene here
    Engine::setScene(&scene);
}
```

Choosing one entry point

If both Lua and C++ call `setScene()`, the C++ call runs last and overrides the Lua one. To disable an entry point, build with `NO_CPP_INIT` or `NO_LUA_INIT`:

- **CMake:** add `-DNO_CPP_INIT=1` (or `-DNO_LUA_INIT=1`) to your configure step.

See Export Window (p. 155) for full details.

If the main Lua script fails, the error is logged and the game keeps running with whatever the script set up before the error. The editor's Play mode does not run `main.lua`, so check it in an exported build.

For editor-authored projects, the start scene and exported scene factories are usually more important than hand-written startup code. Use entry points for engine-level setup, not for recreating an entire editor-authored scene by hand.

The Engine API

The static `Engine` object is your main entry into the runtime. Common calls include:

```lua
Engine.setCanvasSize(1000, 480)   -- set the rendering canvas size
Engine.setScene(scene)            -- set the active scene
```

The same methods are available in C++ through `Engine::` static methods.

Lua bindings

Doriax uses LuaBridge to expose the engine API to Lua. The binding layer registers:

Binding group	Examples
Core	`Engine`, `Scene`, `SceneManager`, `BundleManager`, `Input`, `System`
ECS	Entities and built-in component types
Objects	`Object`, `Sprite`, `Mesh`, `Model`, `Camera`, `Light`, `Light2D`, `Occluder2D`, `Lines`, `Points`, `Sound`
Actions	`Action`, `Animation`, `SpriteAnimation`, `Particles`, `Ease`
Math	`Vector2`, `Vector3`, `Vector4`, `Matrix3`, `Matrix4`, `Quaternion`, `Ray`, `AABB`, `Rect`, `Plane`
I/O	`File`, `UserSettings`, memory data helpers

Most C++ enums are mirrored into Lua namespaces, such as `Scaling.FITWIDTH`, `BodyType.DYNAMIC`, and `GraphicBackend.METAL`.

When a function has several overloads, the argument types pick the one to call, and a value of the wrong type raises an error instead of being converted. For example, `mesh:setFaceCulling(nil)` fails, so pass `false`.

Lua version

Doriax embeds Lua 5.5. A few behaviors differ from older Lua versions and from most tutorials:

- **Floats print every digit needed to read them back.** `0.1 + 0.2` turns into `0.30000000000000004` in a string, so use `string.format("%.2f", value)` for anything players see.
- **The `for` loop variable is read-only.** Assigning to it is a compile error. Declare a local copy inside the loop instead.
- **`#` is only reliable on lists without holes.** After `t[3] = nil` on a five-item list, `#t` can be `2`. Use `table.remove(t, 3)` to delete from a list, and `pairs` to walk a table with gaps.
- **`collectgarbage("setpause")` and `"setstepmul"` are gone.** Use `collectgarbage("param", "pause", 200)` instead.
- **`global` declarations are strict.** After `global score` in a file, every other global it uses has to be declared too, so `Log` or `Engine` stop with `variable 'Log' not declared`. Start the file with `global *` to keep them. `global score = 0` fails with `global 'score' already defined` the second time the file runs, and the editor runs a script file when it reads its properties and again on each Play. Declare the name and assign it apart: `global score` then `score = 0`.

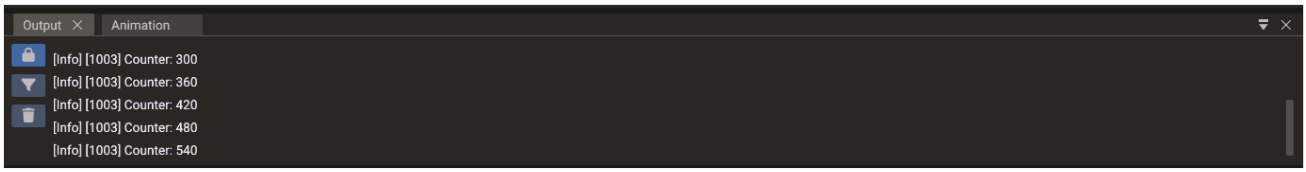
Script components

Attach `ScriptComponent` to an entity when behavior should live with that entity. The component supports C++ subclasses, C++ `ScriptBase` instances, and Lua script modules. Use script components for reusable entity behavior such as player controllers, enemy AI, interactive doors, or UI widgets.

Exposing values to the editor

C++ script properties are parsed by the editor and displayed in the Properties window when they use the supported property annotations. This lets designers tune values without editing source code. Keep exposed properties small and serializable: numbers, booleans, strings, colors, vectors, resource paths, and entity references are the best fit.

Logging



Use `Log` for runtime diagnostics in C++:

```
Log::print("Player spawned");
Log::warn("Missing optional texture: %s", path.c_str());
Log::error("Failed to load level");
```

The same logging functions are exposed to Lua through the `Log` class. Regular Lua `print()` also works for quick diagnostics:

```
Log.print("Player spawned")
Log.warn("Missing optional texture")
Log.error("Failed to load level")
```

require() and module paths

Lua `require("foo")` searches the virtual filesystem:

1. `lua://lua/foo.lua`
2. `lua://foo.lua`

Dots in module names map to directory separators (`require("ui.menu")` → `lua/ui/menu.lua`). The lookup goes through `Data`, so packed Lua modules in a native export resolve the same way as loose files.

Script component paths use the same `lua://` prefix with the stored relative path.

Global helpers

Function	Purpose
`RegisterEvent(self, event, methodName [, tag])`	Subscribe to any `FunctionSubscribe` event
`RegisterEngineEvent(self, methodName [, tag])`	Subscribe to `Engine.onUpdate`, etc.

Reference

- Creating Scripts (p. 199) — editor workflow and lifecycle
- C++ Build Setup (p. 209) — compilers, CMake, and build troubleshooting
- Events (p. 215) — all event types and registration macros
- Script Properties (p. 223) — `DPROPERTY` and Lua properties
- API Index (p. 352) — complete class reference (C++ and Lua)

Creating Scripts

Scripts attach behavior to entities through `ScriptComponent`. A script does not own the entity. It receives the entity ID and scene pointer, then reads or changes the components attached to that entity.

This page covers the full workflow from the Doriax editor through runtime initialization, export, and cleanup.

Mental model

```
Entity
  └── ScriptComponent
        └── scripts[]: ScriptEntry
              ├── type        (SUBCLASS | SCRIPT_CLASS | SCRIPT_LUA)
              ├── path        (.lua or .cpp)
              ├── headerPath  (.h for C++; empty for Lua)
              ├── className   (C++ class or Lua module name)
              ├── enabled
              ├── properties[] (inspector values)
              └── instance    (C++ pointer or Lua registry ref)
```

One `ScriptComponent` holds **multiple** script entries. Each entry is independent and can be enabled or disabled separately.

Create a script in the editor

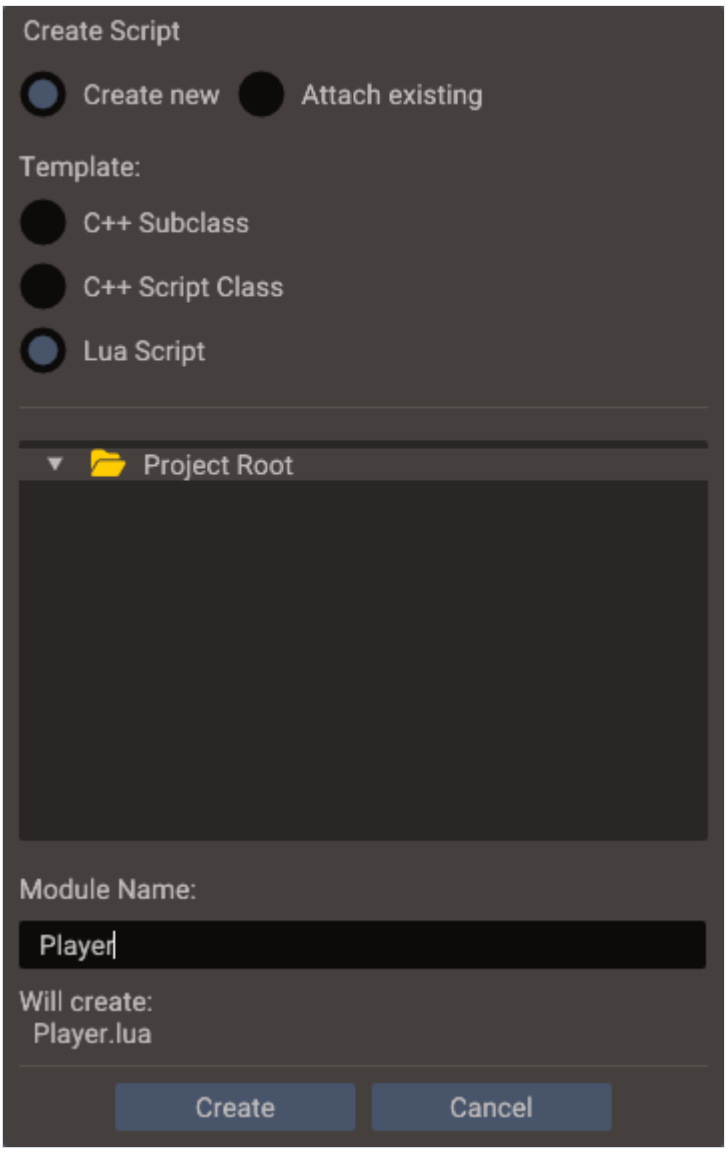
There are two ways to attach a script to an entity. Use **New Script** when you want the editor to generate the files for you; use **Add Script** when the files already exist and you only need a new entry pointing at them.

Flow A — New Script (generates the files)

1. Select an entity in the **Structure** panel.
2. Click **New Script** at the top of the **Properties** window (next to **New component**).
3. Choose **Lua Script**, **C++ Subclass**, or **C++ Script Class**.
4. Enter a class/module name and confirm.

The editor then does everything in one undoable step:

- writes template files into your project (header + source for C++, a single `.lua` module for Lua) with example properties and event registration already in place,
- adds a `ScriptComponent` to the entity if it does not have one,
- appends an enabled script entry linked to the new files.



The same dialog's **Attach existing** option links a script that already exists: browse to the `.lua` file (or the C++ header and source) and click **Attach**. The files are not changed.

Flow B — Add Script (links existing files)

1. Select the entity and add `ScriptComponent` through **New component** (skip if it already has one).
2. In the **ScriptComponent** section, click **Add Script**. This creates an *empty* script entry — no files are written.
3. Click the pencil button on the new entry to open **Edit Script Details** and set the class name, then pick the **Header File** and **Source File** (or the `.lua` file for Lua scripts) from your project.

Use this flow to reuse one script across many entities, or to wire up files you wrote outside the editor.

Managing script entries

Each entry in the ScriptComponent section has its own controls:

Control	Action
Checkbox	Enable or disable the entry without removing it
Pencil button	Open **Edit Script Details** (class name, header, source); the **×** buttons clear a linked file
File buttons	Open the header or source in the integrated Code Editor
Right-click the entry header	**Move Up**, **Move Down**, or **Remove** the entry

Entries run in list order, so use Move Up/Down when one script must initialize before another. All edits are undoable.

Script types

Type	Enum value	Generated base	Best use
Lua Script	`SCRIPT_LUA`	Lua table module	Fast iteration, UI glue, triggers, controllers
C++ Subclass	`SUBCLASS`	The wrapper class matching the entity (`Mesh`, `Camera`, `Terrain`, `Button`, ...)	Behavior that calls wrapper methods like `getPosition()`
C++ Script Class	`SCRIPT_CLASS`	`ScriptBase`	Logic that accesses scene/entity explicitly without a wrapper

How the editor picks a C++ subclass base

The editor inspects the selected entity's components and picks the **most specific** wrapper class for it — the same class it uses when you drag that entity into a header as an entity reference property (p. 227). A few examples:

Entity has	Default base class
`TerrainComponent`	`Terrain`
`ModelComponent`	`Model`
`WaterComponent`	`Water`
`MeshComponent` (and none of the above)	`Mesh`
`ButtonComponent`	`Button`
`CameraComponent`	`Camera`
`Body3DComponent`	`Body3D`
`Transform` (and no recognized component)	`Object`
None of the above	`EntityHandle`

> **Not every base has a transform**
>
> Wrappers such as `Body3D`, `Sound`, and the action classes derive from `EntityHandle` rather than `Object`, so they have no `getPosition()` / `setPosition()` and the generated skeleton leaves the movement example out.

C++ Subclass vs. C++ Script Class

This is the most common source of confusion, so it's worth stating plainly. Both are C++ classes attached through a `ScriptComponent`. The **only** difference is what they inherit from — and that changes how you reach the entity.

C++ Subclass — the script *is* the object.

The class inherits from the object wrapper that matches the entity (the table above shows which one the editor picks). The script instance therefore *is* a `Camera`, `Mesh`, `Object`, etc. for that entity, so you call wrapper methods directly on `this` — no wrapper to construct:

```
// Subclass generated on a Camera entity
class FreeLook : public doriax::Camera {
    void onUpdate() {
        setPosition(getPosition() + offset);   // Object method, called on this
        setView(target);                        // Camera-specific method, called on this
    }
};
```

Yes — exactly as you'd expect: a subclass created on a **Camera** entity inherits from `Camera`, and you get the `Camera` API for free. The wrapper hierarchy is:

```
EntityHandle  →  Object  →  Camera
                         →  Light
                         →  Mesh   →  Shape   →  Mirror
                                  →  Terrain
                         →  Water
              →  Body3D
```

So a `Camera` subclass also has every `Object` method (`getPosition`, `setRotation`, ...) and every `EntityHandle` method (`getComponent`, `addComponent`, `getName`, ...). An `Object` subclass has Object + EntityHandle methods; a bare `EntityHandle` subclass (an entity with no `Transform`) has only the component-access methods.

C++ Script Class — the script *references* the object.

The class inherits from `ScriptBase`, which holds nothing but `getScene()` and `getEntity()`. It implies no object type at all. To touch the entity you construct a wrapper yourself:

```
class ScoreTracker : public doriax::ScriptBase {
    void onUpdate() {
        Object self(getScene(), getEntity());   // wrap the entity explicitly
        self.setPosition(...);
    }
};
```

	C++ Subclass	C++ Script Class
Base class	Closest wrapper (`Camera` / `Terrain` / `Mesh` / `Object` / `EntityHandle`)	`ScriptBase`
Reaching the entity	Call methods on `this` directly	Construct a wrapper from `getScene()` / `getEntity()`
Implies an object type	Yes — matches the entity	No
Best when	The script's job *is* this entity	The script coordinates other entities or holds non-entity logic

Neither is more powerful: a Script Class can do everything a Subclass can by constructing wrappers, and a Subclass can still reach other entities the same way. It is a design choice about ergonomics and intent — pick the base that matches what the script is *about*.

When to use each type

- **Lua** — Prototyping, gameplay that changes often, designers tuning values in the Properties, UI event handlers.
- **C++ Subclass** — Movement, animation control, or physics on the entity the script lives on. You get `setPosition()`, `getBody2D()`, camera/mesh/light methods, etc. directly on `this`.
- **C++ Script Class** — Managers, score trackers, spawners, or logic that touches many entities through `getScene()` and `getEntity()`.

> **Which should a player controller use?**
>
> A player controller almost always drives its **own** entity — reading input and moving its transform — so a **C++ Subclass** is the natural fit. The editor bases it on `Object` (or `Mesh` / `Camera` if the player entity has those components), so `getPosition()` / `setPosition()` and the physics wrappers are available on `this` with no boilerplate. Reach for a **C++ Script Class** only when the controller is really a coordinator that mostly manipulates *other* entities.

Lua script format

A Lua script file **must return a table**. The table is the script prototype; Doriax creates a per-entity instance at runtime.

Minimal template

```lua
local PlayerController = {
    properties = {
        { name = "speed", displayName = "Speed", type = "float", default = 5.0 }
    }
}

function PlayerController:init()
    RegisterEngineEvent(self, "onUpdate")
end

function PlayerController:onUpdate()
    local object = Object(self.scene, self.entity)
    if Input.isKeyPressed(Input.KEY_RIGHT) then
        object.position = object.position + Vector3(self.speed * Engine.deltatime, 0, 0)
    end
end

return PlayerController
```

Required conventions

Rule	Detail
Return a table	`return MyScript` at end of file
Properties table	Optional `properties = { ... }` for Properties fields
`init()`	Optional lifecycle hook called after properties are injected
Event methods	Name methods to match events (`onUpdate`, `onClick`, etc.) and register in `init()`
`self.scene`	Injected `Scene*` — use for systems and entity wrappers
`self.entity`	Injected entity ID

Module loading

`require("myscript")` resolves through the virtual filesystem:

1. `lua://lua/myscript.lua`
2. `lua://myscript.lua`

`ScriptComponent` entries store the path relative to the Lua directory (p. 52) and load it as `lua://` + path. Both lookups go through `Data`, so they still work when a native export packs Lua sources into `resources.pak`.

A module that no entity runs, such as a utility table, is created from the Resources Browser with **New → Lua Script** — see Creating files and folders (p. 109).

Referencing other entities

A script frequently needs to touch entities other than its own — a child model, a sibling UI element, a target the designer picks. There are three ways to get a handle, and choosing the right one (and the right constructor) avoids the most common scripting bug: accidentally creating a new entity instead of referencing an existing one.

1. Your own entity

`self.entity` is the entity the script is attached to. Wrap it with the **two-argument** constructor for the object type you need:

```
local object = Object(self.scene, self.entity)
```

2. An entity the designer picks (entity reference property)

Expose a typed property and drag the target entity onto it in the Properties window. The engine resolves it to a ready-to-use handle (or the target's script instance) before `init()` runs. This is the preferred way to wire up known relationships:

```
MyScript.properties = {
    { name = "target", type = "Object" },
}

function MyScript:init()
    if self.target then
        self.target.position = Vector3(0, 1, 0)
    end
end
```

See Script Properties (p. 223) for the supported reference types.

> **Declare the property by drag and drop**
>
> You don't have to write the property declaration by hand: drag the entity from the **Structure panel** straight into the script open in the Code Editor — onto the `.lua` file or the C++ header. The editor inserts a typed property declaration *and* assigns the dropped entity as its value. See Insert entity references by drag and drop (p. 227).

3. An entity found by name

When a reference isn't wired up, look it up with `Scene:findEntity` (p. 379). It returns the first entity with that name, or `NULL_ENTITY` if none. The two-argument form scopes the search to children of a parent:

```
function MyScript:init()
    local childEntity = self.scene:findEntity("model", self.entity)
    if childEntity ~= NULL_ENTITY then
        self.model = Model(self.scene, childEntity)
    end
end
```

Wrap vs. create: the constructor that bites people

Every object handle has two constructors:

Constructor	Effect	Owns the entity?
`Type(scene, entity)`	**Wraps** an existing entity	No
`Type(scene)`	**Creates a brand-new** entity	Yes

Use the two-argument form whenever the entity already exists (your own entity, a found child, an editor-placed object). Reserve the one-argument form for genuinely spawning new objects at runtime.

Pitfalls to avoid

- **`Type(scene)` does not find anything** — it always creates a new, empty entity. Using it to "get" a model placed in the editor silently produces a duplicate.
- **Don't create or load assets in `onUpdate`.** Loading a model (`Model(scene)` + `loadGLTF`) every frame churns entities and floods the log with errors. Do it once in `init()`.
- **Keep handles to created entities alive.** A one-argument handle *owns* its entity; if you store it only in a local variable, the entity can be destroyed when that handle is garbage-collected. Store runtime-spawned objects in `self`.
- **Cache `findEntity` results.** The lookup is linear over the entity list — resolve once in `init()`, not every frame.

Lua runtime lifecycle

When a scene loads, and again whenever a bundle is spawned into it, `LuaBinding::initializeLuaScripts(scene)` runs three passes over the scripts that have not started yet:

Pass 1 — Create instances

For each enabled `SCRIPT_LUA` entry:

1. Load and execute the Lua module with `require`.
2. Verify the return value is a table.
3. Create an instance table with the module table as `__index`.
4. Set `__name` to the script class name.
5. Inject `scene` and `entity`.
6. Copy Properties property values onto the instance (`self.speed = 5.0`, etc.).
7. Store a Lua registry reference in `ScriptEntry.instance`.

Pass 2 — Resolve entity references

Pointer/entity properties (`EntityReference`) are resolved:

- If the target entity has a matching enabled Lua script, the reference becomes that Lua script instance.
- Otherwise Doriax pushes a typed wrapper (`Object`, `Mesh`, `Camera`, `EntityHandle`, etc.) based on `ptrTypeName`.

Pass 3 — Call `init()`

If the instance has an `init` function, Doriax calls `self:init()`.

Register events inside `init()` so `self` and resolved references are ready.

Cleanup

`LuaBinding::cleanupLuaScripts(scene)` runs on scene unload, and `cleanupLuaScripts(scene, entity)` for each entity `destroyBundle` removes:

1. Remove all `FunctionSubscribe` callbacks whose tag contains the script instance address (`Engine::removeSubscriptionsByTag` and `Scene::removeSubscriptionsByTag`).
2. `luaL_unref` the script instance.
3. Clear `ScriptEntry.instance`.

C++ Subclass

A subclass script derives from the object wrapper that matches the entity and calls wrapper methods directly on `this` — see C++ Subclass vs. C++ Script Class for why.

Header (`PlayerMover.h`)

```
#pragma once

#include "Object.h"
#include "ScriptProperty.h"

class PlayerMover : public doriax::Object {
public:
    DPROPERTY("Speed")
    float speed = 5.0f;

    PlayerMover(doriax::Scene* scene, doriax::Entity entity);
    ~PlayerMover();

    void onUpdate();
};
```

Implementation (`PlayerMover.cpp`)

```
#include "PlayerMover.h"

using namespace doriax;

PlayerMover::PlayerMover(Scene* scene, Entity entity) : Object(scene, entity) {
    REGISTER_ENGINE_EVENT(onUpdate);
}

PlayerMover::~PlayerMover() {
    UNREGISTER_ENGINE_EVENT(onUpdate);
}

void PlayerMover::onUpdate() {
    Vector3 position = getPosition();
    if (Input::isKeyPressed(D_KEY_RIGHT)) {
        setPosition(position + Vector3(speed * Engine::getDeltatime(), 0, 0));
    }
}
```

The editor's **Factory** generates instantiation code when you export or build the project. Subclass scripts are compiled into the game binary.

C++ Script Class (`ScriptBase`)

Use a Script Class when you want a behavior class without implying any object wrapper. It derives from `ScriptBase` — which exposes only `getScene()` and `getEntity()` — so you reach the entity by constructing a wrapper explicitly.

```
#pragma once

#include "ScriptBase.h"
#include "ScriptProperty.h"

class ScoreTracker : public doriax::ScriptBase {
public:
    DPROPERTY("Score")
    int score = 0;

    ScoreTracker(doriax::Scene* scene, doriax::Entity entity);
    ~ScoreTracker();

    void onUpdate();
};
```

```cpp
#include "ScoreTracker.h"

ScoreTracker::ScoreTracker(Scene* scene, Entity entity) : ScriptBase(scene, entity) {
    REGISTER_ENGINE_EVENT(onUpdate);
}

ScoreTracker::~ScoreTracker() {
    UNREGISTER_ENGINE_EVENT(onUpdate);
}

void ScoreTracker::onUpdate() {
    // getScene(), getEntity() available from ScriptBase
}
```

C++ script lifetime

A C++ script is constructed when its scene starts or when `createBundle` (p. 598) spawns its bundle, and deleted when the scene unloads or `destroyBundle` (p. 599) removes the instance. A script that destroys its own bundle is therefore deleted during that call: return right after it without touching a member.

Property sync at runtime

Properties values are stored in `ScriptEntry.properties`. At play/export:

- **C++** — `ScriptProperty::syncToMember()` writes values into the `memberPtr` captured during editor parsing.
- **Lua** — values are set as fields on the instance table before `init()`.

See Script Properties (p. 223) for the full type mapping.

Editor serialization and export

Step	What happens
Scene save	`ScriptComponent` entries serialize to YAML with paths, types, and property values
Property parse	C++: `ScriptParser` reads `DPROPERTY` from headers. Lua: editor loads `properties` table
Play mode	Editor calls `initializeLuaScripts` on scene load and after each `createBundle`
Export	`Generator` emits C++ that registers scenes, creates C++ script instances, and calls `initializeLuaScripts`, for scenes and spawned bundles

Common patterns

UI button handler (Lua)

```lua
local MenuButton = {}

function MenuButton:init()
    local button = Button(self.scene, self.entity)
    local ui = button:getUIComponent()
    RegisterEvent(self, ui.onClick, "onClick")
end

function MenuButton:onClick(x, y)
    print("clicked at", x, y)
end

return MenuButton
```

Physics contact (C++)

```
void Trap::onBeginContact(Body2D bodyA, unsigned long shapeA, Body2D bodyB, unsigned long shapeB) {
    // registered with REGISTER_EVENT(physics->beginContact2D, onBeginContact)
}
```

Referencing another entity (Lua)

```
local Follower = {
    properties = {
        { name = "target", displayName = "Target", type = "Object" }
    }
}

function Follower:init()
    RegisterEngineEvent(self, "onUpdate")
end

function Follower:onUpdate()
    if self.target then
        local obj = Object(self.scene, self.entity)
        obj.position = self.target.position
    end
end

return Follower
```

Troubleshooting

Problem	Likely cause
`init()` never runs	Script entry disabled, wrong type, or Lua module does not return a table
Event handler not called	Forgot `RegisterEngineEvent` / `REGISTER_ENGINE_EVENT` in constructor or `init()`
Property not in Properties	`DPROPERTY` not immediately above member; unsupported type; header path wrong
Entity reference is nil	Target entity missing, wrong scene, or referenced script disabled
Duplicate event callbacks	Same tag registered twice; use `UNREGISTER_*` in destructor
`obj.position.y = v` does not move the object	Properties return a copy of the vector; assign the whole value, e.g. `obj.position = Vector3(x, v, z)` — see Math & Utilities (p. 329)
C++ build or CMake errors	Missing toolchain, or a leftover deleted script still listed in generated sources — see C++ Build Setup (p. 209)

Next steps

- C++ Build Setup (p. 209) — compilers, CMake, and build troubleshooting
- Events (p. 215) — full event list, macros, and Lua registration
- Script Properties (p. 223) — `DPROPERTY` and Lua `properties` tables
- Engine (p. 361) — runtime API
- API Index (p. 352) — all classes

C++ Build Setup

Lua scripts run as-is, but **C++ scripts are compiled on your machine** the first time you press Play (and again whenever a C++ source changes). The editor generates a CMake project for your scripts, configures it, and builds a shared library that is loaded back into the editor. This page covers what tools that build step needs and how to fix the most common failures.

If you only use Lua scripts, no build tools are required.

Required tools

- **CMake 3.20 or newer**, available on `PATH`.
- A compiler for your platform (see below) that supports the project's **C++ Standard** — C++17 by default, set under Project Settings → Build (p. 59).

When you press Play in a project that contains C++ scripts, the editor first checks for these tools and shows a *Missing Build Tools* dialog listing anything it could not find.

Windows

Install one of:

- **Visual Studio 2019/2022** (Community is fine) with the **"Desktop development with C++"** workload — the simplest option. The editor finds it automatically even when `cl.exe` is not on `PATH`.
- **LLVM/Clang** (MSVC target, e.g. the `clang+llvm-*-pc-windows-msvc` release) — also requires **Ninja** on `PATH` (see the warning below).
- **MinGW GCC or Clang** — uses the *MinGW Makefiles* generator. Only works with a **MinGW build of the editor**, *not* the official MSVC download — see the ABI note under Choosing a compiler.

Install CMake from cmake.org/download[8] and check **"Add CMake to the system PATH"** during setup.

Linux

```
sudo apt install cmake build-essential   # Debian/Ubuntu
sudo dnf install cmake gcc-c++           # Fedora
```

macOS

```
xcode-select --install   # Apple Clang
brew install cmake
```

Choosing a compiler

Open **Edit → Editor Settings** (p. 68) **→ Desktop** and use the **Compiler** dropdown. The editor detects available compilers (GCC, Clang, MSVC/Visual Studio) and pairs each one with a compatible CMake generator:

Compiler (Windows)	CMake generator	C++ ABI	Official (MSVC) build
MSVC / Visual Studio	The newest installed `Visual Studio N YYYY` your CMake supports, or Ninja with `cl`	MSVC	Yes
Clang targeting MSVC (`x86_64-pc-windows-msvc`)	Ninja — **required**	MSVC	Yes
MinGW GCC or Clang (`*-mingw*`)	MinGW Makefiles	GNU	No — needs a MinGW editor build

Default does not simply hand the choice to CMake on Windows. The editor runs the same detection as the dropdown and picks the best **ABI-compatible** kit it finds, pinning both the compiler and the generator; the

8. https://cmake.org/download/

result is printed in the Output panel as *"Default compiler resolved to: …"*. On Linux and macOS, where a single system C++ ABI makes any detected compiler usable, Default leaves the toolchain to CMake.

> **Why Default pins an explicit toolchain on Windows**
>
> A bare `cmake` picks whatever toolchain that particular CMake prefers — and an **MSYS2 CMake earlier on `PATH`** than the Visual Studio one prefers Ninja with MinGW GCC, even when Visual Studio is installed. On the official MSVC editor that silently produced a GNU-ABI plugin that cannot link the engine. Pinning removes the ambiguity.

Your selection is stored **per project**, so a project that needs MSVC and one that needs MinGW can coexist, and it is **remembered across sessions** — a project that has no kit yet inherits the compiler you last chose, so you only need to set it once (rather than having each new project silently fall back to **Default**).

Because a compiler path only means something on the machine it was installed on, the kit is written to the project's `.doriax/user/build.yaml` — not to `project.yaml`. That file is not committed, so it never reaches version control and a teammate opening the same project picks their own kit, while a project moved or copied to another folder keeps working. Projects saved by older editors carry the kit in `project.yaml` or in the editor's `settings.yaml`; opening one migrates it once, without overwriting a kit this machine already chose.

> **Your compiler must match the editor's own C++ ABI**
>
> C++ scripts are compiled into a shared library that is loaded **into the running editor and shares its engine instance** — so the plugin must use the same C++ ABI as the editor was built with. MSVC and MinGW (GCC) are **not** ABI-compatible: they disagree on name mangling, class layout, RTTI, exception handling and STL types. A MinGW plugin therefore cannot link against an MSVC-built engine — you get a wall of `undefined reference` errors — and the reverse fails just as badly.
>
> The **official Windows download is built with MSVC**, so with it you must use **MSVC** or **Clang targeting MSVC** (both free — see Required tools). To build plugins with **MinGW GCC**, you need a MinGW build of the editor (see Building from Source → Windows (p. 750)). The two never mix: one editor download supports one ABI.
>
> The editor knows which toolchain it was built with and **greys out incompatible compilers** in the **Compiler** dropdown — hover one to read why. If the kit a project already had is no longer detected, the dropdown falls back to **Default** and warns that builds still use the stored one until you press OK. This is a Windows-only concern; Linux and macOS each have a single system C++ ABI, so GCC and Clang interoperate there.

> **Clang on Windows needs Ninja**
>
> CMake's Visual Studio generator always compiles with MSVC's `cl.exe` — it cannot drive a standalone `clang.exe`. A Clang (MSVC target) compiler therefore only works with the **Ninja** generator. If Ninja is not installed, the editor lists the Clang entry greyed-out with a *"requires Ninja on PATH"* note.
>
> Install Ninja with `winget install Ninja-build.Ninja`, or download `ninja-win.zip` from ninja-build.org[9], unzip it to a folder on `PATH`, and verify with `ninja --version` in a new terminal. Restart the editor afterwards so it re-detects the compiler list.
>
> Do **not** use a pip-installed `ninja.exe` copied out of Python's `Scripts` folder — it is a wrapper that stops working outside its install location. If Windows blocked a downloaded exe, right-click it → **Properties** → **Unblock**.

9. https://ninja-build.org

> **Clang also needs the Windows SDK**
> A standalone Clang compiles your code with its own toolchain but still **links against the Windows SDK and the MSVC C runtime** (`kernel32.lib`, `msvcrtd`, ...). Those ship with Visual Studio's "Desktop development with C++" workload. The editor loads the Visual Studio environment automatically before building with Ninja, so you do **not** need to launch it from a Developer Command Prompt. If linking fails with `lld-link: error: could not open 'kernel32.lib'`, the Windows SDK is either not installed or could not be located — install it via the Visual Studio Installer.

Switching compilers is safe at any time: the editor detects the change — including a change of target architecture — and cleans the build directory automatically (CMake cannot switch generators or compilers in-place).

Parallel builds

C++ scripts compile in parallel: by default the editor runs **one build job per detected logical CPU thread**. That is dramatically faster on multi-core machines but raises peak memory use, since every concurrent compiler process needs its own working memory.

Open **File → Export Project... → Desktop** and use **Build Jobs** to tune this. The field starts at the automatic value — one job per logical CPU thread — and the same number is used for Play-mode script builds, not just for the export you are configuring.

Value	Effect
Automatic (default)	One job per logical CPU thread of the machine running the build
`1`	Serial build — slowest, but minimal memory use
Any other number	At most that many concurrent jobs

Lower the value if builds fail with out-of-memory errors or make the system unresponsive — each concurrent C++ compile can use hundreds of MB of RAM, more with heavy template use.

The value is machine-local, like the compiler: it is saved per project in `.doriax/user/build.yaml` (as `build_jobs`, omitted when automatic), not in `project.yaml`, so a limit chosen on a 32-core workstation is never imposed on a teammate's 4-core laptop. Whatever the stored number, each build caps it at **4× the machine's logical CPU count**, up to 65536.

Projects saved by older editors carry the setting as `cmakeBuildJobs` in `project.yaml`. Opening one migrates the value into the editor settings once — see Editor Settings (p. 72).

> **MSVC rebuilds scripts when the value changes**
> With the MSVC compiler the limit is applied through the compiler's `/MP` flag, because MSBuild alone cannot throttle per-file compilation within a single target. Since that changes compile flags, changing **Build Jobs** triggers a full script rebuild on the next Play. Ninja and Make apply the limit at the build-tool level, so changing it there only affects new compilations.

Script directories

By default the editor compiles only the C++ scripts a scene actually uses — the files referenced by a Script component. A helper class that no entity mentions is never built, and every header is included by its full path from the project root.

Project → Project Settings (p. 58) **→ Directories → Script Directories** changes that. Add any folder inside your project and it becomes two things at once:

Role	Effect
Include root	Headers under it are included by the path *below* the folder, so the full path from the project root is no longer needed
Compile root	Every `.cpp` under it is compiled, even when no entity references it

Only the folders you add behave this way — the rest of the project is untouched, and a project with no script directories builds exactly as before.

This lets you lay out code in modules and include across them by short path. With `Source/Public` added as a root, a header at `Source/Public/Player/Script.h` becomes:

```
#include "Player/Script.h"   // instead of "Source/Public/Player/Script.h"
```

Add as many roots as you need; they are searched in the order listed. Both roles apply to Play builds and to exported projects (p. 155) alike.

To add files to a root, right-click it in the Resources Browser and use **New → C++ Source** or **New → C++ Header** — see Creating files and folders (p. 109).

> **Keep roots inside the project**
> A folder outside the project compiles in the editor but **cannot be exported** — the export copies the project tree only, and warns that the directory was skipped. Use a folder inside the project for anything you intend to ship.

The list is saved as `scriptDirs` in `project.yaml` and travels with the project.

Customizing the build

The `CMakeLists.txt` at your project root is generated: Play and Save overwrite it, so edits to it are lost. To add your own CMake — linking a third-party library, extra compile definitions, platform flags — create **`ProjectBuild.cmake`** next to it. The editor never creates, reads or overwrites that file, and both the editor build and exported CMake builds include it when it exists.

```
target_link_libraries(${DORIAX_TARGET} PRIVATE mylib)
target_include_directories(${DORIAX_TARGET} PRIVATE ${DORIAX_SCRIPTS_DIR}/ThirdParty/include)
```

Two variables are set for you, with the same meaning in both builds:

Variable	Value
`DORIAX_TARGET`	The target to attach to
`DORIAX_SCRIPTS_DIR`	The root your `.h` / `.cpp` paths are relative to

> **Attach through `${DORIAX_TARGET}`, not the target name**
> The target is named after your project in editor builds and after the application name in exported ones, so a hard-coded name works in one and breaks in the other — and breaks again if you rename the project.

> **Not used by the exported Xcode workspace**
> The Xcode workspace of a Source Code export, `workspaces/xcode/Doriax.xcodeproj`, is not a CMake build, so `ProjectBuild.cmake` has no effect there and the export warns about it. Make the same changes in Xcode, or build through CMake with `python3 doriax.py -p ios-xcode` or `-p macos-xcode`, which does include the file.

For include paths alone you do not need this file: Script directories covers that case from the editor. The language standard has its own setting too — `CMAKE_CXX_STANDARD` is written from Project Settings → Build (p. 59), so do not set it here.

Where build files live

Play and Save rewrite the generated C++ glue: `CMakeLists.txt` at the **project root**, `scene_scripts.cpp` and `LocalPaths.cmake` under **`.doriax`**, and `main.cpp` plus the scene and bundle factories under **`.doriax/generated`**. Together they cover every C++ script and `.bundle` file that still exists on disk. Do **not** edit those files by hand — the next Play or Save overwrites them. Put your own build settings in `ProjectBuild.cmake` instead, which is never overwritten.

`CMakeLists.txt` names no path belonging to your machine, so the same file works for everyone: the asset and Lua roots resolve from `${PROJECT_ROOT}`, and the shader cache and engine library directory come from `LocalPaths.cmake`, which the editor rewrites for whichever machine opens the project. Both are regenerated rather than committed — see Version Control (p. 65).

Configuring outside the editor works the same way, as long as the project has been opened in the editor once so `.doriax` exists. Pass `-DDORIAX_SHADER_PATH=<dir>` to override the shader cache location.

The CMake build tree is generated under **`.doriax/build`**. It is a disposable cache — deleting it forces a full clean reconfigure and rebuild, which is a good first step when the build state seems corrupted (for example after moving the project, or upgrading CMake or the compiler).

Troubleshooting

All configure and compile output streams into the **Output panel**. The first error line is almost always the meaningful one.

Symptom in the Output panel	Likely cause and fix
`Missing Build Tools` dialog	CMake or a C++ compiler is not installed / not on `PATH`. Install it and restart the editor.
`C++ toolchain compatible with this editor: not found` in that dialog	Compilers were found, but none matches the editor's C++ ABI. The dialog lists every detected compiler with the reason it was rejected — see Choosing a compiler.
MSVC greyed out with *"requires a CMake version supporting the installed Visual Studio, or Ninja and an MSVC toolchain environment"*	Your CMake is older than your Visual Studio, so it has no matching `Visual Studio N YYYY` generator. Update CMake, install an older Visual Studio, or install Ninja so the editor can drive `cl.exe` through it.
`The C compiler ... is not able to compile a simple test program` together with `CMAKE_LINKER-NOTFOUND` or `TRK0005`	A custom compiler (usually Clang) was configured with the Visual Studio generator. Install Ninja and re-select the Clang compiler in Editor Settings, or switch to the MSVC compiler. Editor versions before June 2026 allowed this broken combination.
`CMake was unable to find a build program corresponding to "Ninja"`	`ninja.exe` is not on `PATH`, or it exists but cannot run (pip wrapper copied out of place, or blocked download). Install the standalone binary and check `ninja --version` works in a fresh terminal.
`undefined reference to` **every** engine symbol (`doriax::Vector3::Vector3`, `doriax::Engine::onUpdate`, ...) at the **link** step	The selected compiler's C++ ABI does not match the editor's — typically MinGW GCC against the official MSVC build. Use MSVC or Clang-targeting-MSVC on the MSVC build (or a MinGW editor for MinGW). Current editors grey out incompatible compilers and stop the build at configure time with a readable message, so this mainly appears in older versions or when building outside the editor. See Choosing a compiler.
`This Doriax editor requires MSVC-compatible C++ plugins` (or `... MinGW/GNU C++ plugins`) at the **configure** step	The selected compiler's ABI does not match the editor's, and the generated CMake project stops before compiling anything. Pick a compatible compiler in **Editor Settings → Desktop → Compiler**; this mainly appears with a compiler saved before the ABI check existed, or one selected outside the editor. See Choosing a compiler.
The build uses `C:\msys64\ucrt64\bin\g++.exe` even though Visual Studio is installed and **Compiler** is **Default**	An MSYS2 `cmake` came earlier on `PATH` than the Visual Studio one, and older editors let CMake choose the toolchain — which that CMake resolves to Ninja with MinGW GCC. Update the editor (Default now pins an ABI-compatible toolchain), or select **MSVC** explicitly in **Editor Settings → Desktop → Compiler**.

Symptom in the Output panel	Likely cause and fix
`lld-link: error: could not open 'kernel32.lib'` (or other `*.lib`)	Clang cannot find the Windows SDK libraries. Install the Windows SDK through the Visual Studio Installer ("Desktop development with C++"). The editor loads the MSVC environment automatically; if it can't find Visual Studio it warns and you can instead launch the editor from a Developer Command Prompt.
`No CMAKE_C_COMPILER could be found` (with `Building for: Visual Studio 17 2022`)	CMake's Visual Studio auto-detection found no usable compiler. Current editors no longer route **Default** through that auto-detection on Windows, so update the editor first. Otherwise pick an explicit compiler in **Editor Settings → Desktop → Compiler** (MSVC, or Clang). The choice is remembered across projects, so you only set it once. If even MSVC fails here, your Visual Studio C++ toolset is incomplete — repair "Desktop development with C++" in the Visual Studio Installer.
`internal compiler error: Killed` (GCC/Clang) or `fatal error C1060: compiler is out of heap space` (MSVC)	The parallel build ran out of memory. Lower **Export Project → Desktop → Build Jobs** (try `1` or `2`) — see Parallel builds.
`Does not match the generator used previously`	The build directory holds a cache from a different generator. The editor normally cleans it automatically when you switch compilers; if it reports the directory is **locked**, close anything using `.doriax/build` (notably the VS Code *CMake Tools* extension) or delete the folder manually, then try again.
`CMake configuration failed` with **no other output**	Update the editor — older versions dropped the output of commands that failed quickly. Current versions print the full CMake error plus a `Process exited with code N` line.
`CMake Error: File ... does not exist.` (or similar) for a script or `.bundle` you already deleted	Generated sources still listed a missing file. Current editors skip files that are gone from disk; press Play or Save once to regenerate. Do not patch the generated `CMakeLists.txt` — it is overwritten. If a closed scene still shows the old ScriptComponent entry, open that scene, remove the leftover, and save.
`Unknown type SomeClass`	A leftover C++ script class name is still registered. Remove that ScriptComponent entry from every scene and bundle that still lists it, then Play or Save. Recreating a script with the same class name also recovers a project that was already broken.
`Compiler kit or engine library directory changed...` or `CMake generator or architecture changed...` followed by *Cleaning build directory*	Not an error — the compiler, generator, target architecture or editor install changed, so the editor wipes `.doriax/build` and reconfigures from scratch.
Errors persist after fixing the toolchain	Delete `.doriax/build` and press Play again to force a clean configure.

Next steps

- Creating Scripts (p. 199) — script types and editor workflow
- Code Editor (p. 136) — editing C++ in the editor and reading build output
- Editor Settings (p. 68) — where the compiler and CMake path are configured
- Export Window (p. 155) — building standalone projects outside the editor

Events

Doriax events use `FunctionSubscribe<Ret(Args...)>`, a multicast delegate that stores callbacks with string tags. Adding a callback with an existing tag **replaces** the previous one, which prevents duplicate subscriptions from the same source.

Events work identically in C++ and Lua. C++ uses macros from `FunctionSubscribe.h`. Lua uses `RegisterEvent` and `RegisterEngineEvent` globals registered by `LuaBinding`.

> **Let the Code Editor write it**
>
> The event menu (p. 137) of the Code Editor adds the registration and an empty handler for any event on this page, in C++ or Lua.

How dispatch works

```
Platform input / frame tick
        │
        ▼
Engine.systemTouchStart / systemUpdate
        │
        ▼
FunctionSubscribe.call ──► C++ callbacks
                       └─► Lua closures
```

1. The platform layer calls `Engine::systemDraw()` (or `systemUpdate()` when nothing can be drawn), `systemTouchStart()`, etc.
2. Engine forwards to static `FunctionSubscribe` members (`onUpdate.call()`, etc.).
3. Subsystems and components fire their own events during simulation (physics contacts, UI pointer events, action start/stop).
4. Each subscriber runs in registration order unless disabled or removed.

A callback can add or remove subscribers, or destroy the object that owns the event. A subscriber added during a call first runs on the next one, and one removed before its turn is skipped.

FunctionSubscribe API

Defined in `engine/core/util/FunctionSubscribe.h`.

Method	Purpose
`add(tag, callback)`	Register a callback; replaces existing tag
`add(tag, lua_State*)`	Register a Lua function with tag
`remove(tag)`	Remove one callback by exact tag
`removeByTagSubstring(substr)`	Remove all callbacks whose tag contains substring
`clear()`	Remove all callbacks
`call(args...)` / `operator()`	Invoke all enabled callbacks
`setEnabled(bool)`	Enable or disable the entire event without removing subscribers

With `DORIAX_CRASH_GUARD` enabled, failing subscribers are removed automatically and optionally reported through a global crash handler. A C++ exception that escapes a subscriber counts as a failure too, and the report carries its message. Copies of an event share their subscribers, so a failing one stops running in all of them. A subscriber stays guarded after it calls another event, and a crash outside the subscribers goes to the handler that was installed before, such as the editor's crash log (p. 769).

Engine events

`Engine` exposes frame, lifecycle, input, and pointer events.

Event	Signature	Use
`onInit`	`void()`	User init via `DORIAX_INIT` macro (access through `getOnInit()`)
`onViewLoaded`	`void()`	View/surface ready
`onViewChanged`	`void()`	Canvas/view changed; Lua name: `onCanvasChanged`
`onViewDestroyed`	`void()`	View destroyed
`onDraw`	`void()`	Every drawn frame, before the scenes are drawn
`onUpdate`	`void()`	Variable-step gameplay update
`onFixedUpdate`	`void()`	Fixed-step simulation update
`onPostUpdate`	`void()`	Work after regular update
`onPause` / `onResume`	`void()`	App/game pause and resume
`onShutdown`	`void()`	Runtime shutdown
`onSceneLoaded`	`void()`	A scene loaded by `SceneManager` finished loading, before its loading scene closes
`onTouchStart/End/Move`	`void(int, float, float)`	Touch identifier, x, y
`onTouchCancel`	`void(int, float, float)`	One touch canceled; identifier and last recorded canvas position
`onMouseDown/Up`	`void(int, float, float, int)`	Button, x, y, modifiers
`onMouseMove/Scroll`	`void(float, float, int)`	Position or scroll offsets plus modifiers
`onMouseEnter/Leave`	`void()`	Pointer enters/leaves view
`onKeyDown/Up`	`void(int, bool, int)`	Key, repeat flag, modifiers
`onCharInput`	`void(wchar_t)`	Text input codepoint
`onGamepadConnect/Disconnect`	`void(int)`	Gamepad id connected/disconnected
`onGamepadButtonDown/Up`	`void(int, int)`	Gamepad id, button index
`onGamepadAxisMove`	`void(int, int, float)`	Gamepad id, axis index, value `-1` … `1`

Touch cancellation

Use `onTouchEnd` for a finger lifting normally and `onTouchCancel` for an interrupted contact. Both identify the finger, but cancellation does not dispatch UI pointer-up or click events. In a per-pointer cancellation, the canceled touch is removed from `Input` before its callback runs; other fingers remain active. Release only the gameplay action associated with that identifier in your handler.

A full cancellation removes all tracked touches before emitting `onTouchCancel` once for each of them. There is no separate global notification; an empty full cancellation emits no callbacks. Each callback receives the last recorded position in logical canvas coordinates. An already removed or unknown identifier produces no additional notification. Use `Input::numTouches()` (`Input.numTouches()` in Lua) to check whether any fingers remain; it is zero during every full-cancel callback.

Subscribe to `onTouchCancel` as well as `onTouchEnd` when tracking held fingers. Cancellation notifications follow `pauseGameEvents`, and are not filtered by `setIgnoreEventsHandledByUI` or the current pointer position.

In the Code Editor, choose **Add event > Input > Touch > onTouchCancel** (`Touch` is the section label). The generated handler has this signature:

```
void onTouchCancel(int pointer, float x, float y);
```

Use `REGISTER_ENGINE_EVENT(onTouchCancel)` in the C++ constructor and `UNREGISTER_ENGINE_EVENT(onTouchCancel)` in the destructor. In Lua:

```lua
function Player:init()
    RegisterEngineEvent(self, "onTouchCancel")
end

function Player:onTouchCancel(pointer, x, y)
    -- Release the action associated with this pointer.
end
```

Input routing options

Configure how events reach gameplay vs. UI:

Engine setting	Effect
`setIgnoreEventsHandledByUI(true)`	Gameplay handlers skip events already consumed by UI
`setAllowEventsOutCanvas(false)`	Clip events to the logical canvas
`pauseGameEvents(true)`	Pause game event delivery (editor pause, etc.)
`setCallMouseInTouchEvent` / `setCallTouchInMouseEvent`	Mirror mouse/touch for cross-platform testing

C++ registration macros

All macros live in `FunctionSubscribe.h` and generate a unique tag: `typeid(ClassName)_objectAddress_methodName`.

Engine events

```cpp
REGISTER_ENGINE_EVENT(onUpdate);
UNREGISTER_ENGINE_EVENT(onUpdate);
```

Any FunctionSubscribe member

```cpp
PhysicsSystem* physics = scene->getSystem<PhysicsSystem>().get();
REGISTER_EVENT(physics->beginContact2D, onBeginContact);
UNREGISTER_EVENT(physics->beginContact2D, onBeginContact);
```

Component events on the same entity

```cpp
REGISTER_COMPONENT_EVENT(UIComponent, onClick, onClick);
REGISTER_UI_EVENT(onClick, onClick);           // shorthand for UIComponent
REGISTER_BUTTON_EVENT(onPress, onPress);       // ButtonComponent
REGISTER_SCROLLBAR_EVENT(onChange, onChange);  // ScrollbarComponent
REGISTER_PANEL_EVENT(onMove, onMove);          // PanelComponent
```

Matching `UNREGISTER_*` macros remove subscriptions in destructors.

Example: full C++ script

```cpp
class DoorTrigger : public doriax::ScriptBase {
public:
    DoorTrigger(doriax::Scene* scene, doriax::Entity entity);
    ~DoorTrigger();

    void onUpdate();
    void onClick(float x, float y);
    void onBeginContact(doriax::Body2D bodyA, unsigned long shapeA, doriax::Body2D bodyB, unsigned lon
g shapeB);

private:
```

```
    doriax::PhysicsSystem* physics = nullptr;
};

DoorTrigger::DoorTrigger(Scene* scene, Entity entity) : ScriptBase(scene, entity) {
    physics = scene->getSystem<PhysicsSystem>().get();
    REGISTER_ENGINE_EVENT(onUpdate);
    REGISTER_UI_EVENT(onClick, onClick);
    if (physics) {
        REGISTER_EVENT(physics->beginContact2D, onBeginContact);
    }
}

DoorTrigger::~DoorTrigger() {
    UNREGISTER_ENGINE_EVENT(onUpdate);
    UNREGISTER_UI_EVENT(onClick, onClick);
    if (physics) {
        UNREGISTER_EVENT(physics->beginContact2D, onBeginContact);
    }
}
```

Lua event registration

`LuaBinding::registerHelpersFunctions` exposes two globals:

Function	Arguments	Behavior
`RegisterEvent(self, event, methodName [, tag])`	Script table, event object, method name	Subscribes `self:methodName(...)` to any exposed `FunctionSubscribe`
`RegisterEngineEvent(self, methodName [, tag])`	Script table, engine event name	Looks up `Engine.<methodName>` and subscribes

If `tag` is omitted, Doriax builds: `ClassName_instancePointer_methodName`

Lua callbacks always run on the main Lua thread, even when they are subscribed or fired inside a coroutine, so they can't yield.

Engine event (Lua)

```
function Player:init()
    RegisterEngineEvent(self, "onUpdate")
end

function Player:onUpdate()
    -- called every frame
end
```

Component event (Lua)

Lua has no per-component macros. The pattern that replaces `REGISTER_UI_EVENT`, `REGISTER_BUTTON_EVENT`, and the other component macros is always the same: wrap the entity, get the component, and pass its event to `RegisterEvent`:

```
function Trigger:init()
    -- Get the UIComponent through a wrapper for this entity
    local image = Image(self.scene, self.entity)
    local ui = image:getUIComponent()
    RegisterEvent(self, ui.onClick, "onClick")
end

function Trigger:onClick(x, y)
    print("clicked", x, y)
end
```

C++ macro	Lua equivalent
`REGISTER_ENGINE_EVENT(onUpdate)`	`RegisterEngineEvent(self, "onUpdate")`
`REGISTER_UI_EVENT(onClick, onClick)`	`RegisterEvent(self, wrapper:getUIComponent().onClick, "onClick")`
`REGISTER_BUTTON_EVENT(onPress, onPress)`	`RegisterEvent(self, button:getButtonComponent().onPress, "onPress")`
`REGISTER_SCROLLBAR_EVENT(onChange, onChange)`	`RegisterEvent(self, scrollbar:getScrollbarComponent().onChange, "onChange")`
`REGISTER_PANEL_EVENT(onMove, onMove)`	`RegisterEvent(self, panel:getPanelComponent().onMove, "onMove")`
`REGISTER_EVENT(physics->beginContact2D, onBeginContact)`	`RegisterEvent(self, physics.beginContact2D, "onBeginContact")`
`REGISTER_EVENT(AdMob::onAdLoaded, onAdLoaded)`	`RegisterEvent(self, AdMob.onAdLoaded, "onAdLoaded")`

```lua
function MenuButton:init()
    local button = Button(self.scene, self.entity)
    local bc = button:getButtonComponent()
    RegisterEvent(self, bc.onPress, "onPress")
end

function MenuButton:onPress()
    print("button pressed")
end
```

Cleanup is also automatic in Lua: when the scene unloads, or `destroyBundle` removes the script's entity, all subscriptions tagged with the script instance are removed — there is no `UNREGISTER_*` to call.

Physics event (Lua)

```lua
function Trap:init()
    local physics = self.scene:getPhysicsSystem()
    RegisterEvent(self, physics.beginContact2D, "onBeginContact")
end

function Trap:onBeginContact(bodyA, shapeA, bodyB, shapeB)
    print("2D contact")
end
```

The filters (`preSolve2D`, `shouldCollide2D`, `shouldCollide3D`) use what the method returns, so end them with an explicit `return`: a method that returns nothing rejects the contact. See PhysicsSystem (p. 568).

Custom tag

```lua
RegisterEngineEvent(self, "onUpdate", "player1_update")
```

Use custom tags when you need to replace or manually remove a specific subscription.

Physics events

`PhysicsSystem` exposes 2D (Box2D) and 3D (Jolt) collision events.

2D events

Event	Signature	Meaning
`beginContact2D`	`void(Body2D, unsigned long, Body2D, unsigned long)`	Contact begins
`endContact2D`	`void(Body2D, unsigned long, Body2D, unsigned long)`	Contact ends
`hitContact2D`	`void(Body2D, unsigned long, Body2D, unsigned long, Vector2, Vector2, float)`	Impact with hit data

Event	Signature	Meaning
`beginSensorContact2D`	`void(Body2D, unsigned long, Body2D, unsigned long)`	Sensor overlap begins
`endSensorContact2D`	`void(Body2D, unsigned long, Body2D, unsigned long)`	Sensor overlap ends
`preSolve2D`	`bool(Body2D, unsigned long, Body2D, unsigned long, Manifold2D)`	Return `false` to disable contact before solve
`shouldCollide2D`	`bool(Body2D, unsigned long, Body2D, unsigned long)`	Filter pairs before collision

`beginContact2D`, `endContact2D`, `hitContact2D` and `preSolve2D` fire only for pairs where one of the shapes has **Contact Events**, **Enable Hit Events** or **PreSolve Events** on, and each is off by default — see Body2D events (p. 548).

`hitContact2D` passes the contact point in world points, the unit normal, and the approach speed in points per second, the units of Body2D (p. 544).

3D events

Event	Meaning
`onBodyActivated3D` / `onBodyDeactivated3D`	Body activation state changed
`onContactAdded3D` / `onContactPersisted3D` / `onContactRemoved3D`	3D contact lifecycle
`shouldCollide3D`	Filter whether two 3D bodies/shapes should collide

Unlike the 2D contact events, these run while Jolt is stepping, which limits what they may do to the world — see What a 3D callback may do (p. 321).

UI component events

`UIComponent` events (also available through `REGISTER_UI_EVENT`):

Event	Signature	When
`onGetFocus` / `onLostFocus`	`void()`	Focus changes
`onPointerEnter` / `onPointerLeave`	`void(float, float)`	Pointer enters/leaves widget bounds
`onPointerMove`	`void(float, float)`	Pointer moves over widget
`onPointerDown` / `onPointerUp`	`void(float, float)`	Pointer button down/up
`onClick` / `onDoubleClick`	`void(float, float)`	Click gestures
`onDragStart` / `onDrag` / `onDragEnd`	`void(float, float)`	Drag gestures

Widget-specific events

Component	Events
`ButtonComponent`	`onPress`, `onRelease`
`ScrollbarComponent`	`onChange`
`PanelComponent`	`onMove`, `onResize`
`TextEditComponent`	`onChange`, `onSubmit`

Action and sound events

Component	Events
`ActionComponent`	`onStart`, `onStop`, `onPause`, `onStep`
`SoundComponent`	`onStart`, `onStop`, `onPause`

Use these to chain gameplay reactions to tweens, timelines, or audio playback.

Service events

The monetization services (p. 337) report through static events of their classes, which belong to no entity or scene:

Class	Events
`AdMob` (p. 615)	`onInitialized`, `onConsentUpdated`, `onAdLoaded`, `onAdFailedToLoad`, `onAdShown`, `onAdFailedToShow`, `onAdDismissed`, `onAdClicked`, `onAdImpression`, `onUserEarnedReward`, `onAdPaid`, `onAdInspectorClosed`
`InAppPurchase` (p. 622)	`onInitialized`, `onDisconnected`, `onProductsQueried`, `onPurchaseUpdated`, `onPurchaseFailed`, `onPurchasesQueried`, `onPurchaseAcknowledged`, `onPurchaseConsumed`
`WebPortal` (p. 628)	`onInitialized`, `onAdStarted`, `onAdFinished`, `onAdError`, `onDataLoaded`, `onDataLoadFailed`, `onDataSaveFailed`

Pass the static member to `REGISTER_EVENT` in C++ or `RegisterEvent` in Lua:

C++

```
REGISTER_EVENT(AdMob::onUserEarnedReward, onUserEarnedReward);
UNREGISTER_EVENT(AdMob::onUserEarnedReward, onUserEarnedReward);

void Shop::onUserEarnedReward(AdMobFormat format, std::string type, int amount) { }
```

Lua

```
RegisterEvent(self, AdMob.onUserEarnedReward, "onUserEarnedReward")

function Shop:onUserEarnedReward(format, rewardType, amount) end
```

The platform answers on its own thread, and the engine calls these subscribers at the start of the next frame, on the game thread. A C++ handler must take the parameters by value, exactly as the event declares them (`std::string`, not `const std::string&`). Since the events outlive scenes, remove C++ subscriptions in the destructor; Lua script subscriptions go away with the script. See How results arrive (p. 337).

Easing as an event

`Ease` extends `FunctionSubscribe<float(float)>`. Timed actions can use custom easing curves by subscribing to an `Ease` instance.

Cleanup

Context	Cleanup mechanism
C++ script destructor	Call matching `UNREGISTER_*` macros
Lua script unload	`cleanupLuaScripts` removes tags containing instance pointer
Bundle instance destroyed	`destroyBundle` stops its C++ and Lua scripts as a scene unload does
Scene unload	`Scene::removeSubscriptionsByTag`, including the `PhysicsSystem` events
Component callbacks	`Engine::clearComponentSubscriptions(scene)`
Full reset	`Engine::clearAllSubscriptions()`

Always unregister in C++ destructors when the object can outlive a scene reload. A callback added straight to a component event (`button.onPress.add("pressed", ...)`) is not removed with the script that added it, so unregister it too — `Engine::clearComponentSubscriptions(scene)` drops every component callback of a scene at once.

Best practices

- Register in a constructor or Lua `init()`. Never register in hot per-frame code.

- Unregister in destructors (C++) or rely on Lua cleanup for script components.
- Use `onFixedUpdate` for physics-sensitive simulation, not `onUpdate`.
- Inside `onFixedUpdate`, move physics bodies with `Body2D`/`Body3D` `setPosition` or velocities — an `Object::setPosition` on a body-carrying entity is discarded there. See Moving a body (p. 320).
- Enable `setIgnoreEventsHandledByUI(true)` when UI and gameplay share screen space.
- Return `false` from `preSolve2D` or `shouldCollide*` filters to implement one-way platforms, team filters, or trigger volumes.

Reference

- Engine (p. 361) — engine events (`onUpdate`, input, lifecycle)
- PhysicsSystem (p. 567) — collision and contact events
- AdMob (p. 610), InAppPurchase (p. 618), WebPortal (p. 626) — service events
- Script workflow: Creating Scripts (p. 199)

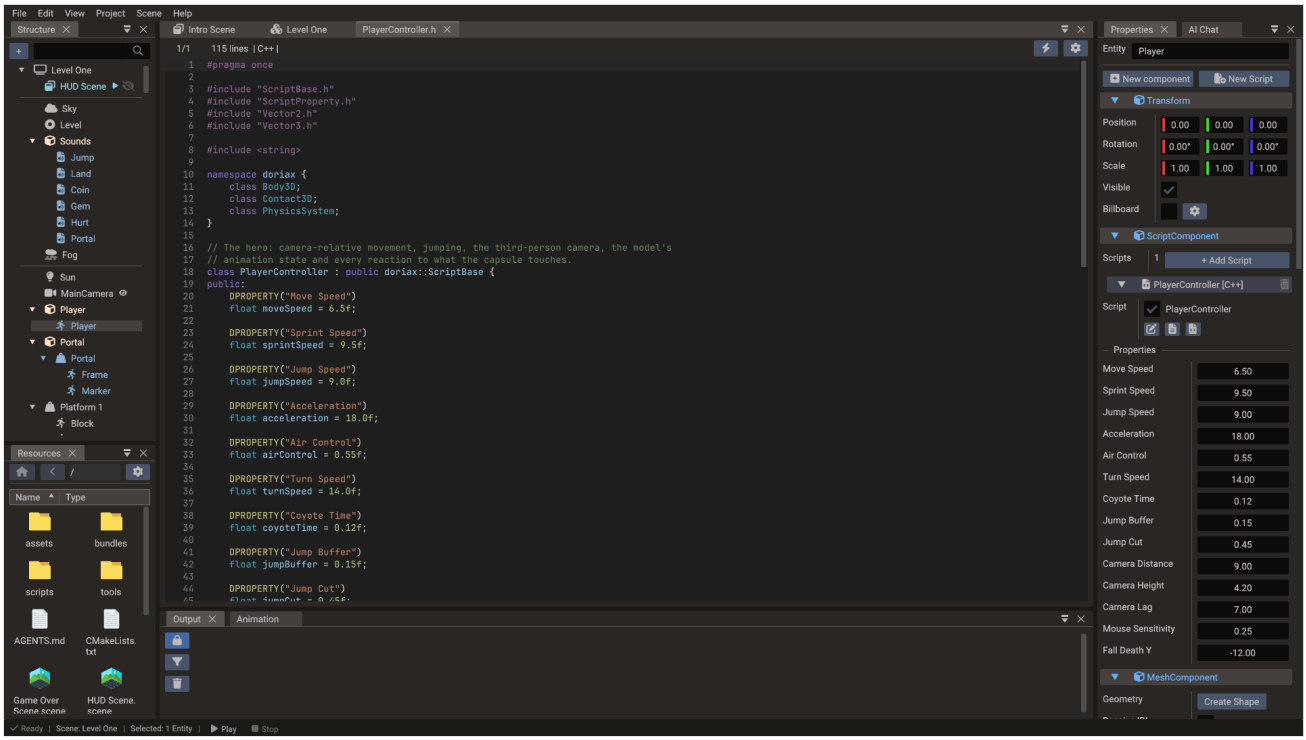
Script Properties

Script properties are designer-tunable values stored on each `ScriptEntry` and edited in the Properties window. They serialize with the scene and sync into runtime script instances on play and export.

Overview

Source	Declaration	Parsed by
C++ subclass / ScriptBase	`DPROPERTY` macro before a member	`ScriptParser` (editor)
Lua script	`properties` table in returned module	`ProjectUtils::loadLuaScriptProperties`

At runtime, `ScriptProperty::syncToMember()` (C++) or direct field injection (Lua) applies Properties values before `init()` runs.



C++ DPROPERTY macro

Defined in `engine/core/script/ScriptProperty.h`:

```
#define DPROPERTY_1(DisplayName) public:
#define DPROPERTY_2(DisplayName, Type) public: /* @DPROPERTY_TYPE: Type */
#define DPROPERTY(...) GET_DPROPERTY_MACRO(__VA_ARGS__, DPROPERTY_2, DPROPERTY_1)(__VA_ARGS__)
```

The macro expands to `public:` plus an optional `/* @DPROPERTY_TYPE: ... */` comment for the editor parser. **Place it immediately before the member variable** — no other declarations between the macro and the member.

Basic examples

```
DPROPERTY("Speed")
float speed = 5.0f;

DPROPERTY("Target Position")
doriax::Vector3 targetPosition = doriax::Vector3(0, 0, 0);

DPROPERTY("Mesh Color", Color4)
doriax::Vector4 meshColor = doriax::Vector4(1, 1, 1, 1);

DPROPERTY("Follow Target")
doriax::Object* followTarget = nullptr;
```

Syntax forms

Form	Meaning
`DPROPERTY("Display Name")`	Infer editor type from the C++ member declaration
`DPROPERTY("Display Name", Color4)`	Force an explicit editor type hint
`DPROPERTY("Display Name") /* @DPROPERTY_TYPE: Color4 */`	Equivalent annotation form

The editor regex expects this pattern:

```
DPROPERTY("Label")
Type memberName = defaultValue;
```

Type inference priority

1. Explicit second argument: `DPROPERTY("X", Color4)`
2. Comment annotation: `/* @DPROPERTY_TYPE: Color4 */`
3. C++ type inference (see table below)

Supported C++ property types

C++ declaration	Editor type	Properties widget
`bool`	Bool	Checkbox
`int`, `int32_t`, `uint32_t`, `short`, `long`	Int	Integer field
`float`, `double`	Float	Float field
`std::string`, `string`	String	Text field
`Vector2`, `doriax::Vector2`	Vector2	X/Y fields
`Vector3`, `doriax::Vector3`	Vector3	X/Y/Z fields
`Vector4`, `doriax::Vector4`	Vector4	X/Y/Z/W fields
`Vector3` + `Color`/`Color3` hint	Color3	RGB color picker
`Vector4` + `Color`/`Color4` hint	Color4	RGBA color picker
Pointer (`Object*`, `Mesh*`, `Camera*`, etc.)	EntityReference	Entity picker

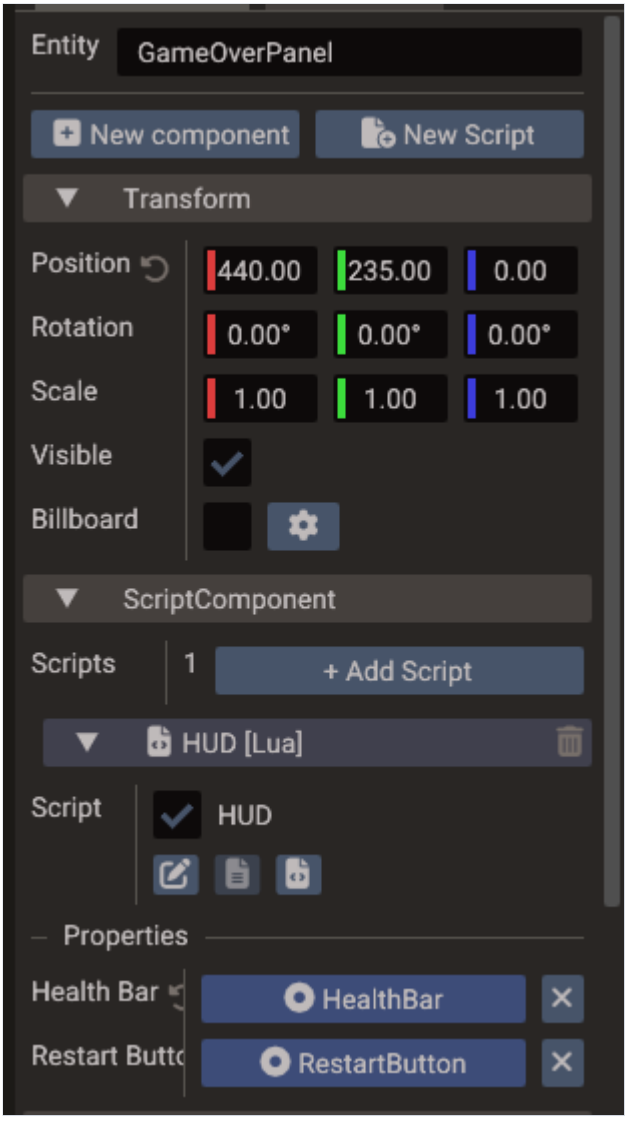
Entity reference pointers

Pointer properties store an `EntityReference { entity, sceneId }` in serialized data, **not** a raw C++ pointer. At runtime:

- **C++** — resolved when the script instance is created.
- **Lua** — Pass 2 of `initializeLuaScripts` resolves to a Lua script instance (if the target has a matching enabled script) or a typed wrapper (`Object`, `Mesh`, etc.).

`sceneId` is `0` when the target is in the script's own scene. When you pick an entity from a child scene shown inline in the Structure panel, `sceneId` holds that child scene's ID instead, and the target is looked up through `SceneManager::getScenePtr` (p. 608). That scene must already be built when the script starts, or the property stays `nil`/`nullptr`. The editor's component fields, such as joint bodies, take same-scene entities only (see Entity references (p. 105)).

Supported pointer types for entity references include all gameplay wrappers listed in `LuaBinding::pushEntityHandleByType` (Object, Mesh, Camera, Light, Sprite, Body2D, Action, etc.).



Supported default values

```
DPROPERTY("Enabled")
bool enabled = true;

DPROPERTY("Lives")
int lives = 3;

DPROPERTY("Speed")
float speed = 5.0f;

DPROPERTY("Name")
std::string name = "Player";

DPROPERTY("Offset")
doriax::Vector2 offset = doriax::Vector2(1, 2);
```

```
DPROPERTY("Target")
doriax::Vector3 target = doriax::Vector3(0, 3, 0);

DPROPERTY("Tint", Color4)
doriax::Vector4 tint = doriax::Vector4(1, 1, 1, 1);
```

Non-null pointer defaults are not supported. Use `nullptr` and assign the target in the Properties.

A `nullptr` or `NULL` default on a member that is not a pointer is ignored: the property starts at the type's default and the editor logs a warning.

Lua script properties

Declare properties in the table returned by your Lua module:

```
local Door = {
    properties = {
        { name = "openSpeed", displayName = "Open Speed", type = "float", default = 2.5 },
        { name = "startsOpen", displayName = "Starts Open", type = "bool", default = false },
        { name = "linkedDoor", displayName = "Linked Door", type = "Object" }
    }
}

function Door:init()
    print(self.openSpeed, self.startsOpen)
end

return Door
```

Lua property fields

Field	Required	Purpose
`name`	Yes	Field name on `self` (e.g. `self.openSpeed`)
`displayName`	Yes	Label shown in Properties
`type`	Yes	Type string (see table below)
`default`	Recommended	Default value when first added

Supported Lua type strings

Lua `type` value	Stored `ScriptPropertyType`
`bool`, `boolean`	Bool
`int`, `integer`	Int
`float`, `number`	Float
`string`	String
`vector2`, `vec2`	Vector2
`vector3`, `vec3`	Vector3
`vector4`, `vec4`	Vector4
`color3`	Color3
`color4`	Color4
`entity`, `entitypointer`, `pointer`	EntityReference
Any other string	EntityReference with that string as desired wrapper/script type

For entity references, the code editor picks the property type from the referenced entity: it prefers an enabled Lua script class name when present, otherwise the wrapper type detected from components (`Object`, `Mesh`, `Camera`, etc.).

Insert entity references by drag and drop

The fastest way to declare an entity reference property is to drag the entity from the **Structure panel** into the script file open in the **Code Editor**. This works for both Lua files and C++ headers, and does three things at once:

1. **Inserts the declaration** at the right place — a new entry in the Lua `properties` table, or a `DPROPERTY` line plus pointer member after the last existing `DPROPERTY` in the C++ header.
2. **Picks the type from the dropped entity** — the entity's script class name if it has an enabled script of the matching language, otherwise the most specific wrapper type detected from its components (`Mesh`, `Camera`, `Terrain`, `Button`, etc.). In C++ the editor also adds the missing `#include`, whether the type is a script class or an engine one.
3. **Assigns the value** — every entity already using this script gets the dropped entity linked as the property value, so it shows up wired in the Properties window immediately.

The property name is derived from the entity name (camelCase), with a numeric suffix if the name is already taken.

Lua (inserted)

```lua
{
    name = "mainCamera",
    displayName = "Main Camera",
    type = "Camera",
    default = nil
}
```

C++ header (inserted)

```cpp
DPROPERTY("Main Camera")
doriax::Camera* mainCamera = nullptr;
```

> **Drop on the header, not the source**
>
> For C++ scripts the declaration belongs in the header. Dropping an entity onto a `.cpp` file shows a reminder instead of inserting code.

Runtime sync

`ScriptProperty` (`engine/core/script/ScriptProperty.cpp`):

Method	Direction
`syncToMember()`	Stored Properties value → C++ member or Lua instance field
`syncFromMember()`	C++ member or Lua field → stored value (editor refresh)

C++ scripts store `memberPtr` during parsing. Lua scripts store `luaRef` to the instance table field.

Properties behavior

1. Editor parses script headers / Lua modules when a `ScriptEntry` is added or the file changes.
2. New properties are appended; existing properties with the same name and type keep their edited values.
3. Property edits go through undoable `PropertyCmd` commands.
4. Values serialize into scene YAML with the entity's `ScriptComponent`.

The integrated code editor can insert `DPROPERTY` blocks (C++) or `properties` entries (Lua) from templates.

ScriptPropertyType enum

```
enum class ScriptPropertyType {
    Bool, Int, Float, String,
    Vector2, Vector3, Vector4,
    Color3, Color4,
    EntityReference
};
```

Values are stored in a `std::variant` (`ScriptPropertyValue`).

Common mistakes

Mistake	Fix
Code between `DPROPERTY` and member	Put the macro directly above the member
Function call as default	Use literals only (`5.0f`, `"text"`, `Vector3(...)`)
`nullptr` default on a non-pointer member	Ignored with a warning, use a literal of the member's type
Missing `doriax::` prefix	Add namespace or `using namespace doriax` in header
Expecting serialized C++ pointer	Entity references store `{entity, sceneId}`
Lua property name mismatch	`name` must match `self.<name>` used in code
Changed property type	Properties may reset value; rename if you need a fresh field

Editor tooling reference

File	Role
`editor/util/ScriptParser.cpp`	Parses `DPROPERTY` from C++ headers
`editor/util/ProjectUtils.cpp`	Loads Lua `properties` tables
`editor/window/CodeEditor.cpp`	Inserts property templates
`editor/window/dialog/ScriptCreateDialog.cpp`	New script wizard with examples
`editor/Catalog.cpp`	Maps `ScriptPropertyType` to Properties widgets

Next steps

- Creating Scripts (p. 199) — attach scripts and understand lifecycle
- Events (p. 215) — react to input and gameplay with event handlers
- ScriptBase (p. 630) — C++ script base class

Input

Doriax exposes all input through the static `Input` class. It covers keyboard, mouse, touch, gamepad, and modifier state for desktop, mobile, and web targets. All polling APIs are safe to call from any script or system update.

Key and button constants

Key and button constants are based on GLFW key codes. The naming differs by language:

Language	How to access	Example
C++	`D_KEY_*` / `D_MOUSE_BUTTON_*` / `D_MODIFIER_*` preprocessor macros	`D_KEY_SPACE`
Lua	Static properties on the `Input` class (drop the `D_` prefix)	`Input.KEY_SPACE`

There is no global `D_KEY_SPACE` in Lua — always use `Input.KEY_SPACE`. The same applies to mouse buttons (`Input.MOUSE_BUTTON_LEFT`) and modifiers (`Input.MODIFIER_SHIFT`).

Keyboard

`Input.isKeyPressed(key)` returns `true` while the key is **currently held down**. It is a polling call: read it every frame inside `onUpdate`. There is no separate "down/released this frame" polling method — for one-shot press and release *edges*, subscribe to the `Engine.onKeyDown` / `Engine.onKeyUp` (p. 361) events instead.

Lua

```lua
if Input.isKeyPressed(Input.KEY_SPACE) then
    player:jump()
end

if Input.isKeyPressed(Input.KEY_A) then
    player:moveLeft()
end
```

C++

```cpp
if (Input::isKeyPressed(D_KEY_SPACE)) {
    player.jump();
}

if (Input::isKeyPressed(D_KEY_A)) {
    player.moveLeft();
}
```

Common key constants

C++ macro	Lua property	Key
`D_KEY_A` … `D_KEY_Z`	`Input.KEY_A` … `Input.KEY_Z`	Letter keys
`D_KEY_0` … `D_KEY_9`	`Input.KEY_0` … `Input.KEY_9`	Number row
`D_KEY_KP_0` … `D_KEY_KP_9`	`Input.KEY_KP_0` … `Input.KEY_KP_9`	Numpad
`D_KEY_LEFT`, `D_KEY_RIGHT`, `D_KEY_UP`, `D_KEY_DOWN`	`Input.KEY_LEFT`, …	Arrow keys
`D_KEY_SPACE`	`Input.KEY_SPACE`	Spacebar
`D_KEY_ENTER`, `D_KEY_ESCAPE`, `D_KEY_BACKSPACE`, `D_KEY_TAB`	`Input.KEY_ENTER`, …	Editing keys
`D_KEY_F1` … `D_KEY_F12`	`Input.KEY_F1` … `Input.KEY_F12`	Function keys
`D_KEY_LEFT_SHIFT`, `D_KEY_LEFT_CONTROL`, `D_KEY_LEFT_ALT`	`Input.KEY_LEFT_SHIFT`, …	Modifier keys

See Input (p. 380) for the full constant list.

On Android, the Back button and the back gesture arrive as `D_KEY_ESCAPE`, so a game that pauses or closes a menu on Escape does the same on Back. Back never closes the game by itself. To close it from the main menu, call `System::quit()` (p. 410) when Escape is pressed there.

Mouse

Like the keyboard, `Input.isMousePressed(button)` returns `true` while the button is held down. For click and release edges use the `Engine.onMouseDown` / `Engine.onMouseUp` (p. 361) events.

Lua

```lua
if Input.isMousePressed(Input.MOUSE_BUTTON_LEFT) then
    local pos = Input.getMousePosition()
    shoot(pos.x, pos.y)
end
```

C++

```cpp
if (Input::isMousePressed(D_MOUSE_BUTTON_LEFT)) {
    Vector2 pos = Input::getMousePosition();
    shoot(pos.x, pos.y);
}
```

Mouse methods

Method	Purpose
`isMousePressed(button)`	Button is currently held down
`isMouseEntered()`	Cursor is inside the canvas
`getMousePosition()`	Current cursor position in canvas coordinates
`getMouseScroll()`	Accumulated scroll wheel delta `(x, y)` since last frame

Cursor visibility, locking, and positioning

Use `Input` to read the current mouse state. Use `Engine` to control how the OS cursor behaves:

API	Purpose
`Engine.mouseMode` / `Engine::setMouseMode(MouseMode)`	Set how the cursor behaves: `NORMAL`, `HIDDEN`, `CAPTURED`, or `CONFINED`
`Engine.setMousePosition(x, y)` / `Engine::setMousePosition(x, y)`	Set the mouse position in canvas coordinates

A single mouse mode covers both cursor visibility and movement. `HIDDEN` only hides the cursor — it still moves freely. For first-person cameras or free-look controls use `CAPTURED`, which hides and locks the pointer so movement keeps generating input without letting it escape the window. `CONFINED` keeps a visible cursor trapped inside the window without switching to relative motion.

Lua

```lua
function PlayerLook:init()
    Engine.mouseMode = MouseMode.CAPTURED
    RegisterEngineEvent(self, "onMouseMove")
end

function PlayerLook:onMouseMove(x, y, mods)
    local pos = Input.getMousePosition()
    -- Compare pos with the previous frame, or use the event values directly.
end
```

C++

```
class PlayerLook : public doriax::ScriptBase {
public:
    PlayerLook(doriax::Scene* scene, doriax::Entity entity)
        : ScriptBase(scene, entity) {
        Engine::setMouseMode(MouseMode::CAPTURED);
        REGISTER_ENGINE_EVENT(onMouseMove);
    }

    ~PlayerLook() {
        UNREGISTER_ENGINE_EVENT(onMouseMove);
    }

    void onMouseMove(float x, float y, int mods) {
        Vector2 pos = Input::getMousePosition();
        // Compare pos with the previous frame, or use x/y directly.
    }
};
```

To place the cursor manually, pass logical canvas coordinates:

Lua

```
Engine.setMousePosition(Engine.canvasWidth * 0.5, Engine.canvasHeight * 0.5)
```

C++

```
Engine::setMousePosition(
    Engine::getCanvasWidth() * 0.5f,
    Engine::getCanvasHeight() * 0.5f
);
```

Desktop GLFW builds also move the OS cursor. On platforms that do not expose cursor warping, the engine updates its internal input position and the next real mouse event may replace it.

Mouse button constants

C++ macro	Lua property	Button
`D_MOUSE_BUTTON_LEFT`	`Input.MOUSE_BUTTON_LEFT`	Primary / left button
`D_MOUSE_BUTTON_RIGHT`	`Input.MOUSE_BUTTON_RIGHT`	Secondary / right button
`D_MOUSE_BUTTON_MIDDLE`	`Input.MOUSE_BUTTON_MIDDLE`	Middle button / scroll wheel click

Touch

Touch input is the primary interaction model on mobile and web. Use the touch APIs for multi-touch aware code; configure mouse/touch mirroring in Engine settings if you want single-touch to also fire mouse events.

Lua

```
if Input.isTouch() then
    local n = Input.numTouches()
    for i = 0, n - 1 do
        local pos = Input.getTouchPosition(i)
        handleTap(pos.x, pos.y)
    end
end
```

C++

```
if (Input::isTouch()) {
    int n = Input::numTouches();
    for (int i = 0; i < n; i++) {
        Vector2 pos = Input::getTouchPosition(i);
        handleTap(pos.x, pos.y);
    }
}
```

Touch methods

Method	Purpose
`isTouch()`	True while at least one touch point is active
`numTouches()`	Number of active touch points
`getTouchPosition(index)`	Position of a touch point by index
`getTouches()`	Array of all active touch records

On-screen controls

`System::isTouchDevice()` (p. 408) tells whether the game runs on a phone or tablet, so it can show its own buttons from the start. Switch them on after a touch and off after a key press too, so a phone with a keyboard or a touch-screen laptop gets the right controls. Ignore Escape for this, because Android's Back button sends it. To try them in the editor, turn on touch simulation (p. 88) before pressing **Play**.

Movement buttons are held while other buttons are pressed, so read them from the active touch points each frame instead of using UI buttons: a UI Button (p. 506) is released as soon as any finger lifts.

C++

```
// true while any finger is on the image
bool isTouched(Image* button) {
    Vector3 pos = button->getWorldPosition();
    Vector3 scale = button->getWorldScale();
    for (const Touch& touch : Input::getTouches()) {
        if (touch.position.x >= pos.x && touch.position.x <= pos.x + button->getWidth() * scale.x &&
            touch.position.y >= pos.y && touch.position.y <= pos.y + button->getHeight() * scale.y) {
            return true;
        }
    }
    return false;
}
```

Gamepad

Gamepad (controller) input is available on desktop (Windows, Linux, macOS), mobile (Android, iOS), and web. Buttons and axes follow the standard Xbox-style layout, so a single code path works across controllers and platforms.

Each connected controller has a numeric **id**. Poll a specific controller by id, or subscribe to the `Engine.onGamepad*` (p. 362) events for connect/disconnect and button/axis edges.

Lua

```lua
-- Poll the first controller (id 0)
if Input.isGamepadConnected(0) then
    if Input.isGamepadButtonPressed(0, Input.GAMEPAD_BUTTON_A) then
        player:jump()
    end

    local moveX = Input.getGamepadAxis(0, Input.GAMEPAD_AXIS_LEFT_X)
    local moveY = Input.getGamepadAxis(0, Input.GAMEPAD_AXIS_LEFT_Y)
    player:move(moveX, moveY)
end
```

C++

```cpp
if (Input::isGamepadConnected(0)) {
    if (Input::isGamepadButtonPressed(0, D_GAMEPAD_BUTTON_A)) {
        player.jump();
    }

    float moveX = Input::getGamepadAxis(0, D_GAMEPAD_AXIS_LEFT_X);
    float moveY = Input::getGamepadAxis(0, D_GAMEPAD_AXIS_LEFT_Y);
    player.move(moveX, moveY);
}
```

Enumerating connected controllers

Gamepad ids are **sparse** — after a controller disconnects, the remaining ids can have gaps (for example ids `0` and `2` with `1` missing). `numGamepads()` is a *count*, not the highest id, so never loop `for i = 0, numGamepads() - 1` and treat `i` as an id. Instead map a dense index to a real id with `getGamepadId(index)`:

Lua

```lua
for i = 0, Input.numGamepads() - 1 do
    local id = Input.getGamepadId(i)
    if Input.isGamepadButtonPressed(id, Input.GAMEPAD_BUTTON_START) then
        pauseGame()
    end
end
```

C++

```cpp
for (size_t i = 0; i < Input::numGamepads(); i++) {
    int id = Input::getGamepadId(i);
    if (Input::isGamepadButtonPressed(id, D_GAMEPAD_BUTTON_START)) {
        pauseGame();
    }
}
```

Events

For one-shot press/release and connect/disconnect notifications, subscribe to the engine events instead of polling:

Lua

```lua
function Controller:init()
    RegisterEngineEvent(self, "onGamepadConnect")
    RegisterEngineEvent(self, "onGamepadButtonDown")
end

function Controller:onGamepadConnect(gamepad)
    print("Controller connected: " .. Input.getGamepadName(gamepad))
end

function Controller:onGamepadButtonDown(gamepad, button)
    if button == Input.GAMEPAD_BUTTON_A then
        player:jump()
    end
end
```

C++

```cpp
class Controller : public doriax::ScriptBase {
public:
    Controller(doriax::Scene* scene, doriax::Entity entity)
        : ScriptBase(scene, entity) {
        REGISTER_ENGINE_EVENT(onGamepadConnect);
        REGISTER_ENGINE_EVENT(onGamepadButtonDown);
    }

    ~Controller() {
        UNREGISTER_ENGINE_EVENT(onGamepadConnect);
        UNREGISTER_ENGINE_EVENT(onGamepadButtonDown);
    }

    void onGamepadConnect(int gamepad) { /* ... */ }

    void onGamepadButtonDown(int gamepad, int button) {
        if (button == D_GAMEPAD_BUTTON_A) player.jump();
    }
};
```

The full event list is `onGamepadConnect`, `onGamepadDisconnect`, `onGamepadButtonDown`, `onGamepadButtonUp`, and `onGamepadAxisMove` — see Engine (p. 362).

Gamepad methods

Method	Purpose
`isGamepadConnected(id)`	True while a controller with this id is connected
`getGamepadName(id)`	Human-readable controller name
`isGamepadButtonPressed(id, button)`	Button is currently held down
`getGamepadAxis(id, axis)`	Current axis value, `-1.0`…`1.0`
`numGamepads()`	Number of connected controllers
`getGamepadId(index)`	Real id of the `index`-th connected controller (`-1` if out of range)

Gamepad button constants

Buttons use the standard Xbox layout. In Lua, replace the `D_GAMEPAD_BUTTON_` prefix with `Input.GAMEPAD_BUTTON_`. PlayStation-style aliases map to the same values.

C++ constant	Value	Xbox	PlayStation alias
`D_GAMEPAD_BUTTON_A`	0	A	`D_GAMEPAD_BUTTON_CROSS`
`D_GAMEPAD_BUTTON_B`	1	B	`D_GAMEPAD_BUTTON_CIRCLE`
`D_GAMEPAD_BUTTON_X`	2	X	`D_GAMEPAD_BUTTON_SQUARE`
`D_GAMEPAD_BUTTON_Y`	3	Y	`D_GAMEPAD_BUTTON_TRIANGLE`

C++ constant	Value	Xbox	PlayStation alias
`D_GAMEPAD_BUTTON_LEFT_BUMPER`	4	LB	L1
`D_GAMEPAD_BUTTON_RIGHT_BUMPER`	5	RB	R1
`D_GAMEPAD_BUTTON_BACK`	6	View / Back	Share
`D_GAMEPAD_BUTTON_START`	7	Menu / Start	Options
`D_GAMEPAD_BUTTON_GUIDE`	8	Guide	PS
`D_GAMEPAD_BUTTON_LEFT_THUMB`	9	Left stick click	L3
`D_GAMEPAD_BUTTON_RIGHT_THUMB`	10	Right stick click	R3
`D_GAMEPAD_BUTTON_DPAD_UP`	11	D-pad up	
`D_GAMEPAD_BUTTON_DPAD_RIGHT`	12	D-pad right	
`D_GAMEPAD_BUTTON_DPAD_DOWN`	13	D-pad down	
`D_GAMEPAD_BUTTON_DPAD_LEFT`	14	D-pad left	

Gamepad axis constants

Axis values range from `-1.0` to `1.0`. In Lua, replace `D_GAMEPAD_AXIS_` with `Input.GAMEPAD_AXIS_`.

C++ constant	Value	Range
`D_GAMEPAD_AXIS_LEFT_X`	0	Left stick, `-1` left … `+1` right
`D_GAMEPAD_AXIS_LEFT_Y`	1	Left stick, `-1` up … `+1` down
`D_GAMEPAD_AXIS_RIGHT_X`	2	Right stick, `-1` left … `+1` right
`D_GAMEPAD_AXIS_RIGHT_Y`	3	Right stick, `-1` up … `+1` down
`D_GAMEPAD_AXIS_LEFT_TRIGGER`	4	Left trigger, `-1` released … `+1` pressed
`D_GAMEPAD_AXIS_RIGHT_TRIGGER`	5	Right trigger, `-1` released … `+1` pressed

> **Stick and trigger conventions**
>
> The Y axes are **down-positive** (up is `-1`), matching screen coordinates. **Triggers rest at `-1`** and reach `+1` when fully pressed — a released trigger reads `-1`, not `0`, and this holds from the moment a controller connects. Apply a small dead zone to the sticks (for example, ignore values under `0.15`) to avoid drift from analog noise.

Modifiers

Modifier key state is available for keyboard shortcuts:

`getModifiers()` returns a bitmask, so test individual modifiers with bitwise AND.

Lua

```lua
local mods = Input.getModifiers()
if (mods & Input.MODIFIER_CONTROL) ~= 0 then
    -- Ctrl is held
end
```

C++

```cpp
int mods = Input::getModifiers();
if (mods & D_MODIFIER_CONTROL) {
    // Ctrl is held
}
```

Modifier constants (C++ `D_MODIFIER_*` / Lua `Input.MODIFIER_*`): `SHIFT`, `CONTROL`, `ALT`, `SUPER`, `CAPS_LOCK`, and `NUM_LOCK`.

Mouse and touch mirroring

Configure the engine to mirror mouse events as touch events or vice versa. The flags are `callMouseInTouchEvent` (fire mouse callbacks from touch input) and `callTouchInMouseEvent` (fire touch callbacks from mouse input):

Lua

```lua
Engine.callMouseInTouchEvent = true   -- touch input also fires mouse callbacks
Engine.callTouchInMouseEvent = true   -- mouse input also fires touch callbacks
```

C++

```cpp
Engine::setCallMouseInTouchEvent(true);   // touch input also fires mouse callbacks
Engine::setCallTouchInMouseEvent(true);   // mouse input also fires touch callbacks
```

This simplifies cross-platform code — write one input path that works for both desktop and mobile.

UI input

UI widgets consume pointer and touch events automatically. Once a script focuses a button, the UI also takes the arrow keys, Tab, Enter, Space, the d-pad, the left stick, and the gamepad A button, so menus work with a keyboard or a controller. See Keyboard and gamepad navigation (p. 295).

The events a UI widget uses don't reach the gameplay input handlers (so the player doesn't shoot when tapping a button), because `ignoreEventsHandledByUI` is on by default. Turn it off to let the handlers see them too:

Lua

```lua
Engine.ignoreEventsHandledByUI = false
```

C++

```cpp
Engine::setIgnoreEventsHandledByUI(false);
```

Virtual keyboard (mobile / web)

On platforms without a hardware keyboard, open the system virtual keyboard for text input:

Lua

```lua
System.showVirtualKeyboard("")   -- the text argument is required in Lua
System.hideVirtualKeyboard()
```

C++

```cpp
System::showVirtualKeyboard();   // text argument defaults to L""
System::hideVirtualKeyboard();
```

Cross-platform advice

- Always provide both keyboard/mouse and touch code paths.
- Use the named key constants (`Input.KEY_*` in Lua, `D_KEY_*` in C++) instead of raw numbers so the code stays readable.
- Avoid relying on platform-specific keys (e.g. Windows key, macOS Command) for core gameplay mechanics.
- Test canvas/input coordinate scaling together — a touch at screen position (x, y) maps to logical canvas coordinates depending on the active scaling mode.

See also

- Input (p. 380) — full API reference with all constants
- System (p. 407) — virtual keyboard and platform utilities
- User Interface (p. 292) — UI event handling

Resources & Assets

Doriax projects combine editor-managed resources with runtime pools and file helpers. Media assets are ordinary files that the runtime opens directly, both in the editor and in an exported game. Export copies them unchanged or places their bytes in an optional native resource pack, compiles shaders for the target backend, and turns scenes and bundles into generated C++.

Asset types

These are read from a file path by the runtime, on demand, at any point during gameplay:

Asset type	Formats	Loaded by
Textures	PNG, JPG, TGA, BMP, PSD, HDR, SVG	`Texture`, texture pools
3D Models	GLTF, GLB, OBJ, FBX (converted)	`Model`, `MeshSystem`
Audio	OGG, WAV, MP3, FLAC	`SoundPool`, `Sound`
Fonts	TTF, OTF, TTC	Font pool, `Text` components
Shaders	`.sdat` shader data (engine format)	`ShaderPool`, `RenderSystem`

There is no import or preprocessing step: the `.glb` or `.png` you export from your content tool supplies the same bytes the runtime reads. A native export may package those original bytes into `resources.pak`, but it does not convert the asset format or create a reprocessed copy.

FBX is the one exception. The editor converts each FBX file to a binary glTF, and the runtime loads that conversion wherever the `.fbx` path is used. The FBX file itself is never modified. See 3D Graphics — FBX models (p. 251).

Authoring formats

Scenes (`.scene`), bundles (`.bundle`), and materials (`.material`) are YAML files that only the **editor** reads. Export converts them into generated C++ that is compiled into your game, so the runtime never parses them:

Format	Becomes	Used at runtime through
`.scene`	A scene factory function	`SceneManager` (by name or ID)
`.bundle`	A bundle factory function	`BundleManager` (by name or ID)
`.material`	Material values baked into the generated scene or bundle	`Material` on a mesh submesh

Because these become code rather than data, a scene or entity hierarchy cannot be loaded from a file the executable was not built with. Spawning hierarchies on demand is done with `BundleManager.createBundle` (p. 598), which works for any bundle that existed at build time — see Bundles (p. 188). Textures referenced by a material are shipped and loaded like any other texture.

Path roots

A path handed to the engine is resolved against a root, chosen by its scheme:

Scheme	Root	Holds
`asset://`, or no scheme	Assets directory	Textures, models, sounds, fonts
`lua://`	Lua directory	Lua modules and script entries
`shader://`	Compiled shader directory	`.sdat` shader data
`data://`	Writable user data directory	Saves and settings

Both project directories are set in Project Settings (p. 52) and default to the project root. Paths are stored relative to their root, so `sprites/hero.png` means `<assets directory>/sprites/hero.png` in the editor and `assets/sprites/hero.png` next to an exported executable. Absolute paths are used as given.

A leading `..` never escapes a root: it is dropped while the path is normalized, so keep every referenced file inside the assets directory.

Native resource packs preserve these logical roots. Packed lookups support asset-relative paths, `asset://`, and `lua://`; absolute paths, `data://`, and other schemes continue to use the filesystem. See Native resource pack (p. 160) for export layouts and limitations.

Loading assets at runtime

Most high-level objects accept a file path and load the resource automatically:

Lua

```lua
-- Texture loaded by a Sprite
sprite = Sprite(scene)
sprite:setTexture("sprites/hero.png")

-- Model loaded from GLTF
model = Model(scene)
model:loadGLTF("models/hero.gltf")

-- Sound loaded by Sound object
sfx = Sound(scene)
sfx:loadSound("audio/jump.ogg")
```

C++

```cpp
Sprite sprite(&scene);
sprite.setTexture("sprites/hero.png");

Model model(&scene);
model.loadGLTF("models/hero.gltf");

Sound sfx(&scene);
sfx.loadSound("audio/jump.ogg");
```

These calls are valid at any time, not just while a scene is being built, so models, textures, and sounds can be swapped in during gameplay. Set `Engine.asyncLoading = true` to move the parsing and decoding onto worker threads and keep the frame from stalling; see Threading & Async Loading (p. 333).

Vector images (SVG)

`.svg` files are treated as textures: they are **rasterized to RGBA when loaded**, so they work anywhere a raster texture does — sprites, UI images, and material maps. Only the features of the bundled rasterizer are drawn (basic shapes, paths, solid and gradient fills, and strokes); there is no text or filter/effect support, so convert text to paths when authoring.

By default an SVG is rasterized at its intrinsic size (the `width`/`height` or `viewBox` in the file). Because a vector is resolution-independent, you can rasterize it larger for crisp results on high-DPI displays or when the image is drawn bigger than its native size. The rasterization scale is set per texture (`2.0` = twice the intrinsic resolution) and has no effect on raster images:

Lua

```lua
local data = TextureData()
data.svgScale = 4
data:loadTextureFromFile("ui/icon.svg")  -- a 24x24 SVG -> 96x96 texture
```

C++

```
TextureData data;
data.setSVGScale(4.0f);
data.loadTextureFromFile("ui/icon.svg");  // a 24x24 SVG -> 96x96 texture
```

The scale can also be set directly on a `Texture` reference — `svgScale` in Lua, `setSvgScale()` in C++ (see Texture (p. 740)):

Lua

```
local icon = Texture("ui/icon.svg")
icon.svgScale = 4  -- a 24x24 SVG -> 96x96 texture
```

C++

```
Texture icon("ui/icon.svg");
icon.setSvgScale(4.0f);  // a 24x24 SVG -> 96x96 texture
```

The property is saved with the scene and applied in exported projects, and because the scale is part of the texture's identity, different slots can use the same SVG at different resolutions. This is what the editor's per-slot **SVG Scale** control edits; see Properties — texture fields (p. 90). The legacy path form `"ui/icon.svg?svgScale=4"` is still accepted and absorbed into the property when the path is set. The rasterized size is capped to a GPU-friendly limit, so very large scales are rejected rather than allocated.

Material files

`.material` files store PBR parameters outside scene YAML so many meshes can reference one definition. Each file lists colour factors, metallic/roughness scalars, and relative texture paths — the same fields as the Material (p. 734) struct.

In the editor, drag a material preview from **Properties** into the **Resources Browser** to create a file, or drag an existing `.material` from the browser onto a mesh to link it. Linked materials reload when the file changes. See Resources Browser — Material files (p. 108).

Linking is an editor-side relationship: at export the material's values are written into the generated scene or bundle code, and the textures it references are copied like any other asset. The exported game does not read the `.material` file, so replacing one in a shipped build has no effect — re-export to pick up a change.

Runtime pools

Pools avoid loading the same resource data twice and centralize ownership. Objects that use the same file path share the underlying data.

Pool	Managed resource
`TextureDataPool`	Raw decoded texture data (pixels)
`TexturePool`	GPU-side texture handles, one per image and filter/wrap combination
`ModelPool`	Parsed 3D model and skeleton data
`ShaderPool`	Compiled shader programs
`SoundPool`	Loaded audio data
`FontPool`	Parsed font data

Pools are managed internally. You normally interact with them through high-level objects (`Sprite`, `Model`, `Sound`), but you can query pool state or pre-warm a pool for level loading.

For C++ teardown code, `Engine::clearUnusedPools()` releases entries that are owned only by their pool and preserves resources still referenced by active scenes or engine objects. `Engine::clearPools()` empties every pool and destroys pooled GPU textures and shaders, so reserve it for engine or graphics-view shutdown. If

asynchronous model loads may still be running, call `MeshSystem::cancelAllAsyncModelLoads()` before either cleanup operation; cancellation waits for the active worker tasks to finish before it returns.

The editor performs the safe sequence automatically when you switch projects: it quiesces project-specific background work, destroys the old scenes, and then clears unreferenced pool entries.

See Engine (p. 372) and MeshSystem (p. 593) for the complete C++ API.

File I/O

`FileData` is the abstract base for byte-level access; `File` reads and writes files on disk and `Data` wraps an in-memory buffer. All three share `readString`, `writeString`, `read8/16/32`, `seek`, `pos`, `length`, and `eof`.

In a native build with resource packing enabled, `Data::open()` checks `resources.pak` before falling back to disk. Lua `require()`, script entries, textures, sounds, and GLTF extra files all go through that path. `FileData::newFile()` also returns an in-memory `Data` object when the requested resource is packed, even if a handle was requested. A directly constructed `File` still requires a real file and therefore cannot open packed assets or Lua files. Editor Play mode never opens a pack.

Lua

```lua
-- Read a file into memory (second argument: open for writing)
local f = File("saves/player.dat", false)
local contents = f:readString()
f:close()

-- Write
local out = File("saves/player.dat", true)
out:writeString(contents)
out:flush()
out:close()
```

C++

```cpp
File file("saves/player.dat", false);
std::string contents = file.readString();
file.close();
```

A `File` that can't be opened logs an error and then acts like an empty file, so check what `open` returns when a missing file matters. See File — Failed opens (p. 705).

File paths follow the same roots as the rest of the engine: a relative path resolves against the assets directory, and absolute paths are used as given. On mobile and web, write-accessible directories are separate from read-only resource paths — use `System.getUserDataPath()` (`data://`) for player save data, and `System.getAssetPath()` for read-only bundled assets.

User settings

`UserSettings` persists key/value pairs using each platform's native preferences storage. It is suited for player preferences, audio/video settings, and small game state. The API is typed — use the matching `set…ForKey` / `get…ForKey` pair for `Bool`, `Integer`, `Long`, `Float`, `Double`, `String`, or `Data`:

Lua

```lua
UserSettings.setFloatForKey("volume", 0.8)
UserSettings.setBoolForKey("fullscreen", true)

local vol = UserSettings.getFloatForKey("volume", 1.0)  -- second arg is the default
UserSettings.removeKey("obsoleteSetting")
```

C++

```
UserSettings::setFloatForKey("volume", 0.8f);
UserSettings::setBoolForKey("fullscreen", true);

float vol = UserSettings::getFloatForKey("volume", 1.0f);
```

Keep heavy save data (inventory, world state) in custom binary or JSON files. Reserve `UserSettings` for small configuration.

Asset path best practices

Guideline	Reason
Use lowercase file names	Avoids case-sensitivity issues on Linux and Android
No spaces in paths	Prevents build and script path parsing issues
Keep source and generated output separate	Generated export data should not be committed to version control
Use relative paths from the assets directory	Paths remain valid across machines and in exported builds
Keep referenced files inside the assets directory	A path outside it cannot be stored or shipped
Prefer GLTF for animated 3D assets	Carries mesh, materials, skins, node hierarchy, animations, and morph targets
Compress textures for mobile and web	Keeps bundle sizes manageable

Export and asset packaging

At export time the editor copies the **contents** of the assets and Lua directories into the `assets` and `lua` folders of the output. With the experimental **Native Resource Pack** project setting enabled, native exports then combine those two trees into `resources.pak` and remove the loose contents. Editor Play mode keeps using the source directories. Shaders are compiled for the target platform. The source project folder is not modified in either case.

Scene and bundle YAML is not needed by the exported game, since export has already turned it into compiled code. Those files may still appear inside the output `assets` folder when your assets directory is the project root, because the copy takes the directory as it is; point the assets directory at a dedicated subfolder to keep authoring files out of the shipped build.

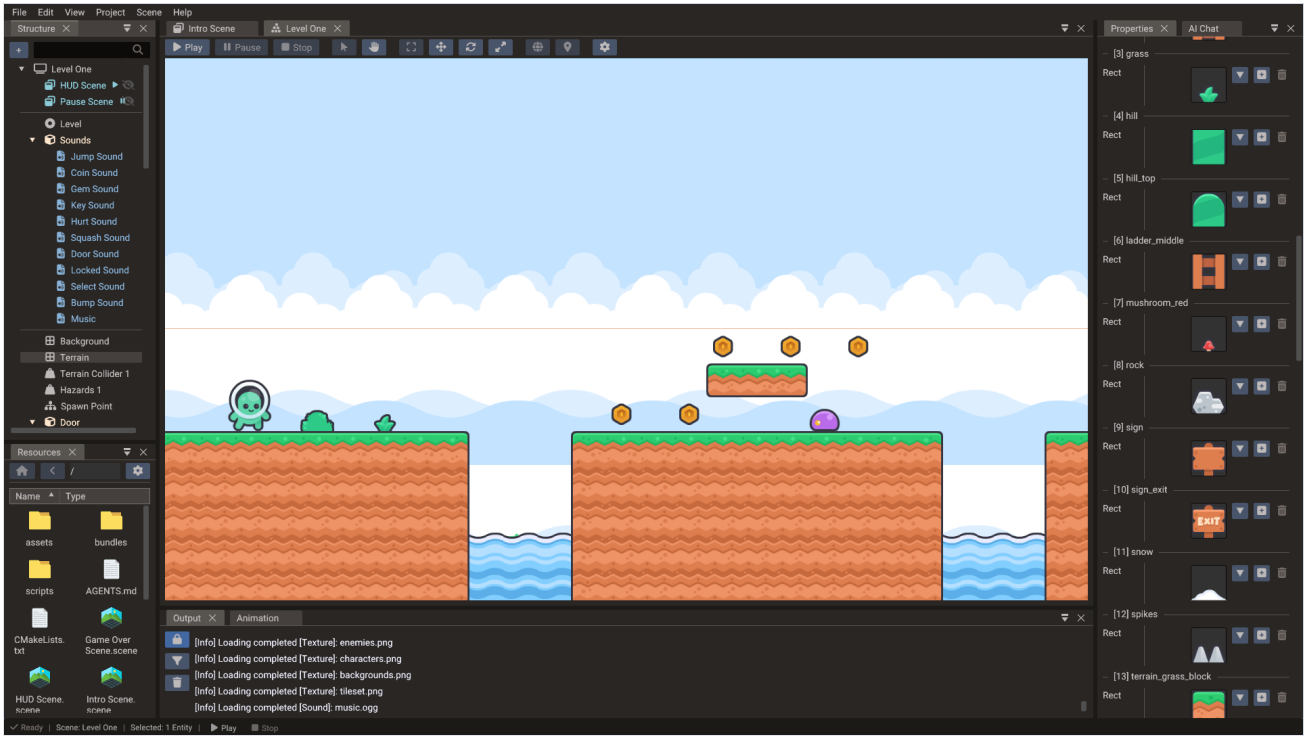
The pack is an organization and light-obfuscation feature, not encryption. Packed data is opened in memory, and the complete pack is limited to 2 GiB. See Export Window (p. 160) for supported export modes, exact output locations, reserved names, and runtime restrictions.

See also

- Texture (p. 738)
- Model (p. 450)
- Sound (p. 475)
- Data (p. 703)
- UserSettings (p. 707)
- BundleManager (p. 597)
- Threading & Async Loading (p. 333)
- Resources Browser (p. 106)

2D Graphics

Doriax has full support for 2D game development with sprites, tilemaps, primitive shapes, and dedicated 2D lighting with shadows, backed by the same ECS used for 3D.



Polygons

The `Polygon` object lets you build arbitrary 2D shapes from vertices. This is the simplest way to draw something on screen.

Lua

```lua
scene = Scene()
triangle = Polygon(scene)

triangle:addVertex(0, -100)
triangle:addVertex(-50, 50)
triangle:addVertex(50, 50)

triangle.position = Vector3(300, 300, 0)
triangle:setColor(0.6, 0.2, 0.6, 1)

Engine.setCanvasSize(1000, 480)
Engine.setScene(scene)
```

C++

```cpp
#include "Doriax.h"
using namespace doriax;
#include "Polygon.h"

Scene scene;
Polygon triangle(&scene);

DORIAX_INIT void init() {
    triangle.addVertex(0, -100);
    triangle.addVertex(-50, 50);
    triangle.addVertex(50, 50);
```

```
    triangle.setPosition(Vector3(300, 300, 0));
    triangle.setColor(0.6, 0.2, 0.6, 1);

    Engine::setCanvasSize(1000, 480);
    Engine::setScene(&scene);
}
```

Sprites

Sprites display images on screen and are the building blocks of most 2D games. In the editor, the **sprite slicer** tool cuts a sprite sheet into individual frames you can use for animation.

Common sprite operations include:

Operation	C++ method
Set size	`setSize(width, height)`
Set texture region	`setTextureRect(rect)`
Choose pivot	`setPivotPreset(preset)`
Register animation frame	`addFrame(name, x, y, width, height)`
Play frame range	`startAnimation(startFrame, endFrame, interval, loop)`

Unlit sprites that share a texture are drawn together, and a sprite that doesn't change keeps its batch data between frames. Keep sprites that use the same atlas next to each other in the hierarchy so they batch. See Sprite batching (p. 273).

Tilemaps

Tilemaps let you compose large 2D worlds from reusable tiles. Use the **tileset slicer** in the editor to cut a tileset image into tiles, then paint them into a tilemap in the scene view.

Tilemaps are designed for large grids of repeated texture regions. Keep individual tiles consistent in size, and split very large worlds into scenes or chunks so editing and export remain fast. For collision, give the tilemap a 2D Physics Body with a shape built from its tiles, see Tileset Slicer (p. 121). In the editor, turn on **Snap tile** in the scene view settings to snap placed tiles to their own size so they group edge-to-edge — see Scene View (p. 86).

Rendering cost follows what the camera sees, not the size of the map: tiles are batched by texture and, on larger maps, grouped into spatial chunks that are frustum-culled individually, so a world far bigger than the screen only submits the chunks in view. A single tilemap renders up to 16 383 tiles — use several tilemaps beyond that.

2D lighting

2D scenes have their own lighting model, separate from the 3D PBR path: a **Light2D** is a point light living in the scene's XY plane with a radius falloff, designed for the classic 2D look — torches, lamps, day/night moods.

Light from every 2D light **adds** on top of the scene's **2D ambient light**. Ambient defaults to full white, so a scene looks unchanged until you dim it — that darkness is what makes lights visible:

Lua

```lua
scene:setAmbientLight2D(0.15, Vector3(1, 1, 1))  -- dark scene

local torch = Light2D(scene)
torch.position = Vector3(300, 200, 0)
torch:setRange(250)
torch:setColor(1.0, 0.8, 0.5)
torch:setIntensity(1.2)
```

C++

```
scene.setAmbientLight2D(0.15f, Vector3(1, 1, 1)); // dark scene

Light2D torch(&scene);
torch.setPosition(Vector3(300, 200, 0));
torch.setRange(250);
torch.setColor(1.0f, 0.8f, 0.5f);
torch.setIntensity(1.2f);
```

Key light properties:

Property	Effect
`range`	Radius in world units; nothing beyond it is lit
`falloff`	Attenuation exponent — `1` linear, higher values concentrate light near the center
`height`	Virtual Z height of the light above the 2D plane, used for normal maps (see below)
`intensity` / `color`	Brightness and tint, added per light

Sprites, tilemaps, and mesh polygons receive 2D light automatically (controlled by the **Receive Lights** flag on their Mesh component). UI objects are not affected. Up to 16 2D lights can be active at once. In the editor, add one from the create menu under **2D → 2D Light** — selecting it shows its range circle in the scene view.

Normal maps on sprites

Set a **normal texture** on a sprite's material and give the light a `height` greater than zero: the light then has a direction relative to the surface, and the normal map produces relief shading that shifts as the light moves. With `height = 0` the light is purely radial and normals are ignored.

2D shadows

Shadows are cast by **Occluder2D** components. Each shadow-enabled Light2D renders the occluders around it into a small shadow map, producing radial shadows with optional soft edges.

An occluder has two shapes:

Shape	Outline
`AUTO_QUAD`	The bounds of the mesh on the **same entity** — add the component to a sprite and it casts a shadow matching the sprite quad
`POLYGON`	A custom point list in local space, editable in the editor or via code

Lua

```
torch.shadows = true
torch.shadowSoftness = 3.0

local crate = Sprite(scene)
crate:setTexture("crate.png")
crate:setSize(96, 96)
crate:getOccluder2D() -- AUTO_QUAD outline from the sprite bounds

local wall = Occluder2D(scene)
wall.position = Vector3(500, 200, 0)
wall:addVertex(-60, -20)
wall:addVertex(60, -20)
wall:addVertex(60, 20)
wall:addVertex(-60, 20)
```

C++

```
torch.setShadows(true);
torch.setShadowSoftness(3.0f);

Sprite crate(&scene);
crate.setTexture("crate.png");
crate.setSize(96, 96);
crate.getOccluder2D(); // AUTO_QUAD outline from the sprite bounds

Occluder2D wall(&scene);
wall.setPosition(Vector3(500, 200, 0));
wall.addVertex(-60, -20);
wall.addVertex(60, -20);
wall.addVertex(60, 20);
wall.addVertex(-60, 20);
```

Shadow behavior:

- Each light's shadow darkens **only that light's** contribution — ambient light and other lights still reach the shadowed area.
- `shadowSoftness` on the light widens the penumbra (`0` = hard edges); `shadowBias` fixes self-shadowing artifacts.
- The scene-wide **filter quality** (`Scene::setShadow2DQuality`) controls how smooth the penumbra looks: `NONE`, `LOW` (default), `MEDIUM`, or `HIGH`. Raise it if wide penumbras show banding — in the editor it's the **Filter Quality** combo under the scene's **2D Shadows** settings.
- Meshes opt out of receiving 2D shadows with the **Receive Shadows** flag.

In the editor, create one from **2D → 2D Occluder** (a standalone polygon), or add an **Occluder2D component** to an existing sprite for an `AUTO_QUAD` outline. Polygon points can be dragged directly in the scene view (p. 86) or edited as a list in Properties.

In code, `Sprite::getOccluder2D()` and Lua's `sprite:getOccluder2D()` are the convenience path for adding that `AUTO_QUAD` occluder directly to a sprite entity.

For how the technique works under the hood, see Rendering Pipeline — 2D lighting (p. 261).

Screen scaling

For 2D games it is important to decide how the canvas scales to different screen sizes and aspect ratios. Doriax provides scaling modes so your game looks correct across devices and window sizes — configure the canvas size and scaling behavior through the `Engine` API.

Lua

```
Engine.setCanvasSize(1000, 480)
Engine.scalingMode = Scaling.FITWIDTH   -- scalingMode is a property in Lua
```

C++

```
Engine::setCanvasSize(1000, 480);
Engine::setScalingMode(Scaling::FITWIDTH);
```

Scaling mode	Behavior
`FITWIDTH`	Match the configured canvas width and adjust height
`FITHEIGHT`	Match the configured canvas height and adjust width
`LETTERBOX`	Preserve aspect ratio with empty bands if necessary
`CROP`	Preserve aspect ratio and crop overflow
`STRETCH`	Stretch to the output size
`NATIVE`	Use the native output size

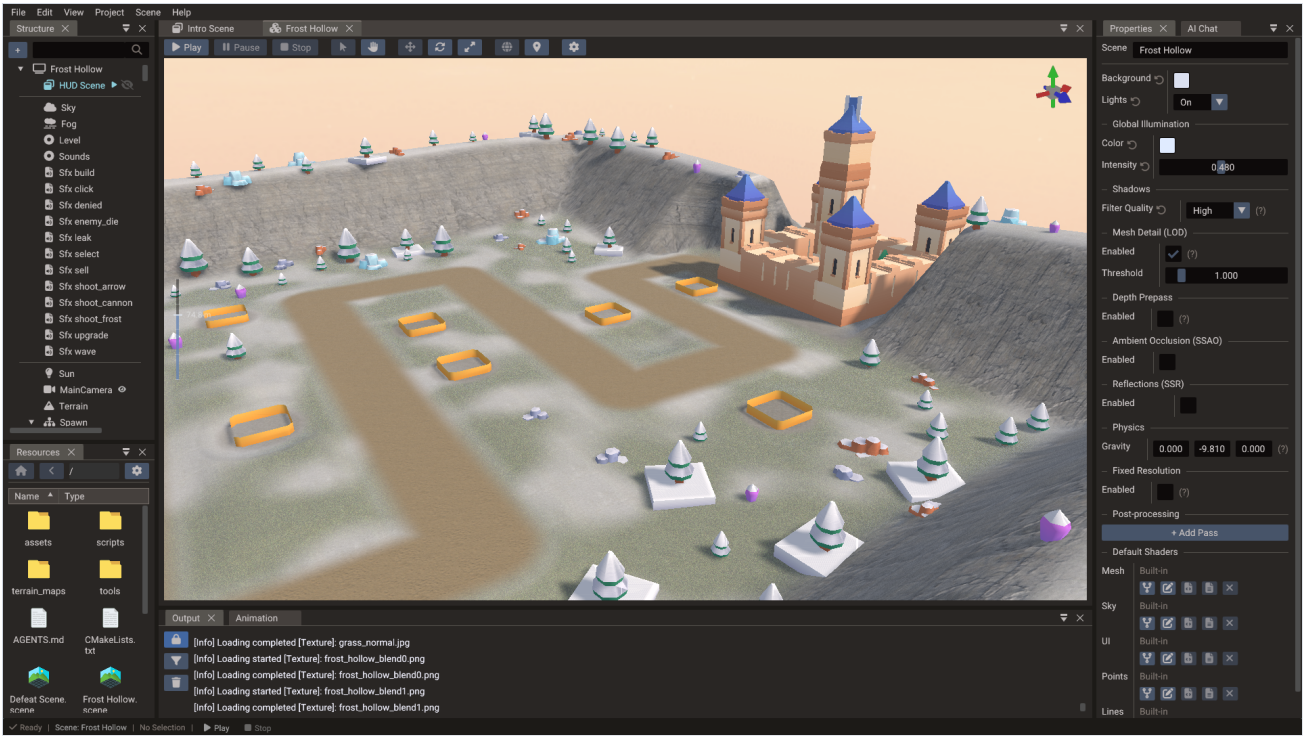
For a full guide to scaling — including the scene-level **fixed resolution** setting that renders the game at a fixed internal size and upscales it (ideal for pixel-art games), see Multiple Resolutions (p. 288).

Next steps

For 3D rendering, see 3D Graphics (p. 248). To add interactions, see Physics (p. 315).

3D Graphics

Doriax provides a complete 3D pipeline with model loading, physically-based rendering, dynamic lighting, and environment effects.

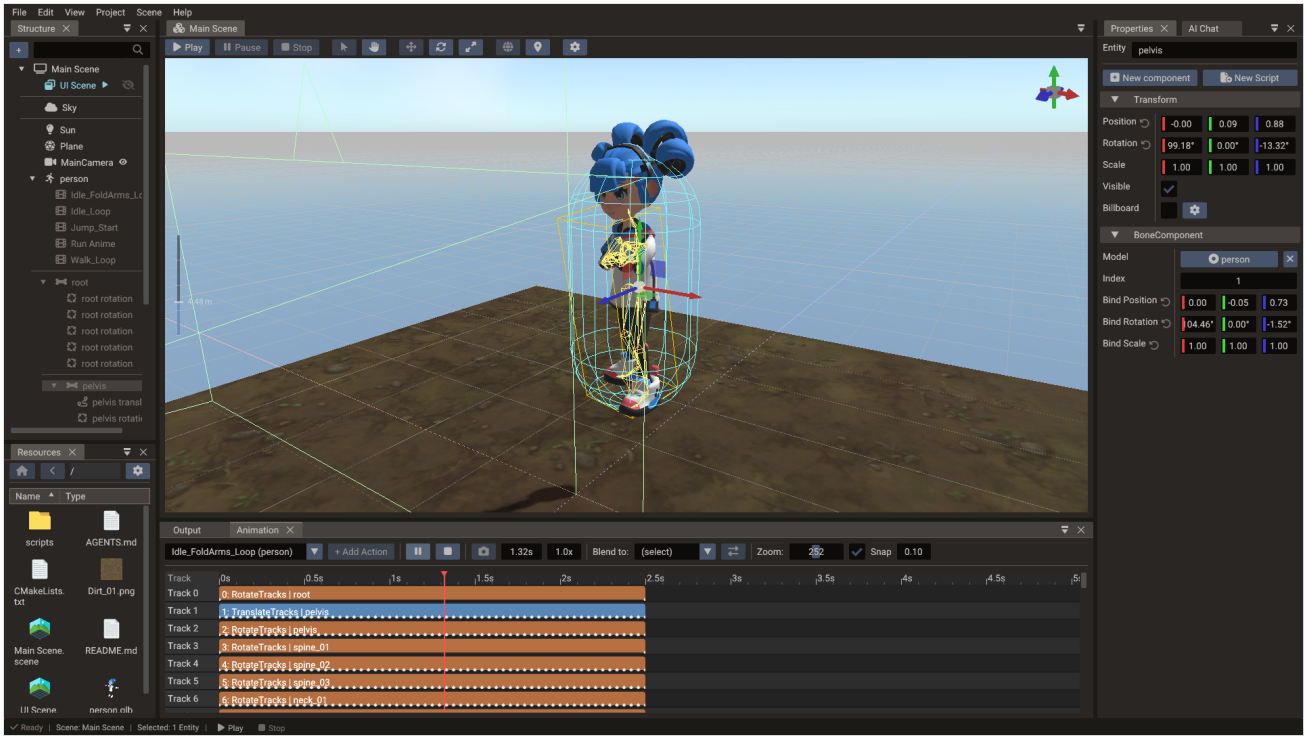


Models

Doriax loads 3D models in **GLTF** and **OBJ** formats, and **FBX** through an automatic conversion to glTF (see FBX models). Import a model as a resource and add it to a scene as an entity, then position it with its transform.

Models support:

- **Skeletal animation** — animate rigged characters via bones
- **Node animation** — play transform clips on the imported glTF node tree
- **Morph targets** — blend between mesh shapes for facial animation and deformation



GLTF compatibility and limits

Skinned GLTF models are not limited by a fixed-size shader uniform block. Bone matrices are uploaded each frame through a **storage buffer** (Vulkan, Metal, Direct3D 11) or an **unfilterable bone texture** (OpenGL / OpenGL ES), so large skeletons skin correctly on every backend.

Each skinned mesh keeps its bone matrices in a `SkinComponent`, which the loader adds to the mesh entity. A mesh without one draws rigid, in its bind pose. The matrix array has a compile-time capacity, `MAX_BONES` (default **96**), and meshes without a skin don't carry it. The editor grows that array for larger skins; export then raises `MAX_BONES` in the generated CMake project to cover the largest skin in your scenes (never below 96). A model loaded only at runtime that exceeds the exported capacity is rejected and the loader writes an error to the log. Raise `MAX_BONES` yourself in a standalone engine build if you spawn such models from code. See the Model reference (p. 451) and Build Options — Engine capacity macros (p. 359).

Morph-target capacity is per mesh primitive and depends on the data exported for each target:

Morph data	Maximum loaded targets
Position only	8
Position plus normal or tangent	4
Position, normal, and tangent	2

The importer bases that capacity on the leading targets it actually loads and ignores the unsupported tail. Sparse GLTF accessors are supported for morph-target attributes, including sparse accessors without a dense base `bufferView`; Doriax expands them into dense GPU vertex data during loading. See the Model reference (p. 451) for the exact import behaviour.

GLTF node hierarchy

How a GLTF becomes entities depends on what the file contains:

File contents	Scene layout
Animation clips, or more than one skin	The **full node tree** is imported. Every glTF node becomes a child entity (transform-only helpers, joints, and mesh nodes), parented as in the file.
Several mesh nodes, no clips and a single skin (or none)	One **child mesh entity per mesh node**, parented under the model root. Joints still use the legacy skeleton path when the file has one skin and no clips.

File contents	Scene layout
A single static mesh	Geometry lives on the Model entity itself.

On both child paths the Model entity keeps only its `Transform` and `ModelComponent`. It gets a `MeshComponent` only when the geometry lives on it: a single mesh, an OBJ, or a merged model.

On the full-tree path, bone and mesh mappings point into those node entities rather than sitting in a separate skeleton. Animation channels target the node they were authored against, so transform-only nodes (not just joints) play back. Models may use **more than one skin**: each skinned mesh keeps its own joint order and inverse-bind matrices, and the renderer builds that mesh's bone matrices from those joints into its `SkinComponent`.

Imported models with child nodes start collapsed in the Structure panel (p. 76). Expand the model when you need to select a specific node, joint, or child mesh.

On the child-mesh path the parts are yours to arrange: nest them under each other or under a plain entity inside the model, and the scene saves that layout. Full node-tree imports keep joints, helper nodes, animated parts, and rigid parts that inherit a node's animation where the file authored them; skinned parts and rigid parts the file does not animate can be moved, onto bones included, and any entity can be attached to a bone. See Structure — Organizing the parts of a model (p. 77).

Merging static model meshes

A GLTF with several mesh nodes normally keeps those meshes on child entities. That hierarchy is useful for editing parts separately, but GPU instancing (p. 274) draws only the geometry on the entity that owns `InstancedMeshComponent`.

For static props (no skinning, morph targets, or embedded animations), bake the children into the root:

1. Select the Model in the Structure panel (p. 78).
2. Right-click and choose **Merge static model**.
3. Add **Instanced Mesh** (or set `ModelComponent::mergeStaticMeshes` before reload in C++ / exported project code).

Restore model mesh children reverses the flatten. Merge is refused for skinned, animated, or morph-target models, and when the flattened result would exceed the root submesh limit. It is also refused while the parts are rearranged inside the model; use **Reset mesh parenting** first.

Mesh settings (lighting, shadows, detail levels, transparency and the custom shaders) follow the geometry. A reload of the same file gives each part the settings saved for its node, found by name and then by index, so reordered nodes keep theirs. Restoring gives the merged mesh's settings to every part, and merging takes those of the first part. A different file is a fresh import, so the meshes it creates start with the defaults.

Reloading a rearranged model

A model normally rebuilds its child entities on every reload: the old parts are deleted and the file creates fresh ones. Re-assigning the **same** model file — the usual step after re-exporting from your DCC — keeps what you arranged, in one of two ways depending on how the file imports.

Child-mesh models (static, or a single skin without clips) whose parts have been rearranged (p. 77) keep the existing part entities and refresh their geometry and materials in place. Assigning a different file, merging, or restoring still rebuilds the parts.

Whenever the file loads against existing part entities — that in-place reload, or reopening a saved scene — each part is paired with a mesh node of the file by **node name**, so re-exporting with nodes reordered keeps every part on its own geometry. The pairing goes:

1. A part whose name still matches the node at its saved index keeps that node.

2. Remaining nodes claim the part with the same name, wherever it moved to.
3. Nodes without a name, and parts you renamed in the editor, fall back to the saved index.

A part left without a node stays in the scene as a plain entity and no longer belongs to the model (a warning names it in the log); delete it or keep it as you see fit. A new node in the file gets a new part under the model root. Kept parts keep the transform saved in the scene — the file's node transform is applied only when a part is created — and their submesh overrides are preserved.

Full node-tree models cannot be refreshed in place, because the node transforms, joints, and animation clips all come from the file. They are rebuilt, and the editor then puts your arrangement back by node name: every part you moved returns to the node or group you left it under, keeping the local transform you gave it. A part the re-export turned into something that may no longer move — it gained keyframes, or now sits under an animated node — stays where the new file puts it, with a warning in the log. Part entities are new after this kind of reload, so entity references that pointed at a part have to be re-assigned.

Attachments survive both kinds of reload. Anything you hung on a bone, a helper node, or a part — a light, a camera, an empty holding a prop, another model — is kept and re-attached to the same-named node with the same local offset, on static models too. The node is matched by name as well as index, so it follows a re-export that reorders nodes; if the node is gone from the file, the entity is left directly under the model with a warning.

Assigning a **different** model file rebuilds everything from that file and does not carry attachments or arrangements over.

FBX models

The engine does not read FBX itself. The editor converts each FBX file to a binary glTF, and the engine loads that conversion wherever the FBX is used, so scenes, components, and scripts keep the `.fbx` path:

C++

```
Model hero(&scene);
hero.loadModel("characters/hero.fbx");
```

Lua

```
local hero = Model(scene)
hero:loadModel("characters/hero.fbx")
```

A script can load any FBX in the assets directory, because Play and export convert them all first. An FBX without a conversion does not load, so an engine build used without the editor cannot load FBX at all.

While you work, conversions live in `.doriax/imported/`, which mirrors the assets directory (`characters/hero.fbx` becomes `.doriax/imported/characters/hero.fbx.glb`) and is not committed. An exported game ships each one beside its FBX as `hero.fbx.glb` and leaves the `.fbx` out. See Export — FBX models (p. 160).

The editor converts an FBX the first time it is used (dropped into a scene, picked for a component or a terrain layer, or previewed in the Resources Browser), when Play starts, and on export. It converts it again when the FBX or one of its textures changes. Models in open scenes then reload the way a re-assigned file does, keeping your arrangement (see Reloading a rearranged model). **Reimport** in the Resources Browser (p. 109) forces a conversion.

Texture paths stored in an FBX often point at the artist's machine, so each texture is looked up in this order:

1. The paths stored in the FBX, when they lead inside the assets directory.
2. The file name in the FBX's folder, in its `textures`, `images`, `materials`, `maps`, or `tex` subfolder, and in the same places of each parent folder up to the assets root.
3. The stored paths outside the assets directory.

A texture found inside the assets directory stays a separate file that the conversion refers to. One found only outside is embedded in the conversion. A texture that is not found is left out with a warning in the Output

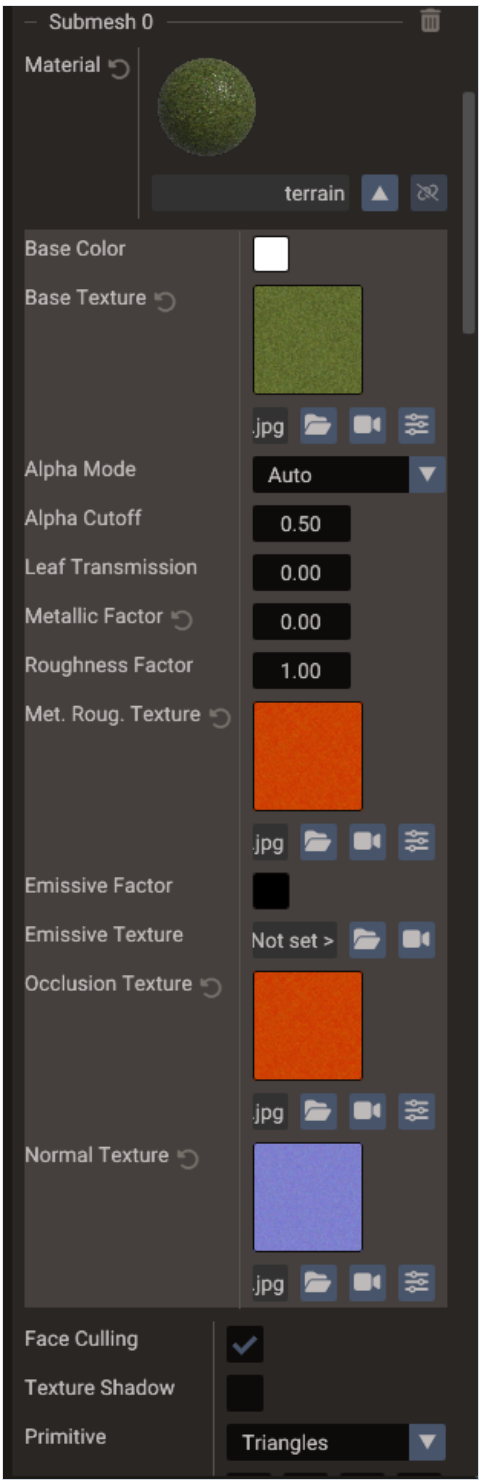
panel, and adding it later converts the FBX again. Maps that glTF keeps in one image are combined into a new embedded image: an opacity map goes into the base colour's alpha, and separate roughness and metalness maps are packed together.

FBX content	Converted to
Units and axes	Meters, Y up
Meshes and materials	glTF meshes with PBR materials. Instances of one mesh that use different materials keep them
UV sets	The first two. Textures on a third set use the first
Vertex colours	Vertex colours
Skins	Up to four joint influences per vertex
Blend shapes	Morph targets with position offsets, up to 8 rendered (see the capacity table above). An in-between shape keeps only its full-weight target
Animation stacks	Clips sampled at 30 frames per second, blend-shape weights included

Texture tiling and offset are not converted, and cameras and lights in the file become empty nodes.

PBR materials

Rendering is **physically based (PBR)**, producing realistic surfaces that respond correctly to lighting. Materials drive how light interacts with each surface, supporting photorealistic results.



Materials can use albedo, normal, roughness, metallic, and emission data. Imported GLTF materials are converted into the engine material representation during loading; editor-created materials are serialized with the project and baked into exported output.

GLTF `OPAQUE`, `MASK`, and `BLEND` alpha modes are preserved. For masked materials, the GLTF `alphaCutoff` value (default `0.5`) controls the combined base-colour factor and texture alpha test in both the visible surface and its shadow/depth passes. Blended submeshes keep depth testing but **disable depth writes** in the colour pass so overlapping translucent surfaces composite instead of occluding each other. When auto-transparency marks the mesh transparent, the SSAO depth pre-pass and SSR G-buffer skip it; shadow maps still render it. In the editor, expand **Mesh → Submesh → Material** to change **Alpha Mode** and **Alpha Cutoff**. See the Material reference (p. 735) for the mode behaviours.

A mesh with **Face Culling** off is lit on both sides: the back face shades with its normal (and normal map) flipped toward the viewer, as glTF specifies for double-sided materials, instead of appearing in shadow. **Leaf Transmission** (`foliageTransmission` (p. 736), `0`–`1`, default `0`) adds the light that passes through a thin surface such as a leaf when the sun is behind it, scaled by the base colour and the same shadow term as direct lighting. Set it per submesh in the editor, or author it in the glTF as the material extra `doriax_foliage_transmission`; ordinary glTF materials are unaffected.

Editing an imported model's submeshes

A model's geometry and materials are rebuilt from its source file every time the scene loads, so the file — not the scene — is what the mesh normally follows. Editing a submesh property of an imported model in the editor records an **override** on the model: the edited field keeps your value across scene restarts, play/stop, and exported builds, while every field you left alone still follows the source file.

That applies to the whole **Submesh** section — material values and textures, linked `.material` files, **Face Culling**, **Texture Shadow**, and **Primitive Type** — on the Model entity itself and on the child mesh entities a multi-node GLTF creates.

Geometry can't be overridden. Each primitive in the file is one submesh, so these meshes have no **Create Shape** and submeshes can't be added or removed. Change the geometry in the model file instead.

Because only the edited fields are pinned, re-exporting the model from your DCC still propagates everything else. Change a texture in Blender and re-export: meshes where you only tweaked base colour pick up the new texture and keep your colour.

Overrides are matched to the primitive they came from by its GLTF node and material name (the material name alone for OBJ), not by position. Reordering nodes or primitives in the source file therefore keeps each edit on the right geometry. When that identity no longer resolves to exactly one primitive — the node or material was renamed or removed, or two of them now share a name — the override is dropped instead of applied to whichever geometry took the old slot. Re-apply the edit after that kind of re-export.

Assigning a **different** model file to the entity is a fresh import and clears the overrides. Reloading the same file — **Merge static model** or **Restore model mesh children** — keeps them.

Scenes authored before overrides existed are migrated the first time they load: the values saved in the scene are compared against the source file, and only the fields that really differ become overrides.

Shared material files (`.material`)

Each mesh submesh can reference a standalone **`.material`** file in your project instead of embedding values only inside the scene. Multiple meshes can link to the same file so they always stay in sync — change the file once and every linked mesh updates.

Create a material file from the editor:

1. Select a mesh and open its **Material** row in the Properties window.
2. Tune base colour, textures, metallic, and roughness as needed.
3. **Drag the material preview** from Properties into the Resources Browser (p. 106).
4. The editor creates `Material.material` (or `Material_1.material`, etc.) in the folder you drop on and **links** the submesh to that file.

Apply an existing material file:

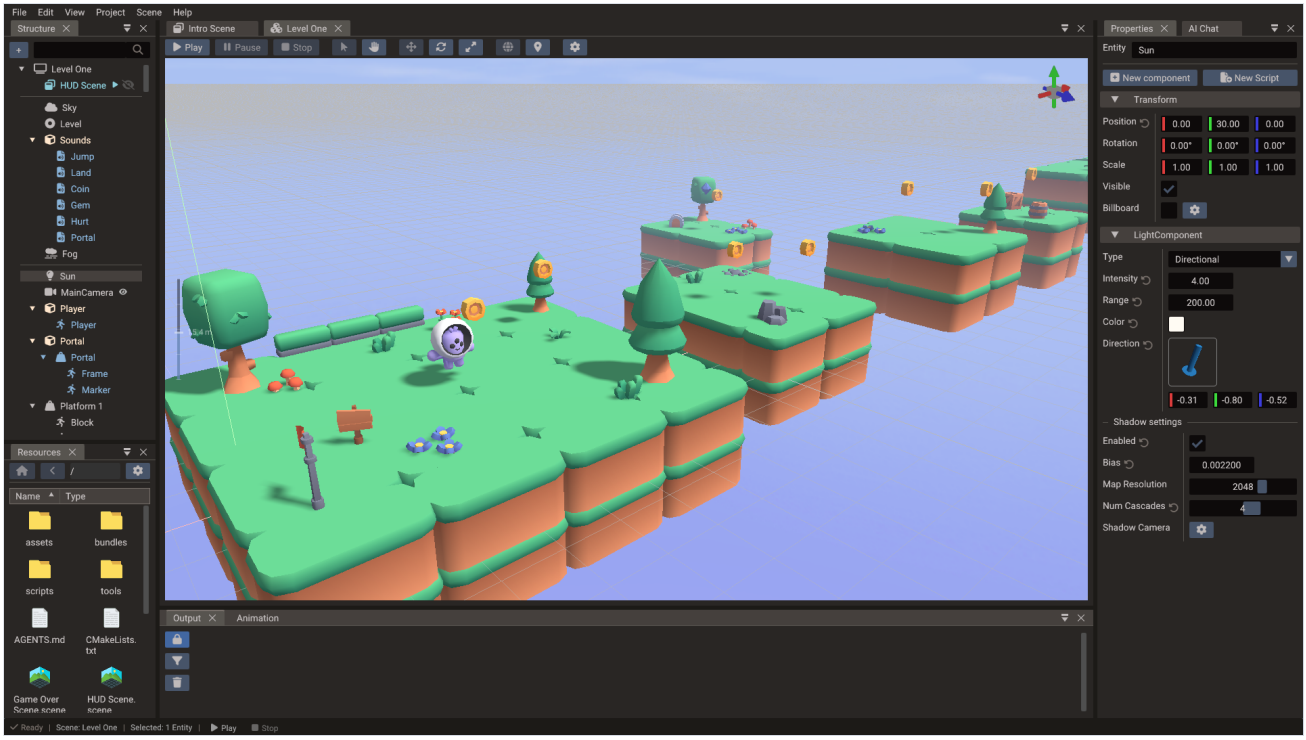
- Drag a `.material` file from the Resources Browser onto a mesh in the Scene view (p. 81), or
- Drop it onto the **Material** field in Properties (live preview while hovering).

Linked materials reload automatically when the file changes on disk. Use the **unlink** button (chain icon) next to the material name in Properties to copy the values back into the scene as a local, unlinked material.

See Resources Browser — Material files (p. 108) and Properties — Mesh materials (p. 95) for the full workflow.

Lighting and shadows

Doriax supports multiple light types with **dynamic shadows**, so moving objects cast and receive shadows in real time.



Light type	Best use
Directional	Sunlight or broad outdoor lighting
Point	Lamps, torches, explosions, and local lights
Spot	Flashlights, cones, and focused effects

A scene can have any number of lights, but only six are used at once: past that, directional lights come first, then the ones nearest the camera (see Rendering Pipeline — More lights than `MAX_LIGHTS` (p. 260)). Directional lights support cascaded shadow maps. Tune edge smoothness with `Scene::setShadowQuality` (`NONE` / `LOW` / `MEDIUM` / `HIGH` PCF filtering).

Projected spot-light masks

A spot light uses a circular inner/outer cone by default. To project an arbitrary shape, select the spot light and assign an image to **Light → Mask**. No separate shape mode is needed: an assigned mask automatically becomes the complete light shape, and clearing it restores the circular cone.

Mask alpha controls light intensity when the image contains transparency; otherwise the engine uses luminance (black is unlit and white is fully lit). The **Angle Cone** property sets the vertical projection angle, while the image aspect ratio sets its width. The scene gizmo and spot shadow frustum use that same orientation and aspect ratio.

For C++ and Lua setup, in-memory masks, and clearing a mask at runtime, see Light.spotMask (p. 427).

Enable **screen-space ambient occlusion** with `Scene::setSSAOEnabled(true)` to add soft contact shading in creases and corners. It affects ambient/indirect light only — see Rendering Pipeline — Ambient occlusion (SSAO) (p. 265).

Environment

Add atmosphere to your scenes with:

- **Fog** — depth-based atmospheric fog
- **Sky system** — a configurable cubemap background that also drives **image-based lighting (IBL)** for reflective surfaces
- **Reflection probes** — local, box-shaped reflection environments for interiors and enclosed areas
- **Water** — animated water surfaces with waves, reflections, refraction, and foam

Sky and reflections

A **Sky** entity provides the scene background and the lighting environment for IBL. Assign a cubemap texture (six faces or a single cross layout). Meshes that should pick up sky reflections and indirect colour need **Receive IBL** enabled on their Mesh component — see Rendering Pipeline — IBL (p. 263).

Use **Visible** on the Sky component when you want IBL without drawing the sky dome (for example, a studio HDR used only for reflections). The sky texture still generates irradiance and prefiltered maps either way.

IBL gives soft, environment-wide reflections suited to curved and rough surfaces. For a **local environment** (a room, garage, or cave) use a **Reflection Probe**; for a **sharp, mirror-like reflection on a flat surface**, add a **Mirror** — see below.

Reflection probes

Sky IBL reflects the same environment everywhere, which looks wrong indoors. A **Reflection Probe** (Structure panel's create menu) captures the scene at its own position and applies that environment to meshes inside a box-shaped influence volume, with box-projected (parallax-corrected) reflections that stay anchored to the room. Probes can be **static** — an authored cubemap or a one-time capture at load — or **dynamic**, re-capturing at runtime on move, on an interval, or on demand. A mesh that wraps around a probe would show up inside its own reflection, so turn off its **Render in Probes** flag to keep it out of captures. See Rendering Pipeline — Reflection probes (p. 263).

Mirrors

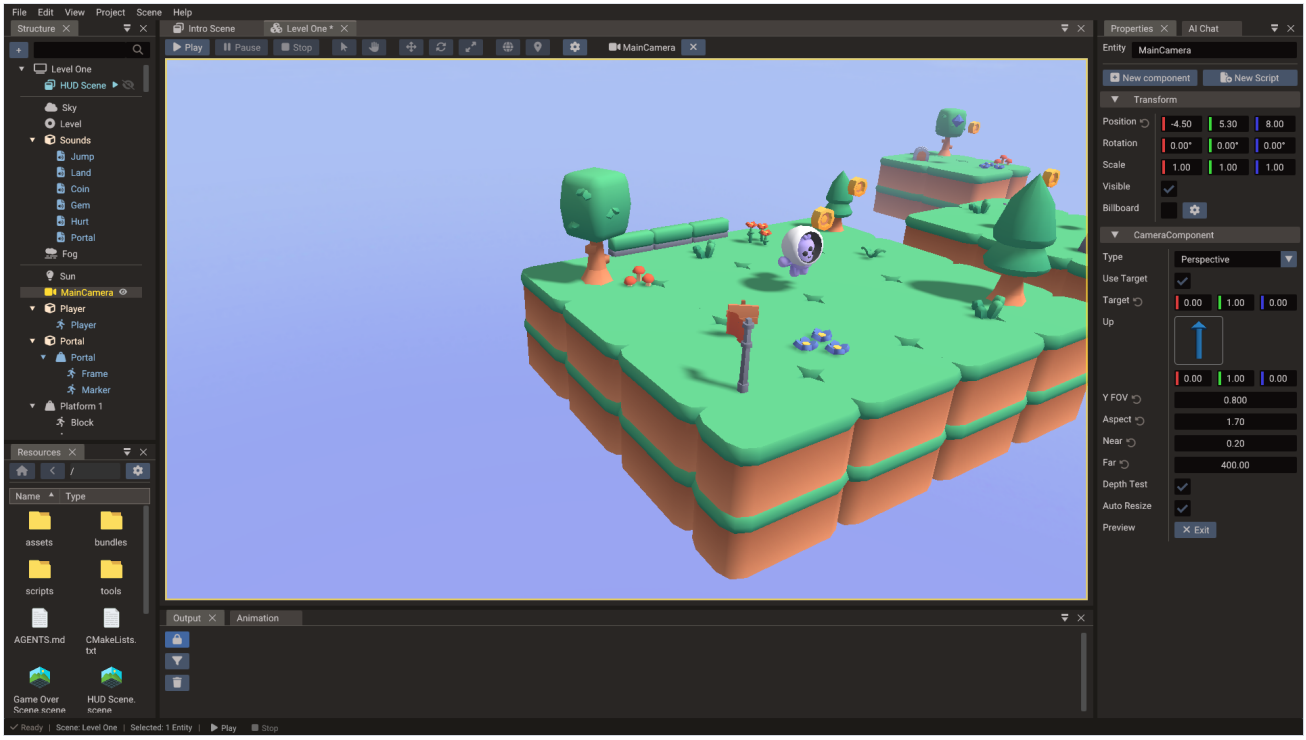
For a true planar reflection on a flat surface — a mirror, still water, or a polished floor — add a **Mirror** from the Structure panel's create menu. It produces an upright reflective wall that reflects the scene from the viewer's mirrored viewpoint, with no camera or texture setup required. Because a mirror shows the side of an object facing it — the side a single key light usually misses — reflected meshes want **Receive IBL** or a fill light to avoid looking washed out. See Rendering Pipeline — Mirrors and planar reflections (p. 269) for how it works, its lighting, and its performance cost.

Water

For a sea, a lake, or a pool, add a **Water** from the Structure panel's create menu. It draws an animated surface at the entity's height: Gerstner waves you shape with a height, length, direction, and steepness, scrolling ripples, sky or planar reflections, refraction of the scene behind it, shore and crest foam, and a fade into the water when the camera goes below the surface. Dynamic 3D bodies with a **Buoyancy** above `0` float on its waves, and scripts can read the surface with `Water.getHeight`. See Rendering Pipeline — Water (p. 270), Physics — Buoyancy (p. 319), and the Water reference (p. 496).

Cameras

Cameras define the viewpoint into a 3D scene. Position and orient a camera entity, and set it as the scene's active camera to control what the player sees.



Cameras can be perspective, orthographic, or UI cameras. A scene owns an active camera entity; editor scene cameras are separate from game cameras so you can navigate while preserving the player's view.

Additional features

The runtime also supports particle systems (p. 306), terrain (p. 309) with level-of-detail (LOD), automatic mesh detail levels (p. 275) for distant props, and mesh instancing (p. 274) for efficiently rendering many copies of the same geometry. An optional depth prepass (p. 276) cuts overdraw on dense opaque scenes.

Next steps

Add interactivity with Physics (p. 315), or learn how to ship your game in Export Window (p. 155).

Rendering Pipeline

Doriax renders through a flexible pipeline that supports 2D, 3D, UI, and render-to-texture workflows. The same codebase targets OpenGL, OpenGL ES, Metal, Direct3D 11, and WebGPU through a backend abstraction layer.

High-level render flow

Each frame, the engine runs the following phases in order:

1. **Camera update** — The active scene camera computes the view matrix and projection matrix.
2. **Frustum culling** — Renderables outside the camera frustum are excluded. Terrain and tilemaps cull below entity level: only the terrain nodes and tilemap chunks that the camera can see are submitted. Terrain foliage (p. 312) is batched into chunks around the camera and culled the same way, with instances scaling in over the last quarter of the layer's draw distance instead of popping. Instanced meshes keep a per-view instance list for the main camera and each shadow atlas slot, so instances outside a view are not submitted to it.
3. **Depth prepass** *(optional)* — When depth prepass is enabled, opaque meshes write depth only so the colour pass can shade each pixel once. Off by default.
4. **Opaque pass** — Opaque geometry is drawn with depth testing, nearest first in distance bands (one per doubling of the distance) so early-Z rejects most hidden surfaces. Inside a band, meshes that share a pipeline, vertex buffer, and texture are drawn one after another, so the GPU state changes less. With mesh detail on, each mesh picks a detail level from its projected geometric error.
5. **Lighting and shadows** — Shadow maps are rendered for each shadow-casting light, then the lighting pass applies directional, point, and spot lights.
6. **Skybox and IBL** — The sky cubemap is drawn (when visible). Environment maps derived from the sky feed **image-based lighting (IBL)** on meshes that opt in.
7. **Transparent pass** — Objects with blending enabled are sorted back-to-front and drawn after opaque geometry. Blended submeshes still depth-test, but they do **not** write depth in the colour pass, so overlapping translucent surfaces composite instead of punching holes in each other. A mesh marked transparent (including by `autoTransparency`) is skipped by the SSAO depth pre-pass and the SSR G-buffer; shadow maps still render it. A Water surface is drawn inside this pass: transparencies on the far side of it first, then the water, then the rest.
8. **UI pass** — UI entities are rendered in screen-space canvas coordinates, on top of the 3D or 2D scene.
9. **Post-processing** — Screen-space effects run over the finished image: SSR composites its reflections, then the scene's post-process chain applies each enabled user pass in order. (Fog is not a post pass — it is computed in the mesh shader.)

Cameras that render to a texture (minimaps, mirrors, water reflections, portals) run this same flow into their own framebuffer before the main view is drawn.

Cameras

Three camera projection modes are available:

Projection	Use	Object
Perspective	3D games	`Camera` with `setType(CameraType::CAMERA_PERSPECTIVE)`
Orthographic	2D games, isometric	`Camera` with `setType(CameraType::CAMERA_ORTHO)`
UI	Screen-space canvas	`Camera` with `setType(CameraType::CAMERA_UI)`

Use `Camera::setNearClip()` and `Camera::setFarClip()` to tune depth precision. Set a tight near/far range — a large ratio (e.g. 0.1 to 100 000) causes depth fighting on distant surfaces. You can also configure the

projection in one call with `setPerspective(yfov, aspect, near, far)` or `setOrtho(left, right, bottom, top, near, far)`.

Each camera also decides whether it uses the depth buffer at all. `Camera::setDepthTest(false)` drops depth testing and depth writes from its colour pass, so the camera composites renderables in submission order — later draws land on top, whatever their distance — and the back-to-front sorting of `setTransparentSort()` is skipped with it. That is the canvas behaviour a 2D or UI scene wants, so the camera a scene creates for itself and the ones the editor adds to a 2D scene already have it off; a 3D camera keeps it on so geometry occludes by depth.

A camera can render to its own texture instead of the screen — see Framebuffers and render-to-texture below.

PBR materials

Doriax uses a **Physically Based Rendering** (PBR) material model with the following texture slots and scalar properties:

Slot / Property	Controls
`baseColorTexture` + `baseColorFactor`	Base surface color (albedo)
`alphaMode`	Alpha handling: `AUTO`, `ALPHA_OPAQUE`, `MASK`, or `BLEND`
`alphaCutoff`	Cutout threshold for `MASK` materials (default `0.5`)
`normalTexture`	Surface micro-detail
`metallicRoughnessTexture` + `metallicFactor`	Surface reflectivity (0 = dielectric, 1 = metal)
`metallicRoughnessTexture` + `roughnessFactor`	Highlight spread (0 = mirror-smooth, 1 = fully rough)
`occlusionTexture`	Ambient occlusion darkening in crevices
`emissiveTexture` + `emissiveFactor`	Self-illuminated areas that ignore lighting

Metallic and roughness share one texture, following the GLTF convention (roughness in the green channel, metallic in the blue channel).

Alpha is the product of `baseColorFactor.a` and the base-colour texture's alpha. `ALPHA_OPAQUE` forces the result opaque, `MASK` discards pixels below `alphaCutoff`, and `BLEND` preserves partial alpha for transparent rendering. `AUTO` keeps the historical Doriax texture-alpha detection used by editor-created materials. Imported GLTF/GLB materials preserve their explicit alpha mode. A blended submesh (`BLEND`, or `AUTO` when the texture or factor is translucent) disables depth writes on its colour pipeline.

The `MASK` test is identical in the lit, shadow/depth, and SSR G-buffer passes. A cutout therefore casts and contributes to screen-space effects with the same silhouette that is visible in the colour pass.

```
Material mat;
mat.baseColorTexture = Texture("textures/rock_albedo.png");
mat.normalTexture    = Texture("textures/rock_normal.png");
mat.roughnessFactor  = 0.85f;
mat.metallicFactor   = 0.0f;

mesh.setMaterial(0, mat);   // submesh index, material
```

Lighting

The engine supports three light types plus global illumination:

Type	Use
Directional	Sun / moon — affects the entire scene
Point	Omni-directional bulbs, torches

Type	Use
Spot	Flashlights, stage lighting, headlights

Scene ambient light (`Scene::setGlobalIllumination`) fills shadowed areas with a flat tint. It is separate from IBL — ambient light affects all lit meshes uniformly, while IBL adds directional reflections and diffuse fill derived from the sky environment.

```
Light sun(&scene);
sun.setType(LightType::DIRECTIONAL);
sun.setDirection(Vector3(-0.5f, -1.0f, -0.5f));
sun.setColor(Vector3(1.0f, 0.95f, 0.8f));
sun.setIntensity(3.0f);
sun.setShadows(true);
```

More lights than `MAX_LIGHTS`

The shaders take `MAX_LIGHTS` (6) lights at once, but a scene can hold more. Each frame the lights with an intensity above zero fill those slots: directional lights first, then the point and spot lights nearest the main camera. A light already in a slot counts as 10% nearer, so two lights at about the same distance don't swap back and forth as the camera moves. A light at intensity `0` takes no slot, so switching it off frees one for another light.

Shadow atlas slots then go to the shadow-casting lights among the selected ones. The shader variants still follow every light in the scene, so a light moving in or out of the selection never rebuilds a shader.

Render-to-texture cameras, mirrors, reflecting water and reflection probes pick the lights nearest to their own position the same way, favouring the ones the main camera uses. A light that only such a view picks has no shadow or spot mask there.

Shadow maps

Each shadow-casting light renders a depth map. Engine limits, which the editor compiles into the shaders it builds, so an exported project can't change them:

Constant	Default	Controls
`MAX_LIGHTS`	6	Max simultaneous lights
`MAX_SHADOW_ATLAS_SLOTS`	9	Projective atlas slots (one per spot light or directional cascade)
`MAX_POINT_SHADOW_ATLAS_SLOTS`	24	Point atlas slots, `SHADOW_CUBE_FACES` (6) per light
`MAX_POINT_SHADOW_LIGHTS`	4	Shadow-casting point lights (derived from the two above)
`MAX_SHADOWCASCADES`	4	Cascades for directional CSM

Like Unity, Unreal, and Godot, directional cascades are **stable**: each one covers the bounding sphere of its slice of the view and moves in whole texels, so shadow edges keep their width when the camera turns and stay put when it moves, at the cost of some resolution. An orthographic camera uses a single cascade fitted to its view.

A cascade ends at its split by default, which can show as a line where the shadow resolution changes. Turn on **Cascade Blend** on the light (`Light::setCascadeBlend`) to fade each cascade into the next over a short band past the split. It is off by default, as in Godot, because it costs frame time: the pixels in the band sample two cascades, and the finer cascade also draws the casters of the band. Lost Slime 3D, with four cascades, runs about 4% slower with it on.

Shadow edge smoothness is a per-scene setting, `Scene::setShadowQuality`, with **Percentage Closer Filtering (PCF)** kernels of `NONE` (1 tap), `LOW` (3x3, default), `MEDIUM` (5x5), or `HIGH` (7x7). The kernel size is uniform-driven, so changing it applies instantly without shader rebuilds:

```
scene.setShadowQuality(ShadowQuality::MEDIUM);
```

Directional and spot shadows sample a depth atlas through a hardware comparison sampler, so every tap is already filtered across a 2x2 neighbourhood: `NONE` still softens edges slightly rather than producing hard ones. The PCF kernel then pairs those bilinear taps along each axis, so a 3×3 / 5×5 / 7×7 filter costs about a quarter of the naïve tap count while matching the same weighted result. Point lights keep a packed colour depth map and stay hard-edged at `NONE`.

2D lighting and shadows

2D scenes use a **dedicated forward light path** instead of the PBR model: each Light2D (p. 430) contributes `color · intensity · falloff(distance)` and all contributions add on top of the scene's **2D ambient light** (`Scene::setAmbientLight2D`). The result multiplies the sprite's base color, so the unlit fast path is preserved — a 2D-lit sprite costs far less than a PBR-lit mesh. Both paths can coexist: a mesh lit by 3D lights in the same scene simply adds the 2D contribution on top.

A light's optional `height` places it on a virtual Z above the 2D plane, giving normal maps a direction to respond to. Sprites, tilemaps, and mesh polygons generate tangents automatically, so assigning a normal texture to their material is enough.

2D lights ignore the scene's `lightState` flag — that switch controls only the 3D lighting pass.

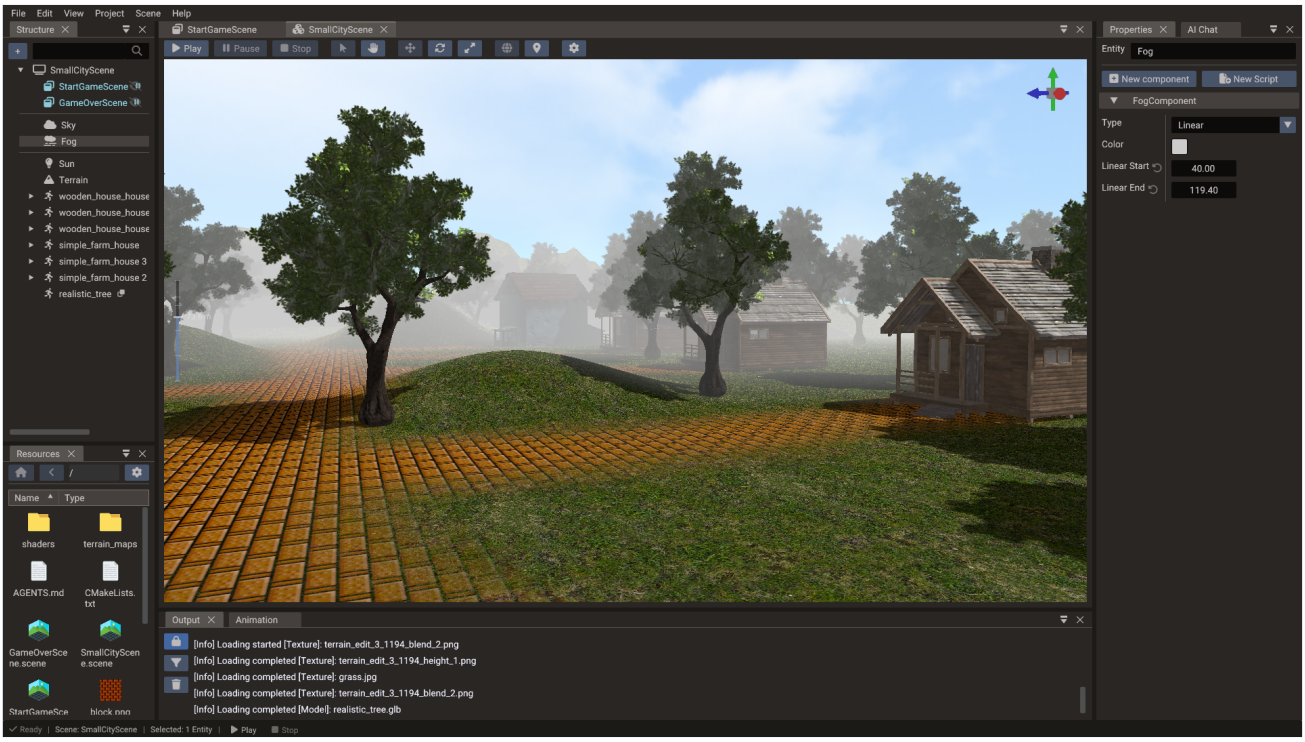
1D polar shadow maps

Shadows from Occluder2D (p. 462) components use per-light **1D polar shadow maps**: all occluder outlines in the scene are merged into one segment buffer and rendered into a one-pixel-tall atlas row per shadow-casting light, where the X axis is the angle around the light and the stored value is the distance to the nearest occluder. Lit fragments compare their own angle and distance against the row, with a PCF filter along it for soft edges: the light's `shadowSoftness` sets the penumbra width and the scene's `shadow2DQuality` (`NONE` / `LOW` / `MEDIUM` / `HIGH` — 1 to 13 taps) sets how smoothly it is sampled. The tap count is uniform-driven, so quality changes apply instantly without shader rebuilds.

Constant	Default	Controls
`MAX_LIGHTS_2D`	16	Max simultaneous 2D lights (and shadow atlas rows)

The shadow pass renders line segments only (no scene geometry), so it stays cheap even with many occluders; each shadow-enabled light adds one atlas row at the light's `mapResolution` width. Each light's shadow attenuates only that light's own contribution — ambient light is never shadowed. See 2D Graphics — 2D lighting (p. 244) for the usage guide.

Fog



`Fog` adds atmospheric depth. Two modes are available:

Mode	Effect
`LINEAR`	Fog blends from `linearStart` to full density at `linearEnd`
`EXPONENTIAL`	Density increases exponentially with distance
`EXPONENTIALSQUARED`	Steeper exponential falloff

```
Fog fog(&scene);
fog.setType(FogType::EXPONENTIAL);
fog.setColor(Vector3(0.7f, 0.7f, 0.8f));
fog.setDensity(0.02f);
// For LINEAR mode use fog.setLinearStartEnd(start, end) instead of density
```

Skybox

Add a **Sky** entity (Skybox component) and assign either a single cubemap texture or six cube-face textures. The sky is rendered as an infinite background behind opaque geometry.

```
SkyBox sky(&scene);
sky.setTexture("textures/sky_cubemap.png");

// or per-face:
sky.setTextures("daysky",
    "sky/px.png", "sky/nx.png",
    "sky/py.png", "sky/ny.png",
    "sky/pz.png", "sky/nz.png");
```

In the editor, the Sky component also exposes **Visible**. When disabled, the sky is not drawn in the viewport but still generates IBL environment maps for meshes that use them. Use this when you want reflections and indirect lighting from an environment without showing the sky dome itself (for example, an interior level with a hidden outdoor HDR environment).

Image-based lighting (IBL)

When a scene contains a Sky entity with a valid cubemap texture, the engine builds two environment maps from that sky:

Map	Purpose
Irradiance	Diffuse ambient fill — soft colour bounced from every direction
Prefiltered specular	Glossy reflections — sharper highlights on smooth (low-roughness) surfaces

These maps follow the glTF-style split-sum approximation used in modern PBR pipelines. Rough surfaces sample blurrier mips; mirror-like surfaces pick up crisp sky detail.

IBL is **per mesh**. Each mesh has a **Receive IBL** flag (default `false` in new scenes). Only meshes with this enabled combine punctual lights (directional/point/spot) with the sky environment. Meshes also need **Receive Lights** enabled and a valid normal (tangent space for normal maps).

Typical workflow:

1. Add a Sky entity and assign a cubemap (HDR or LDR).
2. Select meshes that should reflect the environment (metal, glass, wet stone, etc.).
3. Enable **Receive IBL** on those meshes in the Properties window, or call `setReceiveIBL(true)` (Lua: `mesh.receiveIBL = true`) in code.
4. Tune **Roughness** and **Metallic** on the material — low roughness makes reflections more obvious.

The material preview sphere in the Properties window updates when **Receive IBL** is toggled, so you can compare lit-only vs environment-lit looks before play mode.

> **One sky per scene**
> The render system uses the first Sky component in the scene for both drawing and IBL generation. Keep a single active sky environment unless you know you are replacing it.

Reflection probes

IBL reflects one sky environment everywhere, which reads as "outdoors" on every surface. A **Reflection Probe** captures the environment *at a point in the scene* and applies it to meshes inside a box-shaped influence volume — chrome in a garage reflects the garage, not the sky. Probe reflections are **box-projected** (parallax-corrected): reflection rays are projected onto the influence box, so the reflection stays anchored to the room's walls as objects and the camera move, instead of floating at infinity.

The simplest way to add one is the **Reflection Probe** entry in the Structure panel's create menu. Size its **Box Size** to the room or area it represents, and enable **Receive IBL** (plus **Receive Lights**) on the meshes inside — probes use the same per-mesh opt-in as sky IBL. A Sky is not required: probes supply the specular (reflection) term on their own, while diffuse ambient still comes from the sky irradiance when one is present.

Static and dynamic probes

Mode	Behaviour
Static	Uses an authored cubemap when one is assigned; otherwise captures the scene once at load (or when **Refresh Probe** is pressed) and keeps that result.
Dynamic	Re-captures the scene at runtime according to its update policy.

An authored cubemap is the cheapest option: it is GGX-prefiltered once and cached like the sky environment. Runtime captures are GGX-prefiltered on the GPU each time they finish, so rough surfaces get correct blurry reflections from both.

Dynamic probes choose when to re-capture with **Update**:

Update	When it captures
On Load	Once when the scene starts
On Move	Whenever the probe entity moves
Interval	Every **Update Interval** seconds
Manual	Only when requested — the **Refresh Probe** button in the editor, or setting `needUpdate = true` from code

Runtime captures share a strict budget of **one cubemap face per frame**: a full refresh takes six frames, and multiple pending probes take turns. Even several dynamic probes therefore cost a fraction of a mirror's full extra scene render per frame — the trade-off is latency, not throughput.

Influence volume and blending

Property	Purpose
Box Size	The influence volume's size, scaled by the entity's world scale. The box is world-axis-aligned and centred on the entity.
Box Offset	Moves the influence box in the entity's local space. The cubemap is still captured at the entity origin (shown as a gold marker in the viewport when they differ).
Blend Distance	Fade band inside the box edges where the probe blends into the sky IBL, hiding the seam at the volume boundary. Runtime blending is limited to the box's smallest half-extent.
Intensity	Multiplier on the probe's reflection contribution.
Priority	When influence boxes overlap, the higher-priority probe wins; on a tie, the probe whose centre is nearest the mesh wins.

The engine picks **one probe per mesh** (using the mesh's world-bounds centre), so keep volumes room-sized rather than object-sized: a mesh is either inside a probe's box or it falls back to the sky environment.

Capture settings

Property	Purpose
Cubemap	*(static only)* Authored six-face cubemap. Leave empty to capture at load instead.
Resolution	Capture cubemap face size (16–1024, default 128). The capture is prefiltered into 128-pixel faces, so sizes above 128 make reflections smoother, not sharper.
Near / Far	Clip planes of the capture cameras.
Include Sky	Whether the sky (and scene background colour) appears in the capture.

Individual meshes stay out of captures with **Render in Probes** on the Mesh component (`renderInReflectionProbes`). Turn it off for geometry wrapped around a probe — a car body with a probe at its centre, say — which would otherwise fill its own reflection. The mesh still renders normally everywhere else. Toggling it re-captures every probe in the scene, because static and non-interval probes keep their baked cubemap until something invalidates it.

```cpp
// C++: a dynamic probe covering a 12x6x12 room
ReflectionProbe probe(&scene);
probe.setPosition(0.0f, 2.0f, 0.0f);
probe.setBoxSize(12.0f, 6.0f, 12.0f);
probe.setMode(ReflectionProbeMode::DYNAMIC);
probe.setUpdateMode(ReflectionProbeUpdateMode::MANUAL);
probe.refresh();   // manual re-capture from code
```

```lua
local probe = ReflectionProbe(scene)
probe:setPosition(0, 2, 0)
probe:setBoxSize(12, 6, 12)
probe.mode = ReflectionProbeMode.DYNAMIC
probe.updateMode = ReflectionProbeUpdateMode.MANUAL
probe:refresh()   -- manual re-capture from code
```

With SSR enabled, a ray that finds a hit replaces the probe's reflection, as it does the sky's. The composite rebuilds up to four probes a frame, the first ones used by visible objects that receive IBL (two probes sharing a cubemap take two); an object lit by any other probe keeps its probe reflection, with SSR added on top.

> **Capture scope and limitations**
> Runtime captures render opaque meshes and (optionally) the sky. Transparent meshes, UI, and particles are skipped. Point and spot shadows are reused in captures, but directional shadow cascades are fitted to the main camera, so their coverage inside a capture can be partial. A probe never appears in its own capture, and captures do not include other probes' reflections (no recursion).

Ambient occlusion (SSAO)

Screen-space ambient occlusion darkens creases, corners, and contact areas where ambient light is naturally blocked. Like Godot, Unity, and Unreal, it modulates only the **ambient/indirect** term (IBL or global illumination) — direct light from your sun, point, and spot lights is left untouched — so it reads as soft contact shading rather than a second shadow.

Each frame the render system runs a small depth pre-pass for the main camera, derives occlusion from a rotated hemisphere kernel, blurs it, and the lit mesh shader multiplies the result into its ambient term. The blur leaves out samples across a depth edge, so occlusion stays on its own surface instead of haloing around silhouettes.

SSAO is a scene setting:

```
scene.setSSAOEnabled(true);
scene.setSSAORadius(0.5f);     // view-space sampling radius (world units)
scene.setSSAOIntensity(1.0f);  // strength (exponent on the occlusion factor)
scene.setSSAOBias(0.025f);     // depth bias to avoid self-occlusion acne
scene.setSSAOHalfResolution(true); // a quarter of the pixels, a little softer
```

```
scene.ssaoEnabled = true
scene.ssaoRadius = 0.5
scene.ssaoIntensity = 1.0
scene.ssaoBias = 0.025
scene.ssaoHalfResolution = true
```

In the editor, the same controls live under **Scene → Ambient Occlusion (SSAO)** in the Properties window, including a **Debug View** toggle that renders the raw AO buffer so you can tune radius/intensity/bias directly.

Parameter	Effect
Radius	How far samples reach in view space — larger is broader/softer, smaller stays in tight creases
Intensity	Darkening strength; raise to make occlusion more pronounced
Bias	Pushes samples off the surface to stop flat areas self-occluding (acne)
Half Resolution	Computes and blurs the occlusion at half width and height, for about a quarter of the cost, and lit meshes upsample it. The depth pre-pass stays full size

Because the effect only touches ambient light, it is most visible with a strong ambient source — raise **Global Illumination Intensity** or use IBL if SSAO looks too subtle in a scene lit mainly by direct light.

> **Scope and limitations**
> SSAO is computed for the main camera; render-to-texture cameras and terrain are currently excluded (terrain would exceed the shader's sampler limit). Normals are reconstructed from depth when SSAO runs alone; when SSR is also enabled SSAO reuses the SSR G-buffer's depth **and** geometric normals — a single shared geometry pass, and sharper at silhouettes. Enabling SSAO recompiles lit mesh shaders.

Screen-space reflections (SSR)



Screen-space reflections add real-time reflections of on-screen geometry — wet floors, polished metal, glossy surfaces — by marching the camera depth buffer in screen space and sampling the lit scene colour where a reflected ray hits. Like Godot and Unity, SSR is *energy-conserving*: where a ray finds a hit it **replaces** the surface's environment reflection — the IBL sky or a reflection probe — rather than adding on top of it, and where a ray misses (off-screen or occluded) that reflection remains as the fallback. So SSR refines what IBL and probes already provide instead of double-counting it.

When SSR is enabled the main camera first renders a small **G-buffer** geometry pre-pass — packed depth, view-space normal, roughness/metallic, base colour, and the material's ambient occlusion — then runs three fullscreen passes. Terrain (p. 310) evaluates its painted layers in this pass too, so a reflective wet patch reflects like one:

1. **March** — reflect the view ray about the G-buffer normal and step it through depth (with a binary-search refine) until it crosses on-screen geometry.
2. **Glossy blur** *(optional)* — blurs the reflection in proportion to surface roughness, so rough materials get soft reflections while mirrors stay sharp.
3. **Composite** — recomputes the surface's environment specular (the sky, or its probe blended over it) and blends the reflection over it with the correct GGX reflectance (including metal tint), darkened by the material's ambient occlusion and SSAO as the colour pass darkens it, writing the final image. Pixels with nothing reflected skip it.

The opaque colour pass is rendered into an offscreen buffer first so the march has a full scene-colour image to sample. SSR therefore requires a framebuffer destination — the editor viewport, a render-to-texture camera, or an engine framebuffer.

SSR is a scene setting:

```
scene.setSSREnabled(true);
scene.setSSRMaxDistance(8.0f);    // max ray length in view-space units
scene.setSSRThickness(0.5f);      // depth-compare tolerance (view-space units)
scene.setSSRMaxSteps(48);         // linear march sample count (quality vs cost)
scene.setSSRIntensity(1.0f);      // reflection strength multiplier
scene.setSSRBlur(0.0f);           // glossy blur amount [0..1] (0 = sharp/mirror)
```

```
scene.ssrEnabled = true
scene.ssrMaxDistance = 8.0
scene.ssrThickness = 0.5
scene.ssrMaxSteps = 48
scene.ssrIntensity = 1.0
scene.ssrBlur = 0.0
```

In the editor the same controls live under **Scene → Screen-Space Reflections (SSR)** in the Properties window, plus a **Debug View** dropdown for tuning.

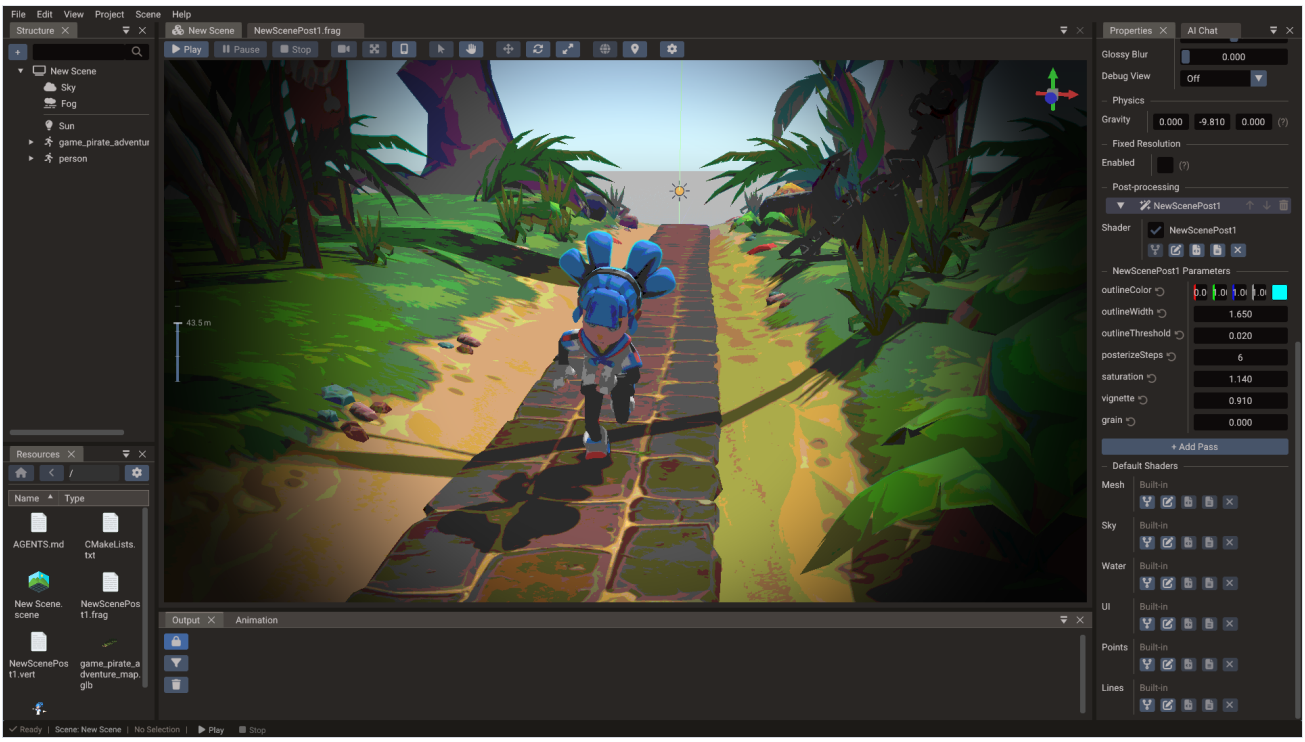
Parameter	Effect
Max Distance	How far a reflection ray travels in view space before giving up
Thickness	Depth tolerance for accepting a hit — smaller is stricter; larger fills gaps but can smear at contacts
Max Steps	March sample count — higher gives sharper/longer reflections at more cost; raise it if reflections miss thin contacts. The shader stops at 128
Intensity	Overall reflection strength
Glossy Blur	`0` = mirror; raise to blur reflections by surface roughness

Debug View renders one G-buffer channel full-screen: `Reflection` (the raw reflection buffer), `Normal`, `Roughness`, `Metallic`, `Albedo`, or `IBL Specular` (the recomputed environment term SSR blends against — compare it with the in-scene reflections to confirm the energy match). `Off` shows the normal render. Per-pixel roughness, metallic, and base colour come from the material factors and the metallic-roughness / base-colour textures.

> **Scope and limitations**
>
> SSR is screen-space, so it only reflects what is currently on screen: reflections fade out near the screen edges and cannot show off-screen or occluded geometry (IBL fills those in). Thin contact lines where an object meets a reflective surface can leave a small seam — raise **Max Steps** or lower **Thickness** to tighten it. SSR runs for the main camera only and is skipped when there is no framebuffer destination.

Post-processing

A scene can run an ordered chain of user **fullscreen passes** over its finished image. Each pass is a forked shader picked in the editor's **Post-processing** scene settings; see Custom Shaders — Post-process passes (p. 149) for authoring them.



The chain runs on the scene's active camera, after the opaque, transparent, UI, and SSR passes and before the fixed-resolution upscale. Enabling it redirects the colour pass into an offscreen buffer, exactly like SSR does, and the passes ping-pong between two buffers so a chain of any length costs two render targets. The last pass writes the real destination — the camera framebuffer, the fixed-resolution target, or the screen.

A pass reads the previous pass's output (`u_sceneColorTexture`) and may also sample depth (`u_depthTexture`), the SSR G-buffer (`u_gbufferTexture`), and ambient occlusion (`u_ssaoTexture`). When a pass samples depth while SSR and SSAO are both off, the engine runs the depth pre-pass for it, so depth-driven effects work on their own.

Values the pass declares with `#pragma param` lines become fields in the scene settings, and a pass that reads `Engine.resolution` or `Engine.time` gets them from the engine. Scripts set the values with `Scene.setPostProcessUniform`, which rewrites them without rebuilding the chain. Component shaders declare their values the same way; see Custom Shaders — Shader parameters (p. 147).

> **Scope**
>
> The chain runs for the scene's active camera only — secondary cameras such as mirrors and minimaps skip it, and so does a scene drawn as a layer, where a fullscreen pass would overwrite the scene below. Disabled passes are still exported, so a pass can be turned back on without recompiling.

Framebuffers and render-to-texture

A camera can capture its output to a texture instead of the screen — for minimaps, portals, mirrors, security monitors, or dynamic UI previews. Enable render-to-texture on the camera and pass its framebuffer wherever a texture is accepted (`Mesh`, `Image`, `Polygon`, and others have a `setTexture(Framebuffer*)` overload):

```
Camera minimapCam(&scene);
minimapCam.setRenderToTexture(true);
minimapCam.setFramebufferSize(512, 512);

// Use the camera's output as a texture on a UI image
Image preview(&uiScene);
preview.setTexture(minimapCam.getFramebuffer());
```

Pass `nil` (`nullptr` in C++) to these overloads to clear the texture.

In the editor, any **Texture** field can use a camera as its source instead of an image file: click the camera button on the field (or drag a camera entity from the Structure panel onto it). The chosen camera is switched to render-to-texture and its output feeds the slot — the basis for the manual mirror setup below. A camera used this way cannot also be the scene's main camera.

A related mechanism lets the **main camera itself** render offscreen at a fixed internal resolution that is then upscaled to the window — for pixel-art rendering or GPU performance scaling. That is a scene setting rather than a camera one; see Multiple Resolutions — Fixed resolution (p. 289).

Mirrors and planar reflections

A **Mirror** turns a flat surface into a true planar reflection — the kind used for mirrors, still water, and polished floors. It is built on render-to-texture: the engine renders the scene a second time from the viewpoint *reflected across the mirror plane*, then maps that image back onto the surface. For an animated water surface, use Water instead, which has a planar reflection option of its own.

The simplest way to add one is the **Mirror** entry in the Structure panel's create menu (or **Basic shape → Wall** plus a **Mirror** component). This creates an upright Wall (p. 471) whose surface normal faces the camera, with a `MirrorComponent` already attached. No camera or texture wiring is required — the component manages its own reflection camera internally.

In code, the Mirror (p. 448) class is the same thing: it derives from `Shape`, so it builds its own surface and attaches the component in one step.

C++

```
Mirror mirror(&scene);
mirror.createWall(10.0f, 10.0f);   // vertical quad, +Z normal (faces the camera)
mirror.setReceiveLights(false);    // optional: show the reflection unshaded
```

Lua

```
local mirror = Mirror(scene)
mirror:createWall(10.0, 10.0)
mirror.receiveLights = false
```

To make an **existing** mesh reflective instead of creating a new one, use `Mesh::setAsMirror()`, which works on any flat mesh; pass a normal — `setAsMirror(Vector3(0, 1, 0))` — when the surface isn't a Wall (for example a floor created with `createPlane`). Use `removeMirror()` / `isMirror()` to toggle or query it.

How it works

Stage	What happens
Reflected camera	Each frame the active camera is mirrored across the surface plane (entity position + normal). This preserves the handedness flip a real mirror has, so reflected geometry is rendered with **reversed face winding** to stay front-facing.
Projective sampling	The surface samples the reflection texture by screen position, not by mesh UVs, so the reflection stays correctly aligned regardless of the surface's size or placement.
Oblique clipping	The reflection camera's near plane is bent onto the mirror plane (Lengyel oblique projection), so geometry **behind** the mirror cannot leak into the reflection. The sky is excluded from this clip and reflects normally.

Lighting in the reflection

A mirror shows the side of an object that **faces the mirror** — usually not the side your key light hits. That side is lit only by whatever indirect lighting the scene has, so a single directional light plus a flat global illumination colour renders it as a washed-out silhouette: with a white ambient, a pale grey ghost of the object.

This is not a mirror artefact — the reflection is rendered with the same lights as the main view. The fix is the scene's indirect lighting:

- Enable **Receive IBL** (p. 441) on the reflected meshes so their shaded sides take colour from the sky environment instead of a flat ambient. This is the closest match to how objects look in the main view.
- Or add a low-intensity fill light pointing back toward the mirror, so the reflected side gets some direct light of its own.

The same applies to the mirror surface itself: leaving **Receive Lights** on shades the reflection with the mirror's own material, while turning it off shows the reflected image directly.

The Normal field

`MirrorComponent` exposes a single **Normal** — the reflecting surface direction in the mesh's local space (default `+Z`, matching a Wall), reachable in scripts as `Mirror::normal` (p. 448). It is transformed by the entity's rotation to build the world mirror plane, so rotating the mirror entity orients the reflection automatically.

> **If the reflection looks wrong**
>
> The reflection image is the same whichever way the normal points, but the behind-the-mirror clipping depends on its sign. If the reflection is clipped on the wrong side (showing geometry that should be hidden, or going mostly empty), flip the sign of the **Normal**.

Cost

A mirror renders the visible scene **one additional time per frame** into its own off-screen target — the same inherent cost planar reflections have in every engine. Use them deliberately:

- Prefer one hero mirror / water plane over many.
- A mirror never reflects itself, and reflections of other mirrors are not recursive.
- The reflection target matches the canvas resolution by default; lowering it trades sharpness for performance.

Water

A **Water** entity draws an animated water surface: waves, ripples, reflections, refraction, foam, and a view from below the surface, all from one component. It has its own render path, like the sky, so it needs no mesh, material, or camera wiring. Add one with the **Water** entry of the Structure panel's create menu, or in code with the Water (p. 496) class:

C++

```
Water water(&scene);
water.setSize(80.0f, 80.0f);
water.setWaveHeight(0.4f);
water.setPlanarReflection(true);
```

Lua

```
local water = Water(scene)
water:setSize(80, 80)
water.waveHeight = 0.4
water.planarReflection = true
```

Every field is listed in Properties — Water component (p. 100).

Waves and ripples

The surface is a grid **Size** wide (X) and deep (Z), centred on the entity at its height, with **Subdivisions** cells along each side. It stays level even when the entity is tilted; moving, rotating, or scaling the entity changes the area it covers. Four **Gerstner waves** displace the grid, all derived from the wave **Height**, **Length**,

Direction, and **Steepness**. They are computed in world space, so scaling the entity never stretches them, and longer waves travel faster, as they do on deep water. Steepness is lowered automatically where the crests would fold over.

Small **ripples** come from a tiling normal map (a built-in one unless you assign your own), sampled in two layers that drift across each other.

The engine evaluates the same waves on the CPU, so Water.getHeight (p. 502) and the buoyancy (p. 319) of 3D bodies follow the surface the next frame draws.

In the editor the waves hold still while the scene is stopped or paused and move during Play. An exported game always animates them.

Reflection and lighting

The water reflects the sky (or the background colour when the scene has no sky), mixed by a Fresnel term so it reflects more at grazing angles and shows its own colour when looked into. Scene lights add sun glints and light the water body; with **Receive Shadows** on and a shadow-casting light in the scene, shadows darken both. The water is fogged like any mesh when the scene has Fog.

Planar Reflection reflects the scene as well, through a reflection camera mirrored across the water plane, like a Mirror, drawn at half the canvas resolution. Only the main camera sees it; render-to-texture cameras, mirrors, and other waters' reflection passes see the water reflect the sky.

Depth effects

With **Depth Effects** on, the water reads the scene depth for shore foam, soft edges where it meets geometry, and a tint that deepens with the water under each pixel. The depth comes from the SSR G-buffer or the SSAO / post-process depth pre-pass when one of them already ran; otherwise the engine adds a depth pre-pass for the water. Only the main camera has the scene depth, so other cameras draw the water without these effects.

Foam

Foam comes from the shore, where the water is shallower than **Shore Foam** (with Depth Effects), from the wake (p. 504) of a moving Water Exclusion, and from whitecaps, set by **Crest Foam**. Dense foam covers the surface; as it thins it opens into cells, leaving a web along their edges that breaks into flecks. The cells are about a third of a meter across and blend into an even cover where they would be smaller than a pixel.

Whitecaps are short streaks along the crests. They run with the longest wave, come and go, leave thinner foam behind them, and gather where rougher patches drift through. Their size follows **Length**, and higher water breaks a little sooner. Far away, where the streaks would be smaller than a pixel, they blend into the even cover they would leave. A Crest Foam of `0.05` to `0.15` gives a breezy sea; towards `1` most of the surface is white.

A fork (p. 143) of the water shader keeps the foam it was forked with.

Refraction

With **Refraction** on, the scene behind the water shows through it, bent by the ripples and absorbed by the water it crosses: the clearest channel of the shallow colour fades over **Depth Fade**, the others faster, and the deep colour fills in the scattered light. For it, the main camera splits its colour pass in two:

1. Opaque geometry, the sky, and the transparencies on the far side of the water.
2. A copy of that image into a scene-copy target.
3. The water, sampling the copy, then the remaining transparencies.

With SSR on and a water in view, SSR is composited into the copy, so its reflections land under the water and the water is drawn over them. An exported game drawing straight to the screen renders into an offscreen target while a refracting water is drawn, then copies it to the screen.

Other cameras, and a water with refraction off, show the water alpha-blended over the scene instead.

Caustics

With refraction and **Depth Effects** on, the floor seen through the water carries **Caustics**: drifting webs of sunlight focused by the waves. The water places them on the floor it finds in the scene depth, so they follow any geometry under it. They are brightest in shallow water, fade as it deepens, and scale with the directional light reaching the surface, so an overcast or night scene shows little of them. **Caustics Scale** is the world size of one cell of the pattern; set **Caustics** to 0 to turn them off.

Transparent objects and water

The water writes depth, so a wave hides the trough behind it. Transparent objects — blended meshes, particles, and points — are therefore split per camera: an object entirely on the far side of the surface from the eye, within the water's area, is drawn before the water and seen through it; everything else, including an object that crosses the surface, is drawn after it.

Underwater

When the main camera goes below the surface, inside the water's area, the surface seen from below shows the sky only through a bright window overhead, and the rest of the scene fades into the water with distance, using the same shallow colour, deep colour, and **Depth Fade**. With Depth Effects off the fade is even, as if everything were Depth Fade away. The fade is a fullscreen pass with its own shader, `underwater.frag`, which can be forked (p. 145). Turn **Underwater** off on a water to skip it; while editing, the viewport's **Disable underwater** setting keeps the editor camera's view clear.

Water exclusions

A WaterExclusion (p. 503) keeps the water out of a volume, like a boat floating with its floor below the waterline. The water shader tests the world position of each fragment against the volumes over the water, so the cut follows the waves, shows in every view, including reflections and render-to-texture cameras, and leaves the waves in front of the boat drawn. A box or sphere is tested exactly. A hull, the convex hull of the entity's meshes, is kept as a 128 × 128 map of its bottom and top seen from above, one layer per hull in a shared texture array: sloped sides come out within a few millimetres and straight walls within a texel. Each water draw takes the 8 visible volumes nearest to the camera.

The same volumes stop the underwater fade for a camera inside them and the buoyancy of the bodies inside them.

Cost

Feature	Extra work
Grid	`(Subdivisions + 1)²` vertices, displaced in the vertex shader
Planar Reflection	The visible scene drawn once more, at half resolution
Refraction	One copy of the scene per frame, and the main pass split around it
Depth Effects	A depth pre-pass, only when SSAO, SSR, and post-process depth are all off
Underwater	One fullscreen pass while the camera is below the surface
Water Exclusion	A volume test per water fragment, with one texture lookup inside a hull's bounds

Each water compiles its own shader variant (p. 280) from the features it uses, and the scene can set a default water shader like it does for other types.

Shaders

Shaders are authored in GLSL and transpiled by the shader builder for each supported backend. Shader data files are generated at export time.

Each renderable type (Mesh, UI, Points, Lines, Sky, Water) has a built-in shader. In the editor you can **fork** any of them — per component, or as a scene-wide default for that type — and edit the GLSL; the engine keeps driving the variant system, lighting, and the shadow/depth/G-buffer passes. A Mesh can fork its depth shader too, so a colour fork that displaces vertices or discards keeps its shadows in step. A shader set on the component wins over the scene default, which wins over the built-in. The same fork mechanism provides the scene's post-process passes. See Custom Shaders (p. 143).

Built-in skinned variants bind bone matrices through a **storage buffer** (`sbo_skinning`) on Vulkan, Metal, and Direct3D 11, or through an **unfilterable RGBA32F bone texture** sampled with `texelFetch` on OpenGL / OpenGL ES. The colour, depth/shadow, and G-buffer passes share that path, so large skeletons are not capped by a uniform-block size. Shader reflection keeps the backend-specific storage-buffer bindings, which Metal requires for validation. The matrices come from the mesh's `SkinComponent`, an array sized by `MAX_BONES` (default 96); see 3D Graphics — GLTF compatibility and limits (p. 249). The editor revisions its compiled shader cache when this built-in shader interface changes, so an updated editor does not reuse incompatible older skinned variants.

GPU pipeline creation can be deferred by the graphics backend. If a pipeline variant fails creation or validation, Doriax skips that draw instead of submitting its uniform, binding, and draw commands against the failed handle. Diagnose the original pipeline creation message in the Output panel; the dependent validation-error cascade is suppressed.

Objects with the same shader variant, vertex layout, and render state share one GPU pipeline, and equal static vertex and index data is uploaded once and shared, so many copies of a mesh cost one set of GPU resources.

Supported graphics backends (`GraphicBackend` enum):

Backend constant	Target
`GLCORE`	Desktop OpenGL 3.3+
`GLES3`	OpenGL ES 3 (Android, WebGL2)
`D3D11`	Windows Direct3D 11
`METAL`	macOS and iOS Metal
`WGPU`	WebGPU

Sprite batching

Sprites drawn one after another share one draw call when they use the same texture, shader, and blending. A sprite qualifies when it is a single quad drawn unlit with the built-in mesh shader: no light reaches it (the scene has none, or its Mesh has **Receive Lights** off), neither the mesh nor the scene sets a custom mesh shader, and its texture is an image rather than a framebuffer. Instanced meshes, tilemaps, and terrains are drawn on their own and never join a sprite batch.

Batches follow draw order. Without a depth test, as in most 2D scenes, objects draw in hierarchy order, so keep sprites that share a texture or atlas next to each other: an object with another texture between them splits the batch.

A sprite that nothing changes keeps a copy of its batch data from frame to frame, so drawing it costs very little, and moving it only moves its corners. Changing it through the Sprite or Mesh API, such as a new texture region, color, or flip, makes it take the copy again on the next update. Only sprites keep a copy: other quads still batch but are rebuilt every frame, and so are billboards and sprites whose entity also has a camera, light, sound, model, or bone component. C++ code that writes a sprite's `MeshComponent` directly sets `SpriteComponent::needUpdateBatch`, see Writing components directly (p. 191).

GPU instancing

Draw large numbers of identical objects (trees, rocks, enemies) efficiently with mesh instancing. Each instance has its own transform (and optional color and texture region) stored in a GPU buffer, eliminating per-draw CPU overhead:

```
mesh.createInstancedMesh();
mesh.setMaxInstances(500);

for (int i = 0; i < 500; i++) {
    mesh.addInstance(Vector3(i * 2.0f, 0, 0), Quaternion(), Vector3(1, 1, 1));
}
```

Instances can be modified later with `updateInstance(index, ...)` and read back with `getInstance(index)`. Instance transforms may rotate, mirror, and scale non-uniformly; normals and tangents are transformed correctly, so a tree stretched taller is still lit as a tree.

Instancing uses geometry on the **same entity** as the instanced mesh. Multi-node GLTF models that keep child mesh entities will not draw instances until you merge static model meshes (p. 250) into the root (or build instances on a single-mesh entity / basic shape). In the editor, the Instanced Mesh panel warns when a model still uses the child-mesh layout.

Distance fade

Distance Fade shrinks instances to nothing between **Fade Start** and **Fade End**, both measured from the camera in the entity's model space. It is a vertex-shader scale, not a dither: the engine has no temporal anti-aliasing, so a stipple fade reads as dots on thin foliage. Turning it on selects a different shader variant (`Ifd`), while the two distances are plain uniforms that can be dragged freely.

On the built-in triangle mesh shaders — no custom colour or depth fork, no scene default mesh shader, and SSR off — instances past **Fade End** are also dropped from the draw list, so they cost neither vertices nor a fragment. The list keeps a small margin past it and is cut again only once the camera has moved beyond that margin, not on every frame it moves. Custom shaders and the SSR G-buffer still draw every instance and rely on the vertex scale alone.

Fade is what keeps a field of instances from popping at the edge of its draw distance; the terrain's foliage layers (p. 312) set it up for themselves from each layer's draw distance. From script, `Mesh::setDistanceFade` / `setFadeRange` expose the same controls as the Instanced Mesh panel.

Culling granularity

An instanced entity is culled as a whole: its bounds are rebuilt from its instances, so one entity holding a whole forest has a bounding box covering the map and is drawn in full every frame. Splitting the same instances into one entity per region — the terrain foliage uses chunks around the camera — gives the frustum something to reject.

That trades against batch size, because instancing batches per model and per entity. A region that offers every model variation ends up with two or three instances per draw call. Picking a small subset of the palette per region, deterministically from its coordinates, keeps the batches large while the whole map still uses every model.

Cull Instances (default on) then filters *inside* the entity. The main camera and each shadow atlas slot keep their own instance list: only instances that intersect that view's frustum are uploaded, nearest first and grouped by detail level. Mirrors and other render-to-texture cameras still draw the full batch. Turn culling off when a shader moves instances away from the bounds the CPU computed — those instances would otherwise vanish as soon as their authored positions leave the frustum. With it off the entity is not frustum-culled as a whole either, so the batch always reaches the GPU.

Cull Distance (default `0` = unlimited) drops instances farther than that many world units from the main camera from the same culled views, shadow slots included. Unlike distance fade it is a hard cut that keeps instances full-sized until they disappear, and it works with custom shaders and the SSR G-buffer. Hide the pop with fog or a distant representation. It follows **Cull Instances**, so turning that off disables the distance too.

Wind

Wind is a shader effect (p. 285). The scene parameter `Wind.velocity` is an air velocity in meters per second, set under **Shader Parameters** in the scene properties once a mesh sways, or from scripts. Meshes sway in it through the engine's `offsets/wind.glsl` vertex offset (p. 283). The lean grows faster than the wind: against 10 m/s it is about a sixth at 3 m/s and nearly twice at 15 m/s, so light air barely stirs a plant and a gale lays it over. Gusts roll downwind at the wind's speed, in fronts broken across it. The plant springs and shivers at a pace set by the air going past, and stays still in calm air. It moves the color, depth and G-buffer passes alike, so shadows, SSAO and SSR follow, and it moves no physics body.

```
scene->setShaderParameter("Wind", "velocity", Vector3(5.0f, 0.0f, -2.0f));

grass.setVertexOffsetShader("offsets/wind.glsl");
grass.setShaderUniform("windSway", 0.3f);
grass.setShaderUniform("windSwayHeight", 0.5f); // tips bend, the base at y = 0 stays put
grass.setVertexOffsetMax(0.8f);                 // its reach in 15 m/s of wind
```

Terrain foliage layers (p. 309) have a **Wind sway** of their own: it runs the same offset on the layer's instances, bending them up to their tips, with the culling reach set for winds up to 20 m/s. The sway runs on the scene's update time, so it stops when the game pauses, and in the editor it holds while the scene is stopped or paused.

See-through

See-through is a shader effect (p. 285): a mesh whose coverage is `coverage/see_through.glsl` dithers away where it stands between a camera and a target, so a character behind a tree or a roof stays in view. A script writes the target into the camera's `SeeThrough` view parameters, usually each frame from the player. The depth pre-pass and the SSR G-buffer drop the same pixels as the color pass, and shadow maps keep the whole mesh.

```
house.setCoverageShader("coverage/see_through.glsl");

// each frame, round the player's chest, for whichever camera renders the scene
scene->setMainViewShaderParameter("SeeThrough", "target", player.getWorldPosition() + Vector3(0,
0.9f, 0));
scene->setMainViewShaderParameter("SeeThrough", "radius", 2.5f);
```

Mesh detail (LOD)

Meshes can drop geometric detail as they recede, so a distant prop costs fewer triangles than the same mesh at the camera. It is **off by default**: every mesh draws its full geometry and no levels are built. Turning the scene's **Enabled** switch on is the only step needed; each mesh then follows it unless its **Detail Levels** is turned off.

While the switch is on, the engine builds extra index ranges with meshoptimizer[10] for each mesh — asynchronously on the thread pool, keyed by a hash of the geometry so identical meshes share the result — and turning it off frees their GPU buffers. Level 0 is the source; up to three simplified levels keep roughly 50%, 25%, and 12.5% of the source triangles, stopping early when a mesh cannot shrink further. `MAX_MESH_LODS` (default 4) is the compile-time cap.

10. https://github.com/zeux/meshoptimizer

A detail level is chosen from **projected geometric error**: the simplification's deviation from the source, in mesh units, is compared with how many world units one screen pixel spans at that distance. The scene's **Threshold** is how many pixels of error a simplified mesh may show (default `1`). Shadow maps use a fixed two-texel budget instead, so casters stay sharp enough for the PCF filter.

Control	Default	Effect
`Scene::setMeshLodEnabled`	`false`	Master switch; while off no levels are built
`Scene::setMeshLodThreshold`	`1.0`	Allowed screen-space error in pixels; lower keeps more detail
`Mesh::setLodEnabled`	`true`	Per-mesh opt-out (follows the scene switch); toggling it reloads the mesh
`Mesh::setLodBias`	`1.0`	Above 1 keeps detail longer; below 1 drops it sooner

Terrain and tilemaps keep their own LOD and are not simplified this way. Detail levels apply to the main camera and to shadow maps; mirrors and other render-to-texture cameras draw the source mesh. Instanced meshes pick a level *per instance* in those same views.

In the editor the scene controls live under **Scene → Mesh Detail (LOD)** when nothing is selected, and each mesh has **Detail Levels** / **Detail Bias** on the Mesh component.

```
scene.setMeshLodEnabled(true);
scene.setMeshLodThreshold(1.0f);

mesh.setLodBias(1.5f);       // keep this hero prop sharper than the default
statue.setLodEnabled(false); // this silhouette must never change
```

```
scene.meshLodEnabled = true
scene.meshLodThreshold = 1.0

mesh.lodBias = 1.5
statue.lodEnabled = false
```

Depth prepass

An optional **depth-only pass** runs on the main camera before the opaque colour draws, writing the same depth the colour pass will test. Later fragments fail the depth test instead of running the lit mesh shader, so each pixel is shaded once. That pays off when many opaque surfaces overlap in screen space — dense foliage, layered interiors — and costs an extra geometry pass when they do not.

The setting is off by default. Toggling it reloads meshes, because the prepass pipelines are built with each mesh. It runs only for the main camera and only while that camera has depth testing on; 2D and UI cameras skip it. A mesh whose colour shader is forked without a matching depth fork (p. 144) is left out of the prepass, so its vertices stay consistent with the colour pass.

This is separate from the small depth target SSAO (and some post-process passes) already build when they need a depth texture. Enabling the scene prepass does not replace those.

In the editor the toggle is **Scene → Depth Prepass**. From script:

```
scene.setDepthPrepassEnabled(true);
```

```
scene.depthPrepassEnabled = true
```

Performance guidelines

Area	Guideline
Draw calls	Reduce with instancing, atlases, and shared `.material` files; keep sprites that share a texture next to each other so they batch

Area	Guideline
Mesh LOD	Turn scene Mesh Detail on for triangle-heavy scenes; raise **Threshold** or lower a mesh's **Detail Bias** to drop triangles sooner
Instance culling	Keep **Cull Instances** on unless a shader moves instances off their bounds; split huge batches into regional entities
Depth prepass	Enable for heavy opaque overdraw (forests, interiors); leave off when most pixels are shaded once already
IBL cost	Environment maps are rebuilt when the sky texture changes; disable **Receive IBL** on distant or unimportant meshes
Shadow casters	Limit shadow-casting lights; cascade only when needed; turn off **Cast Shadows** on grass-scale terrain foliage (p. 312)
Transparent objects	Keep transparent draw counts low; sort correctly
Tilemaps	Chunk culling is automatic, so map size costs little; a tilemap is capped at 16 383 tiles
Mobile shaders	Simplify PBR (skip normal maps, lower cascade count)
Render targets	Minimize framebuffer resolution for off-screen effects
Mirrors	Each mirror re-renders the scene once per frame; keep one hero reflection and lower its target resolution if needed
Water	Keep **Subdivisions** as low as the waves allow; Planar Reflection re-renders the scene and Refraction copies it once per frame
SSR	Adds a G-buffer geometry pass plus fullscreen march/blur/composite passes; lower **Max Steps** for cost, and it shares its geometry pass with SSAO when both are on
Textures	Use compressed formats (ETC2/BC) on mobile/desktop respectively

The editor footer reports FPS, frame time, draw calls, and triangles for the last drawn frame. `Engine::getFrameStats()` returns the same counters from C++. For a repeatable capture with VSync forced off, use the `benchmark` (p. 169) command.

See also

- Camera (p. 414)
- Light (p. 424)
- Material (p. 734)
- Texture (p. 738)
- Fog (p. 422)
- Skybox (p. 472)
- ReflectionProbe (p. 468)
- Water (p. 496)
- Mesh (p. 436)
- RenderSystem (p. 594)

Shader Variants

The engine does not ship a single mesh shader. It compiles one **variant** per feature combination a scene actually uses — lit or unlit, with or without normals, tangents, skinning, vertex colors, shadows, instancing — and each variant is a separate compiled program. A build carries only the variants it was told to compile, so the set has to be known before the export runs.

How a variant is identified

A variant is a **shader type** plus a **property mask**. `mesh:Uv1,Puc,Shw,Nor,Tan,Aop` is the lit, textured, shadowed, opaque mesh shader with tangents; `depth:Ski` is the skinned depth pass that feeds shadow maps. That spelling is what `--shader` (p. 170) accepts and what the Export Window (p. 156) **Shaders** list shows.

A custom shader (p. 143) adds a third dimension: the fork is compiled per property combination as well, alongside the built-in variant it falls back to when the fork fails to build.

Where the set comes from

Each scene keeps its own list in the `shaderKeys` field of its `.scene` file. The editor refreshes that list when it saves the scene, by walking the renderable components and reading the variant each one resolved to.

The catch is that a component reports its variant only once **RenderSystem has loaded it**, and loading needs a live graphics context. The refresh therefore only sees what the editor has actually drawn. Two consequences:

- Saving **merges** into the stored list instead of replacing it, so a component that has not been rendered yet does not lose its variant from the file. Entries accumulate; the only ones a save drops are those of a forked shader whose files were deleted from the project, which no export could build. Drop other stale ones in the Export Window's **Shaders** list.
- A headless `doriax-editor export` (p. 167) has no renderer, so it discovers nothing and exports exactly what the `.scene` files already carry.

Opening the project in the editor and playing each scene once is what fills the list in. When Play finds variants that a scene file does not list yet, Stop adds them, including those of entities scripts created while it played, such as a HUD built from C++ or Lua, as long as they were drawn before Stop. A scene that only loaded during Play, such as a level opened from the title menu, gets just its `shaderKeys` rewritten, and the rest of its file stays as it was.

Each save also stores a `shaderKeysHash` of the rest of the scene and of the bundles its instances use. When the scene file or one of those bundles changed outside the editor, or the scene was saved before everything in it had loaded, the export warns:

```
Shader keys of Level1 may be out of date, and the game exits if it needs a missing shader. Open and
save these scenes in the editor to update them.
```

Opening the scene lets the editor draw what changed, and saving it then writes a complete list.

Property reference

A property name is a three-letter abbreviation of the feature it switches on, and its position in the list below is the bit it occupies in the numeric mask. Both spellings are accepted by `--shader` and the `set_export_shaders` AI tool (`mesh:Uv1,Nor` and `mesh:0x42` are the same variant), and the numeric form is what a scene file's `shaderKeys` entries hold.

`mesh`

Bit	Name	Feature
0	`Ult`	Unlit material
1	`Uv1`	Primary UV set
2	`Uv2`	Secondary UV set
3	`Puc`	Punctual lights
4	`Shw`	Shadow maps
6	`Nor`	Vertex normals
7	`Nmp`	Normal map
8	`Tan`	Vertex tangents
9	`Vc3`	Vertex color, RGB
10	`Vc4`	Vertex color, RGBA
11	`Txr`	Texture rect
12	`Fog`	Fog
13	`Ski`	Skinning
14	`Mta`	Morph targets
15	`Mnr`	Morph normals
16	`Mtg`	Morph tangents
17	`Ter`	Terrain
18	`Ist`	Instancing
19	`Ibl`	Image-based lighting
20	`Mir`	Mirror
21	`Sao`	SSAO
22	`L2d`	2D lights
23	`S2d`	2D shadows
24	`Ams`	Alpha mask
25	`Aop`	Alpha opaque
26	`Ifd`	Instance distance fade
27	`Tpb`	Terrain PBR layers
28	`Dlo`	Directional lights only

Bit 5 is reserved: it used to select the PCF shadow filter, which is uniform-driven now (Scene shadow quality).

A scene whose lights are all directional draws built-in lit meshes with `Dlo`, which leaves out the point and spot light code, also when the mesh only adds effect functions. An export ships every such variant both with and without it, so a script can still add a point or spot light at runtime.

`depth`

Bit	Name	Feature
0	`Tex`	Texture (alpha-masked shadow casters)
1	`Ski`	Skinning
2	`Mta`	Morph targets
3	`Mnr`	Morph normals
4	`Mtg`	Morph tangents
5	`Ter`	Terrain
6	`Ist`	Instancing
7	`Ams`	Alpha mask

Bit	Name	Feature
8	`Ifd`	Instance distance fade
9	`Uv1`	Primary UV set for a coverage function (p. 284)

The `gbuffer` type takes `Uv1` at bit 12 for the same reason.

Shader effects (p. 282) need no bit of their own: the vertex offset and the coverage function are part of the custom shader name that follows `@` in a spec, as `fork#vertexOffset#coverage`, for example `mesh:Uv1,Puc,Nor@#offsets/wind.glsl` or `depth:Uv1@##coverage/dissolve.glsl`.

`ui`, `points`, `lines`

Type	Bit 0	Bit 1	Bit 2	Bit 3
`ui`	`Tex` texture	`Ftx` font texture	`Vc3`	`Vc4`
`points`	`Tex` texture	`Vc3`	`Vc4`	`Txr` texture rect
`lines`	`Vc3`	`Vc4`	-	-

`water`

Bit	Name	Feature
0	`Fog`	Fog
1	`Dep`	Scene depth (**Depth Effects**)
2	`Prf`	Planar reflection
3	`Shw`	Shadow maps
4	`Rfr`	Refraction

`Fog` and `Shw` follow the scene: a Water (p. 496) sets them when the scene has a Fog component, or a shadow-casting light and **Receive Shadows** on. The others follow the water's own switches.

`sky`, `blit`, `ssao`, `ssaoblur`, `ssr`, `ssrblur`, `composite`, `shadow2d`, `postprocess`, `underwater`, and `probefilter` have no properties; name the type on its own.

Combinations that are easy to miss

Several properties come from the scene or the frame rather than from the model, so a variant list assembled by hand tends to be short by a factor of two or four:

- **`Sao`** is set on every lit mesh in a scene with SSAO enabled, terrain included.
- **`Ibl`** is off until the sky's environment maps finish generating, so the same mesh needs the variant both with and without it - the first frames are drawn before they exist.
- **`Ifd`** is a separate program from `Ist`, and terrain foliage chunks always set both.
- **`Vc4`** turns on for any model whose glTF carries `COLOR_0`, which many scanned assets do without it being obvious.
- **`Uv1`** needs a texture, not just UVs: a generated mesh with texture coordinates but an untextured material does not set it, unless the mesh has a coverage function, which reads the UV set in every pass.

When a build is missing one

An exported game loads each variant from its precompiled set. When one is absent, it draws with the nearest variant that leaves out some optional features, such as fog, shadows, or a texture, and logs a warning:

```
Shader mesh_Uv1Uv2PucShwNorTanFogSkiSaoAop is missing, using mesh_PucShwNorTanFogSkiSaoAop
```

If no such variant exists either, it prints the missing specs together with the command that generates them, then exits:

```
Doriax is missing precompiled shaders in:
assets/shaders

Missing shaders:
mesh:Uv1,Uv2,Puc,Shw,Nor,Tan,Ski,Aop; mesh:Uv1,Puc,Shw,Nor,Tan,Aop

Bundle precompiled shaders for graphic backend: opengl
To generate them manually with Doriax Editor:

> doriax-editor shaders --out "assets/shaders" --backend opengl --shader
"mesh:Uv1,Uv2,Puc,Shw,Nor,Tan,Ski,Aop" --shader "mesh:Uv1,Puc,Shw,Nor,Tan,Aop"
```

The listed specs are authoritative — they are what the running scene asked for. Fix it in whichever way suits the workflow:

Situation	Fix
Exporting from the editor	Play each scene once, then export. The **Shaders** list fills itself
Scripted or CI export	Pass the specs with `--shader`, which replaces what the scenes report
Scenes generated by a tool	Write the `shaderKeys` entries into the generated `.scene` files

For a CI pipeline the `--shader` route is the reliable one: it does not depend on whether a machine with a GPU ever opened the project.

See also

- Custom Shaders (p. 143) — forking a built-in shader
- Command-Line Tools (p. 167) — `export`, `shaders`, and `benchmark` subcommands
- Rendering Pipeline (p. 272) — what the built-in shaders do

Shader Effects

Shader effects change how a mesh draws without forking its shaders. A **vertex offset** moves its vertices, a **coverage** function removes part of its surface, and **shader parameters** feed both with values that belong to the mesh, the scene or a camera. The engine runs them in the color, depth and G-buffer passes alike, so shadows, SSAO and screen-space reflections agree with what is drawn. Shader parameters also work in forks of every shader type, as Custom Shaders (p. 147) explains.

The engine's wind and see-through are built this way, and so are three small examples it ships: a dissolve, a height cut, and grass pushed aside by a character.

How it fits together

Piece	What it is	Set on
Vertex offset	One `.glsl` with `vec3 getVertexOffset(vec3 localPosition, vec3 worldPosition)`	Mesh `vertexOffsetShader`
Coverage	One `.glsl` with `float getCoverage(CoverageContext context)`	Mesh `coverageShader`
Object parameters	Values of one component, declared with `#pragma param`	The component's shader uniforms
Scene parameters	`Collection.name` values the whole scene shares, declared with `#pragma scene`	Scene properties, `Scene::setShaderParameter`
View parameters	`Collection.name` values of one camera, declared with `#pragma view`	Camera properties, `Camera::setShaderParameter`

A mesh takes one vertex offset and one coverage function. To combine several effects of the same kind, write one function that calls the helpers of each (see Writing your own). The vertex offset runs after skinning, morph targets and instancing and before the instance distance fade. The coverage runs at the start of the fragment stage, before any texture is sampled and before the material's alpha test.

Declaring parameters

Parameters are declared with `#pragma` lines. The GLSL compiler skips a pragma it does not know, so an effect file stays plain GLSL:

```
#pragma param float strength = 0.5 [0, 2] "How far the top moves, in world units"
#pragma param vec4 tint = 1, 0.8, 0.6, 1 color
#pragma scene vec3 Wind.velocity "Air velocity in m/s"
#pragma view float SeeThrough.radius = 0 [0, 20]
```

The type is `float`, `vec2`, `vec3`, `vec4`, `int`, `ivec2`, `ivec3` or `ivec4`. The numbers after `=` are the default, and a single number fills a whole vector. Integer types travel as floats and are exact up to 16777216. `[min, max]` is the range of the inspector field, `color` shows a color picker, `runtime` marks a value that a script or the engine writes (the inspector shows it read-only), and the quoted text is the tooltip.

The code reads each parameter by the name it was declared with:

Declared as	Read in GLSL as
`#pragma param float strength`	`params.strength`
`#pragma scene vec3 Wind.velocity`	`Wind.velocity`
`#pragma view float SeeThrough.radius`	`SeeThrough.radius`

The collection keeps unrelated values apart: `Wind.speed` and `Water.speed` are two parameters. A collection is either a scene or a view one. The build stops with an error that names the file and line when two declarations

give a parameter different types, or use a collection in both scopes, even when one is in the vertex file and the other in the fragment file. The shader's **Parameters** section shows the same error before any build. Collection and parameter names take letters, digits and underscores, and `params` and `Engine` are not collection names.

Declarations are read from the shader file and every file it includes, whatever `#if` surrounds them. A comment may follow one on its line. The shader builder packs the declarations of each stage into one uniform block, `u_vs_params` or `u_fs_params`, with the engine values the code reads. The mesh color pass appends them to its material block instead, so effects and collections take no extra uniform block slots there. Changing a value rewrites that block at the next draw. It never rebuilds a shader or reloads a component.

Engine values

Two values need no declaration. A shader that reads them gets them:

Value	Contents
`Engine.time`	Seconds since the scene started. The clock holds while the editor scene is stopped or paused, and while the game pauses, and starts over when Play mode stops, so each Play session begins like the exported game. For gameplay-driven time, set a parameter of your own from a script.
`Engine.resolution`	`xy` = width and height of the target the shader draws into (the camera's framebuffer, the fixed-resolution target, a shadow slot, a reflection probe face, or a post-process target), `zw` = `1 / width` and `1 / height`.

Where values come from

An **object parameter** reads the component's shader uniform of the same name, then the value of the scene default shader when the component inherits it, then the declared default. A post-process pass keeps its own values.

A **scene parameter** reads the runtime value a script set, then the value saved with the scene, then the declared default.

A **view parameter** reads, for the camera a pass is drawn for:

1. the runtime value of that camera, from `Camera::setShaderParameter`;
2. the runtime value of the main view, from `Scene::setMainViewShaderParameter`, while that camera renders the scene;
3. the value saved on its Camera component;
4. the declared default.

Runtime values are never saved: stopping Play mode or loading the scene again brings back the saved ones, and `Wind.travel` and the effect clock start over. Each scene keeps its own values, and so does each copy of it that Play mode runs. Shadow maps are fitted to the camera that renders the scene, so an effect sees that camera there. A reflection probe capture has no camera and reads the declared defaults of view parameters.

Vertex offsets

```
vec3 getVertexOffset(vec3 localPosition, vec3 worldPosition);
```

It returns how far the vertex moves, in world units. `localPosition` is the vertex in the mesh's own space after skinning, morph targets and instancing, and `worldPosition` is where it lands before the move. `Engine.time` is the seconds since the scene started (see Engine values).

The mesh's **Max Offset** (`vertexOffsetMax`) is the longest offset. Longer ones are clamped to it, and culling widens the mesh bounds and each instance's sphere by it, so moved geometry is not cut at the screen edge or out of a shadow cascade. At 0 nothing is clamped and the bounds stay those of the mesh data. Lighting keeps the normals of the unmoved surface, and terrain ignores vertex offsets.

Coverage

```
float getCoverage(CoverageContext context);
```

It returns how much of the fragment stays: 1 keeps it and 0 removes it. In between, opaque and alpha-tested surfaces keep a steady 4×4 dithered share of their pixels, and blended surfaces multiply their alpha instead. The coverage and the material's alpha test are separate tests that both have to pass.

`CoverageContext` field	Meaning
`vec3 worldPosition`	The fragment in world space.
`vec2 uv`	The mesh's first UV set as stored, before sprite or instance texture rects; 0 when it has none. Every pass interpolates it for a mesh with coverage.
`vec3 eye`	Position of the camera the pass draws for.
`vec3 forward`	Its view direction, normalized.
`bool perspective`	False for an orthographic camera.
`vec2 pixel`	The fragment pixel counted from the corner of the view, the same in every pass, so the depth passes drop the pixels the color pass drops on any viewport offset or flipped target.

The depth pre-pass and the SSR G-buffer cut what the color pass cuts, so SSAO, reflections and depth based post-processing match it. Shadow maps keep the whole silhouette unless the mesh's **Cut Shadows** (`coverageShadows`) is on: a see-through wall should still cast its shadow, but a dissolving one should not. A skipped coverage still applies the material's alpha mask to the shadow.

Coverage functions can use the shapes in `includes/coverage_helpers.glsl`, which every coverage file sees:

Helper	Returns
`coverageBelowHeight(worldPosition, height, softness)`	1 below the height, 0 above, with a dithered band of the given width.
`coverageNoise(position)`	A smooth value noise in [0, 1].
`coverageViewOpening(worldPosition, eye, forward, perspective, target, radius, opacity, margin)`	The see-through opening: a cone from a perspective eye, a cylinder along an orthographic view.

A file can suggest the shadow policy with `#pragma coverage_shadows apply` or `keep`. The editor sets **Cut Shadows** from it when the file is picked.

The engine's effects

File	Parameters	What it does
`offsets/wind.glsl`	`windSway` (0.3), `windSwayHeight` (0); scene `Wind.velocity`, `Wind.travel`	Sways the mesh in the wind, with gusts that roll downwind.
`offsets/interaction.glsl`	`pushStrength` (0.5), `pushHeight` (1); scene `Interaction.position`, `Interaction.radius` (0)	Bends the mesh away from a point.
`coverage/see_through.glsl`	view `SeeThrough.target`, `radius` (0), `opacity` (0.3), `margin` (0.5)	Opens a view to a target behind the mesh.
`coverage/height_clip.glsl`	scene `Clip.height` (1000), `Clip.softness` (0)	Cuts away everything above a height.
`coverage/dissolve.glsl`	`dissolve` (0), `dissolveScale` (4), `dissolveEdge` (0.05)	Dissolves the mesh into world-space noise.

Wind

`windSway` is the bend in 10 m/s of wind, in world units. With `windSwayHeight` above 0 the bend grows as the square of the model height up to that value, so a plant modelled with its origin at its base keeps its foot still. The engine adds the meters of air blown past to `Wind.travel` each update, which moves the gusts at the wind's speed without a jump when it changes. Set the mesh's Max Offset to about `windSway × 2.5` for winds up to 15 m/s, or `windSway × 4` up to 20 m/s.

Terrain foliage layers (p. 309) keep their **Wind Sway** control: it runs this offset on the layer's instances, bending them up to their tips.

```
scene->setShaderParameter("Wind", "velocity", Vector3(5.0f, 0.0f, -2.0f));

grass.setVertexOffsetShader("offsets/wind.glsl");
grass.setShaderUniform("windSway", 0.3f);
grass.setShaderUniform("windSwayHeight", 0.5f);
grass.setVertexOffsetMax(0.8f);
```

```
scene:setShaderParameter("Wind", "velocity", Vector3(12, 0, 0)) -- a gale, until cleared
scene:clearShaderParameter("Wind", "velocity")                  -- back to the saved wind
```

See-through

The opening is a cone from the camera to a sphere of `radius` around the target, so it keeps one size on screen, and a cylinder along the view for an orthographic camera. It is full in its inner half and fades out in the outer half. `opacity` is what stays at the center. Only what lies in front of the target opens, starting `margin` before it and full at twice that, so the ground under a character stays whole. A radius of 0 turns it off.

The engine does not track the target: a script writes it each frame. The main view values go to whichever camera renders the scene, so a camera switch keeps the opening:

```
// in a C++ script deriving from Object, attached to the player
void PlayerReveal::onUpdate(){
    Vector3 chest = getWorldPosition() + Vector3(0.0f, 0.9f, 0.0f);
    scene->setMainViewShaderParameter("SeeThrough", "target", chest);
    scene->setMainViewShaderParameter("SeeThrough", "radius", 2.5f);
    scene->setMainViewShaderParameter("SeeThrough", "margin", 0.3f);
}
```

```
-- the same each frame from Lua, with player an Object handle
scene:setMainViewShaderParameter("SeeThrough", "target", player:getWorldPosition() + Vector3(0, 0.9,
0))
scene:setMainViewShaderParameter("SeeThrough", "radius", 2.5)
```

With two players on split cameras, give each camera its own target:

```
leftCamera:setShaderParameter("SeeThrough", "target", playerOne:getWorldPosition())
rightCamera:setShaderParameter("SeeThrough", "target", playerTwo:getWorldPosition())
```

Radius, opacity and margin can also be saved on the camera in the editor, under its **Shader Parameters**, leaving only the target to the script.

Dissolve, height cut and push

```
-- fade a defeated enemy out over a second, shadow included
enemy.coverageShader = "coverage/dissolve.glsl"
enemy.coverageShadows = true
enemy:setShaderUniform("dissolve", math.min(elapsed, 1.0))

-- hide the floors above the player in every building of the scene
scene:setShaderParameter("Clip", "height", player:getWorldPosition().y + 2.5)

-- grass bends away from the player
scene:setShaderParameter("Interaction", "position", player:getWorldPosition())
scene:setShaderParameter("Interaction", "radius", 1.5)
```

Writing your own

An effect file is a plain `.glsl` in the project. It finds its includes like a fork: first in its own folder, then the project root, then the engine library. This one bobs a boat on the water and sways its sail, reading a scene parameter of its own and the engine's wind:

```
// Bobs the mesh on the water, with the wind on top
#pragma param float bob = 0.05 [0, 1] "Height of the bobbing, in world units"
#pragma scene float Water.level "World height of the water surface"
#include "includes/wind.glsl"

vec3 getVertexOffset(vec3 localPosition, vec3 worldPosition){
    float wave = sin(worldPosition.x * 3.0 + Engine.time * 2.0) * params.bob;
    float nearWater = step(worldPosition.y, Water.level + 0.5);
    return vec3(0.0, wave * nearWater, 0.0) + getWindSway(localPosition, worldPosition, 0.2, 1.0);
}
```

A coverage function the same way:

```
// Keeps only what is above the water
#pragma param float edge = 0.1 [0, 1] "Height of the dithered edge"
#pragma scene float Water.level
#pragma coverage_shadows apply

float getCoverage(CoverageContext context){
    return smoothstep(Water.level - params.edge, Water.level, context.worldPosition.y);
}
```

To change an engine effect, copy it into the project (see below) and edit the copy.

In forks

A color or depth fork runs effect functions when it keeps the lines of the built-in shader that include `includes/vertex_offset.glsl` and `includes/coverage.glsl`, the `loadShaderParameters()` call at the top of `main()`, and in the color fork the `SHADER_PARAMS_FS_MEMBERS` line of its material block. The build warns when a mesh has an effect function its fork cannot run.

A fork can declare and read parameters in its own code too, after that call, and so can a fork of any other shader type:

```
#pragma scene float Weather.wetness = 0 [0, 1] "How wet surfaces look"
#pragma param vec4 wetTint = 0.6, 0.65, 0.7, 1 color
...
color.rgb = mix(color.rgb, color.rgb * params.wetTint.rgb, Weather.wetness);
```

Every fork that declares `Weather.wetness` reads the same scene value, so one script call can wet the meshes, the water and a post-process pass together.

In the editor

The **Shaders** section of a Mesh component has a **Vertex Offset** row and a **Coverage** row. Each row can create a file from a template, use one of the engine's effects (shown as *built-in*), pick a project file, copy the built-in effect into the project to change it, open the file, or remove it. A `.glsl` can also be dropped on the row. **Max Offset** shows under a vertex offset and **Cut Shadows** under a coverage function.

Below the table, each fork and effect file of the component has its own **Parameters** section with the fields its `#pragma param` lines declare, with their ranges, colors and tooltips. A field nobody set shows the value inherited from the scene's default shader, else the declared default, and the arrow next to a set field drops it. The section also lists the scene and view parameters the file reads. Saving the file updates the fields. A forked default shader shows its fields in the scene's **Default Shaders**, where the components using it inherit them, and a post-process pass shows them under its header.

The scene properties list every scene parameter the scene's shaders declare under **Shader Parameters**, the wind among them once something sways, and save the values set there. A camera's properties list the view parameters the same way. Values nothing declares any more stay listed until their arrow removes them. Every edit is undoable.

Export

The effect functions are part of the shader name, written `fork#vertexOffset#coverage`, so each combination compiles to its own variant and each pass gets the variant it needs. Saving a scene records the variants its meshes use, and an export builds and ships those, with any project effect files and the files they include. A composition that only appears at runtime, such as a script giving a mesh a coverage function, goes in the export overrides like any other variant:

```
mesh:Uv1,Puc,Nor@##coverage/dissolve.glsl
depth:Uv1@##coverage/dissolve.glsl
mesh:Uv1,Puc,Nor@shaders/myMesh#shaders/sway.glsl#coverage/see_through.glsl
```

The same specs work with `doriax-editor export --shader`. A mesh with a coverage function and a UV set uses the `Uv1` variants of the depth and G-buffer passes.

Multiple Resolutions

Games run on an enormous range of displays: phones in portrait and landscape, laptops, 4K monitors, ultrawide screens, and resizable desktop windows. Doriax separates the size your game is *designed* for from the size it is *displayed* at, so one project can look correct everywhere. This page covers the three concepts involved and how to combine them:

- **Canvas size and scaling mode** — the logical coordinate space your game is designed in, and how it maps onto the real window (project-wide, set on `Engine`).
- **View rect** — the region of the window the game actually occupies after scaling (computed automatically; may include letterbox bars).
- **Fixed resolution** — an optional per-scene setting that renders the game at a fixed internal resolution and upscales the result to the window, independent of the display's real pixel count.

Canvas size

The **canvas** is the logical coordinate space of your game — the size you design against. Positions of 2D objects, UI elements, and input coordinates are all expressed in canvas units, not physical pixels. Pick a canvas size that matches your target design (for example 1280×720) and the engine maps it to whatever window or screen the game runs on:

Lua

```lua
Engine.setCanvasSize(1280, 720)
Engine.scalingMode = Scaling.LETTERBOX   -- scalingMode is a property in Lua
```

C++

```cpp
Engine::setCanvasSize(1280, 720);
Engine::setScalingMode(Scaling::LETTERBOX);
```

In editor projects, the canvas size and scaling mode are set in **Project Settings** and applied automatically to exported builds. Both can also change while the game runs: UI and auto-resized cameras follow the new canvas right away.

The canvas is independent of the OS window itself: the initial window size, state (windowed, maximized, or fullscreen), resizability, and title of a desktop build are separate project settings. See Project Workflow — Window (p. 54).

Scaling modes

The scaling mode decides what happens when the window's aspect ratio does not match the canvas:

Scaling mode	Behavior
`FITWIDTH`	Keeps the canvas width fixed; the height floats to match the window's aspect ratio. More or less of the world is visible vertically.
`FITHEIGHT`	Keeps the canvas height fixed; the width floats.
`LETTERBOX`	Preserves the full canvas and its aspect ratio; empty bands fill the leftover space.
`CROP`	Preserves the aspect ratio and fills the window completely; the canvas overflows and its edges are cropped.
`STRETCH`	Stretches the canvas to the window size. No bars and no cropping, but the image distorts when aspect ratios differ.
`NATIVE`	The canvas equals the physical output size. Coordinates are real pixels.

Choosing between them is a design decision:

- `FITWIDTH` / `FITHEIGHT` suit games that can safely reveal more of the world on one axis (many 2D side-scrollers fix the height and let the width breathe, and vice versa).
- `LETTERBOX` guarantees every player sees exactly the same framing — good for puzzle boards and fixed compositions. The bands take the scene background color.
- `CROP` guarantees full-screen coverage — keep important gameplay elements away from the edges.
- `STRETCH` is rarely what you want unless aspect ratios are guaranteed to match.
- `NATIVE` fits tools and UI-heavy applications more than games.

The result of canvas + scaling is the **view rect**: the on-screen rectangle where the game is drawn. Input events are converted from window coordinates back into canvas coordinates through the same mapping, so your gameplay code never needs to know the real window size. UI elements additionally use anchors (p. 294) to adapt their layout inside the canvas.

> **Tip**
> You can react to canvas changes (window resize, device rotation) with the `Engine::onCanvasChanged` event, for example to reposition elements that are not anchor-driven.

Fixed resolution

Canvas scaling controls *coordinates*, but the scene is still rasterized at the real window resolution — a 4K monitor renders roughly eight times more pixels than a 720p one. The **fixed resolution** setting decouples the *render* resolution from the display: the scene's main camera renders into an internal buffer of an exact size you choose (say 320×180 or 640×360), and that image is then upscaled to fill the view rect. The player's desktop resolution never changes.

Two situations call for it:

- **Pixel-art and retro games** — render at a small authentic resolution and upscale with nearest-neighbor filtering for crisp, chunky pixels, no matter the display.
- **Performance** — fragment-shader cost scales with pixel count. Rendering at a lower internal resolution and upscaling is one of the most effective ways to speed up a GPU-bound game, especially on mobile and integrated graphics.

Fixed resolution is a **scene setting**. In the editor it lives in the scene's Properties panel (select nothing so the scene itself is shown), under **Fixed Resolution**:

Setting	Default	Description
Enabled	off	Renders the main camera at the fixed size and upscales to the view rect.
Width / Height	640 × 360	The internal render resolution in pixels.
Filter	Nearest	How the image is sampled during the upscale: **Nearest** keeps hard pixel edges (pixel-art look), **Linear** interpolates smoothly.

The same settings are available from code:

Lua

```lua
scene.fixedResolutionEnabled = true
scene:setFixedResolutionSize(320, 180)
scene.fixedResolutionFilter = TextureFilter.NEAREST
```

C++

```cpp
scene.setFixedResolutionEnabled(true);
scene.setFixedResolutionSize(320, 180);
scene.setFixedResolutionFilter(TextureFilter::NEAREST);
```

How it interacts with canvas scaling

Fixed resolution layers *under* the canvas system rather than replacing it. The canvas and scaling mode keep deciding the view rect, framing, letterbox bars, and input mapping exactly as before; the fixed resolution only changes how many pixels the scene is rasterized with before being stretched into that view rect. In practice:

- Keep the **canvas aspect ratio equal to the fixed resolution aspect ratio** (for example canvas 1280×720 with fixed resolution 320×180 — both 16:9). Mismatched ratios produce non-square pixels, since the image is stretched to the view rect.
- Letterbox bars still come from the scaling mode and still use the scene background color.
- Object positions, physics, and input are unaffected — they operate in canvas coordinates regardless of the render resolution.

Rules and behavior

- **Main scene only.** The setting takes effect on the scene shown with `Engine::setScene`. Scenes added as layers (`Engine::addSceneLayer`) always render at native resolution, even if their own setting is enabled.
- **Crisp UI over a low-res game.** Because layers stay native, the standard recipe for a pixel-art game with sharp text is: enable fixed resolution on the game scene and put the HUD/menus in a separate UI scene added as a layer on top.
- **Editor preview.** In the editor, the scene viewport stays at native resolution while editing (so gizmos and selection remain precise) and switches to the fixed resolution when you press **Play**.
- **Screen-space effects follow the fixed resolution.** SSAO and SSR run at the internal resolution, so they get the same pixelated look — and the same performance savings — as the rest of the scene.
- The image is rendered once per frame into the internal buffer and upscaled in a single fullscreen pass; the overhead is negligible next to the savings of shading fewer pixels.

Changing resolution at runtime

Width, height, and filter can be changed at any time — the internal buffer is resized on the next frame with no interruption. This makes a resolution option in a settings menu, or even dynamic resolution scaling, straightforward:

Lua

```lua
-- settings menu: apply a render-scale preset
function applyRenderScale(scene, scale)
    local w = math.floor(1280 * scale)
    local h = math.floor(720 * scale)
    scene:setFixedResolutionSize(w, h)
end

applyRenderScale(scene, 0.5)  -- render at 640x360, upscale to the window
```

C++

```cpp
// settings menu: apply a render-scale preset
void applyRenderScale(Scene& scene, float scale) {
    unsigned int w = (unsigned int)(1280 * scale);
    unsigned int h = (unsigned int)(720 * scale);
    scene.setFixedResolutionSize(w, h);
}

applyRenderScale(scene, 0.5f); // render at 640x360, upscale to the window
```

Toggling `fixedResolutionEnabled` at runtime also works, but is heavier than a size change: the engine rebuilds render pipelines for the scene's objects, which can cause a brief hitch. Prefer leaving it enabled and adjusting the size instead.

Pixel-art checklist

- Fixed resolution at your sprite-native size (e.g. 320×180) with **Nearest** filter.
- Canvas size an integer multiple of it with the same aspect ratio (e.g. 1280×720).
- `LETTERBOX` scaling so the framing is identical on every display.
- HUD and text in a separate UI scene layered on top for full-resolution crispness — or inside the fixed-resolution scene if you want the UI pixelated too.

Common setups

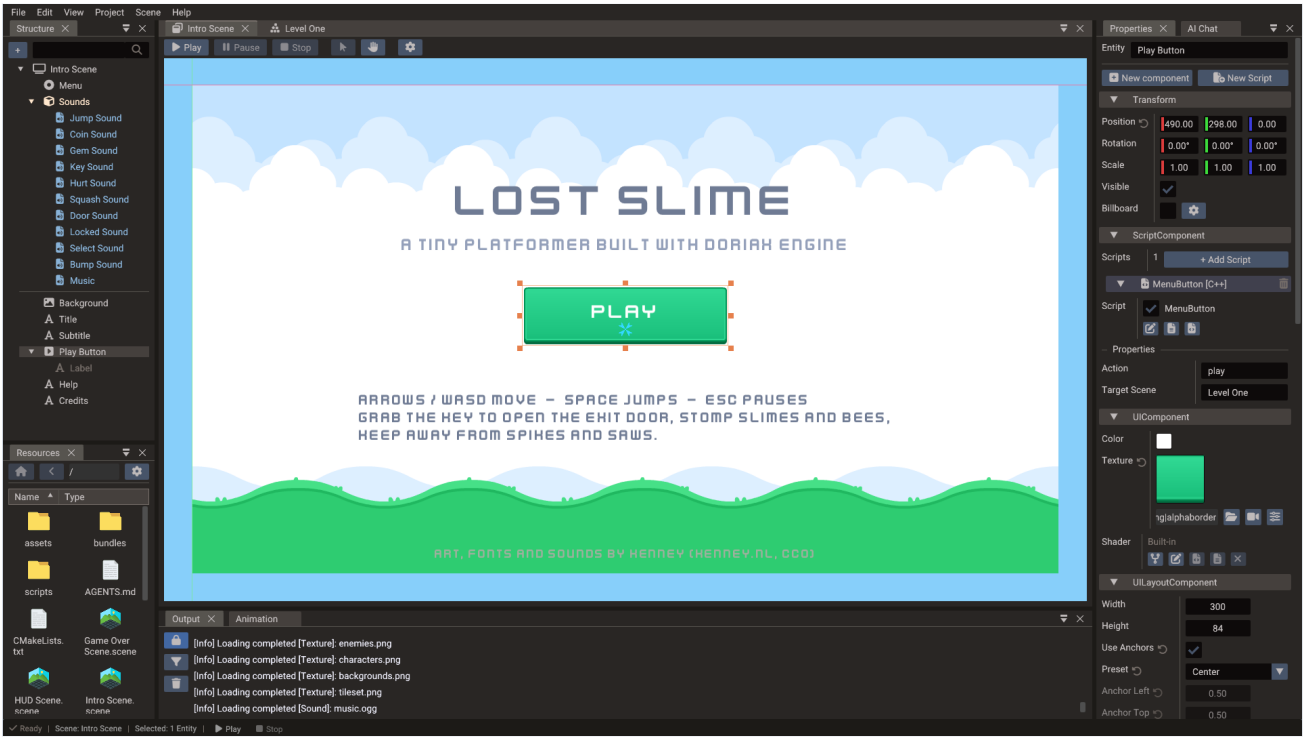
Goal	Canvas + scaling	Fixed resolution
Standard 2D/3D game, sharp at any size	Design-size canvas, `FITWIDTH`/`LETTERBOX`	Off
Pixel-art / retro	Canvas multiple of the art size, `LETTERBOX`	On, art-native size, Nearest
GPU-bound game on weak hardware	Design-size canvas, any mode	On, reduced size (e.g. 70% of 1080p), Linear
Identical framing on all devices	`LETTERBOX`	Optional
Tools / UI applications	`NATIVE`	Off

Next steps

- User Interface (p. 292) — anchors and layout inside the canvas.
- 2D Graphics (p. 243) — cameras and coordinates for 2D scenes.
- Rendering Pipeline (p. 258) — render-to-texture and screen-space effects.
- Scene reference (p. 390) — the full fixed-resolution API.
- Engine reference (p. 361) — canvas size, scaling mode, and events.

User Interface

Doriax includes a complete runtime UI system for menus, HUDs, overlays, and in-game controls. UI entities are regular ECS entities with UI-specific components and layout state. Because the UI system is built on the same ECS foundation as the rest of the engine, you can combine UI with 3D, 2D, and scripting seamlessly.



UI scenes

For anything beyond a single HUD element, put UI into its own **UI scene** and add it as a scene layer on top of your gameplay scene.

```
-- Startup: run gameplay and HUD simultaneously
Engine.setScene(gameplayScene)
Engine.addSceneLayer(hudScene)
```

This approach keeps gameplay entities separate from screen-space widgets, makes it easy to swap menus without touching gameplay data, and lets you load/unload UI independently (for example, showing a pause menu without reloading the game level).

UI entities are laid out in their scene's camera space. In a 2D or UI scene that is the canvas, but in a 3D scene it is the world, so a Text or Image created there is placed among the models rather than on the screen. A 3D game's HUD therefore goes in a UI scene, even a single element, added as a layer or as a child scene (p. 184) of the level.

Typical UI scene examples:

UI Scene	Content
HUD	Health bar, ammo counter, minimap overlay
Main menu	Title, Play/Settings/Quit buttons
Pause menu	Resume, Options, Quit to menu
Settings screen	Sliders, dropdowns, back button
Dialog / modal	Confirmation popup, item description panel

UI Scene	Content
Loading screen	Progress bar and spinner

UI widget classes

Class	Purpose
`UILayout` (p. 537)	Base for all UI elements — anchors, size, and margin
`Image` (p. 513)	Textured or solid-color rectangular region, 9-patch scaling
`Text` (p. 528)	Unicode text with font, size, color, and multiline support; shapes complex scripts and right-to-left
`Button` (p. 506)	Interactive button with normal/over/pressed state textures and label
`Panel` (p. 518)	Framed or colored container with optional 9-patch background
`TextEdit` (p. 533)	Single-line text input with placeholder and password mode
`Scrollbar` (p. 525)	Horizontal or vertical scroll control with step and onChange callback
`Progressbar` (p. 523)	Horizontal or vertical fill display
`Container` (p. 510)	Automatic layout for vertical, horizontal, or wrapped children
`Polygon` (p. 521)	Simple polygon drawing, usable in 2D and UI scenes

Text and fonts

Set a font with `Text.font` (p. 529), pointing at a `.ttf`, `.otf` or `.ttc` file in the project. Leaving it unset uses the fonts built into the engine: a Roboto subset covering Latin, Cyrillic, Greek and Vietnamese, and an Arabic subset of Noto Sans Arabic.

Those two always sit at the end of the chain, so a codepoint the chosen font does not carry is drawn from them. An Arabic-only font still renders Latin digits and labels, and a Latin-only font still renders Arabic. A codepoint no font covers is drawn as the missing-glyph box.

For projects needing several custom faces at once, assign up to three additional fonts to slots `1` through `3`. They are tried in numeric order and empty slots are skipped:

```
label.font = "fonts/MyLatin.ttf"
label:setFont(1, "fonts/NotoKufiArabic.ttf")
label:setFont(2, "fonts/NotoSansCJK.ttc")
```

The `font` property is an alias for slot `0`, so replacing it preserves the fallback chain. C++ has the same `setFont(index, path)` and `getFont(index)` overloads. At direct component level, `TextComponent::font` is a fixed four-element `FontArray`: the main font followed by the three fallbacks. See the `Text.font` (p. 529) reference for all overloads and migration from the former `fontFallbacks` property.

In the editor, set the main face in the Text component's **Font** row. Use the stacked layers button to open **Additional fonts**, where **Fallback 1–3** can each be browsed, cleared, or filled by dragging a font from Resources. All slots are saved and exported as one ordered chain.

Complex scripts

Strings are shaped before being drawn, so scripts that need contextual glyph forms work with no extra setup. Arabic letters take their initial, medial, final or isolated shape from their neighbours, ligatures and kerning are applied, and right-to-left runs are reordered by the Unicode Bidirectional Algorithm. A label mixing Arabic and Latin lays each run out in its own direction.

Shaping is per font, so a run only joins within the font that covers it. If Arabic looks disconnected, the letters are most likely being drawn from a fallback that has no Arabic shaping rules — set an Arabic font explicitly and the joining returns.

> **Note**
> A drawn glyph does not map one to one onto a codepoint. Shaping can merge several codepoints into one ligature or expand one into a base plus marks, so `getNumChars` (p. 531) counts codepoints, not glyphs.

Anchors and layout

Every UI element inherits from `UILayout` which exposes an anchor system for resolution-independent placement. Anchors define how an element is positioned and sized relative to its parent.

Anchor presets

Use anchor presets for the most common layouts:

```
Image logo(&scene);
logo.setAnchorPreset(AnchorPreset::CENTER);
logo.setTexture("ui/logo.png");

Image header(&scene);
header.setAnchorPreset(AnchorPreset::TOP_WIDE);  // stretches full width at the top
```

Available presets include `TOP_LEFT`, `CENTER_TOP`, `TOP_RIGHT`, `CENTER_LEFT`, `CENTER`, `CENTER_RIGHT`, `BOTTOM_LEFT`, `CENTER_BOTTOM`, `BOTTOM_RIGHT`, `LEFT_WIDE`, `RIGHT_WIDE`, `TOP_WIDE`, `BOTTOM_WIDE`, and `FULL_LAYOUT` (stretch to fill parent).

The layout places an anchored element every frame, so to animate its position use a relative track (p. 304), which moves it by its offset from the anchors.

Manual anchors

For precise control, set the four anchor points directly. Anchor point values range from 0 (left/top of parent) to 1 (right/bottom of parent):

```
widget.anchorPointLeft   = 0.25
widget.anchorPointRight  = 0.75
widget.anchorPointTop    = 0.0
widget.anchorPointBottom = 0.5
```

Building UI in code

In Lua, `anchorPreset` and a widget's text/label are **properties**; parenting is done by calling `addChild` on the parent. Button click handlers live on the `ButtonComponent` (`onPress` / `onRelease`).

Lua

```
scene = Scene()

-- Background panel
panel = Panel(scene)
panel.anchorPreset = AnchorPreset.CENTER
panel:setSize(400, 300)
panel:setTexture("ui/panel_bg.png")

-- Label
title = Text(scene)
panel:addChild(title)
title.anchorPreset = AnchorPreset.CENTER_TOP
title.text = "Game Over"
title.fontSize = 36

-- Button
btn = Button(scene)
```

```lua
panel:addChild(btn)
btn.anchorPreset = AnchorPreset.CENTER_BOTTOM
btn.label = "Restart"

local btnComp = btn:getButtonComponent()
btnComp.onPress = function()
    SceneManager.loadScene("Game")
end

Engine.setScene(scene)
```

C++

```cpp
Scene uiScene;

Panel panel(&uiScene);
panel.setAnchorPreset(AnchorPreset::CENTER);
panel.setSize(400, 300);
panel.setTexture("ui/panel_bg.png");

Text title(&uiScene);
panel.addChild(&title);
title.setAnchorPreset(AnchorPreset::CENTER_TOP);
title.setText("Game Over");
title.setFontSize(36);

Button btn(&uiScene);
panel.addChild(&btn);
btn.setAnchorPreset(AnchorPreset::CENTER_BOTTOM);
btn.setLabel("Restart");

btn.getComponent<ButtonComponent>().onPress = []() {
    SceneManager::loadScene("Game");
};
```

UI events and input

UI widgets receive pointer and touch events automatically. The `Button` fires `onPress` and `onRelease`; `TextEdit` fires `onChange` when the text changes; `Scrollbar` fires `onChange` with the new scroll value.

A button shows its state with a texture and a tint colour per state, and it can also grow when hovered or pressed (`scaleHovered`, `scalePressed`). With a **Transition Time**, the colour and scale change over that time instead of at once, so menu hover effects need no script. See Button transitions (p. 507).

UI clicks don't also trigger gameplay input (such as shooting when the player taps a UI button), because `ignoreEventsHandledByUI` is on by default. Turn it off to let the gameplay handlers see those events too:

Lua

```lua
Engine.ignoreEventsHandledByUI = false   -- property in Lua
```

C++

```cpp
Engine::setIgnoreEventsHandledByUI(false);
```

Keyboard and gamepad navigation

Menus can work without a mouse once a script focuses one of their buttons, usually when the menu opens. The focus shows right away:

Lua

```lua
self.playButton:setFocus()
```

C++

```
playButton->setFocus();
```

Until then the arrow keys, Tab, Enter, Space, the d-pad, the left stick, and the gamepad A button stay with gameplay, so buttons on a HUD don't take them, even after a mouse click.

With a button focused, the arrow keys, the d-pad, and the left stick move the focus to the nearest visible button in that direction, and Enter, Space, or the gamepad A button presses it. Tab and Shift+Tab go through the buttons in reading order. Hidden and disabled buttons are skipped, and so are buttons clipped out of view by a parent image or panel. A held d-pad or stick repeats after a short delay, like a held key.

A focused button looks hovered, with its hovered colour, texture, and scale, so a menu that already has a hover effect needs nothing else. Pressing it fires `onPress`, and letting go fires `onRelease` and the `onClick` of its UIComponent, the same as a mouse click. Moving or clicking the mouse hides the focus until the next arrow key, Tab, d-pad, or stick press, which only shows it again. That press continues from a button clicked in the meantime, or from the same button after a click on anything else. Enter, Space, and the A button go to gameplay while the focus is hidden.

While the focused button is hidden, keys and gamepads go back to gameplay, so closing a menu gives the input back. When the menu shows again, or a menu scene that was removed is added back, the focus stays hidden until the next of those presses, so a press right after it opens doesn't click the button left focused last time. Focus a button when the menu opens to show it right away. If the focused button is disabled or clipped out of view while the menu stays open, the next of those presses moves the focus to the first button. A destroyed button takes the focus with it, so focus another one after rebuilding a menu's buttons. To let go of the focus, call `setFocus(NULL_ENTITY)` on the UISystem (p. 595).

When more than one UI scene is shown, the topmost scene with a focusable button gets the keys, so a pause menu over the HUD takes them first. Without a focused button there, they go to gameplay and not to a menu under it. Turn off **Focusable** on a button the focus shouldn't reach or that shouldn't cover a menu, like a HUD button in the same scene as a menu or in a scene over it. Otherwise clicking it while the menu has the focus moves the focus there, and it keeps the keys after the menu closes. A TextEdit being typed in keeps the arrow keys and Enter for its own text, even when another scene has buttons, and they come back to the focused button once it's done. The keys and buttons that the UI uses don't reach `onKeyDown`, `onKeyUp`, `onGamepadButtonDown`, and `onGamepadButtonUp` while `ignoreEventsHandledByUI` is on, and neither does `onGamepadAxisMove` while the left stick moves the focus. When the stick starts moving the focus, `onGamepadAxisMove` gets it back at rest, so gameplay doesn't keep the last tilt, and once no menu takes it anymore, gameplay gets it back as it is. `Input.isKeyPressed` and the other polling functions still see everything.

9-patch scaling

`Image` and `Panel` support 9-patch (nine-slice) scaling so borders remain sharp as the widget resizes. Set the border insets to define the corner and edge regions that should not be stretched:

```
Image frame(&scene);
frame.setTexture("ui/frame.png");
frame.setPatchMargin(12, 12, 12, 12);  // left, right, top, bottom pixel margins
frame.setSize(300, 200);
```

Containers

`Container` arranges its children automatically in a vertical stack, horizontal row, or wrapped grid. This is useful for inventory grids, option lists, and toolbars where the number of items changes at runtime.

```
list = Container(scene)
list.type = ContainerType.VERTICAL   -- type is a property; values: VERTICAL, HORIZONTAL,
VERTICAL_WRAP, HORIZONTAL_WRAP

for i = 1, 5 do
    local item = Button(scene)
    list:addChild(item)
    item.label = "Item " .. i
end
```

Canvas size and scaling

The UI canvas size defines the logical coordinate space. Set a fixed canvas size that matches your design resolution and choose a scaling mode:

Scaling mode	Behavior
`FITWIDTH`	Scale to fit the horizontal axis; may letterbox vertically
`FITHEIGHT`	Scale to fit the vertical axis; may letterbox horizontally
`LETTERBOX`	Fit inside the screen maintaining aspect ratio
`CROP`	Fill the screen, cropping edges
`STRETCH`	Stretch to fill (may distort)
`NATIVE`	Use actual screen pixel resolution

Canvas size and scaling are configured on the global `Engine`, not on the scene:

Lua

```
Engine.setCanvasSize(1920, 1080)
Engine.scalingMode = Scaling.LETTERBOX   -- scalingMode is a property in Lua
```

C++

```
Engine::setCanvasSize(1920, 1080);
Engine::setScalingMode(Scaling::LETTERBOX);
```

See Multiple Resolutions (p. 288) for the full scaling guide, including how to keep the UI crisp over a scene that uses a fixed render resolution.

Practical guidance

- Design against a fixed canvas size (e.g. 1920×1080 or 1280×720) for predictable layouts across screen sizes.
- Use anchor presets wherever possible; reserve manual anchor values for non-standard layouts.
- Separate HUD, pause menu, and modal dialogs into different root containers or separate UI scenes.
- Keep UI textures and fonts organized in a dedicated asset folder.
- Use `Container` for lists and grids so the layout adjusts automatically when items are added or removed.
- Test UI scaling on the target aspect ratios early in development.

Animation

Doriax supports two complementary animation approaches: **editor-authored keyframe animation** built in the Animation Timeline, and **runtime action playback** written in code. Both are updated each frame by `ActionSystem`. You can mix them freely — author the idle loop in the timeline and drive attacks and UI effects with runtime actions.

Animation families

Family	Best for	System
Actions	Scripted one-shot or looping motion — position, rotation, scale, color, alpha	`ActionSystem`, `TimedAction` subclasses
Sprite animation	2D frame cycling through an atlas	`SpriteAnimation`
Skeletal / keyframe	Imported GLTF character and node clips	`Animation`, `Model`
Keyframe tracks	Authored transform and property curves on the timeline	`TranslateTracks`, `RotateTracks`, `ScaleTracks`, `MorphTracks`
Morph targets	Blend shape deformation for facial animation	`MorphTracks`
Particles	Time-based particle playback	`Particles`

Runtime actions

Actions target an entity and update one property over time. They are lightweight, code-friendly, and do not require the timeline editor.

Available action types

Class	Property controlled
`PositionAction` (p. 580)	Entity position
`RotationAction` (p. 582)	Entity rotation
`ScaleAction` (p. 584)	Entity scale
`ColorAction` (p. 577)	Material or UI element color
`AlphaAction` (p. 572)	Opacity / alpha
`TimedAction` (p. 588)	Callback after a duration or on each loop
`SpriteAnimation` (p. 586)	Sprite atlas frame sequence
`Animation` (p. 573)	Skeletal or keyframe clip
`Particles` (p. 724)	Particle system playback

Basic action example

`setAction` takes the start value, end value, duration in seconds, and an optional loop flag. The easing curve is set separately with `setFunctionType`:

Lua

```lua
-- Move an entity from (0, 0, 0) to (100, 200, 0) over 1.5 seconds
move = PositionAction(scene)
move:setTarget(entity)
move:setAction(Vector3(0, 0, 0), Vector3(100, 200, 0), 1.5, false)
move:setFunctionType(EaseType.QUAD_OUT)
move:start()
```

C++

```
PositionAction move(&scene);
move.setTarget(entity);
move.setAction(Vector3(0, 0, 0), Vector3(100, 200, 0), 1.5f, false);
move.setFunctionType(EaseType::QUAD_OUT);
move.start();
```

Chaining actions with events

Every action's `ActionComponent` exposes `onStart`, `onPause`, `onStop`, and `onStep` events. Subscribe to `onStop` to start the next action when one finishes:

Lua

```
move:getActionComponent().onStop = function()
    -- start the next action when this one ends
    nextAction:start()
end
move:start()
```

C++

```
move.getComponent<ActionComponent>().onStop = []() {
    nextAction.start();
};
move.start();
```

Looping, ping-pong and spins

`loop` repeats a timed action, and `pingPong` plays it to the end and back, in twice its duration, so a floating bob is a single `PositionAction` with both set. A `RotationAction` interpolates between two rotations, which never turns more than half a turn; for a full spin use `setSpinAction`, which turns the start rotation by an angle of any size around an axis (**Spin**, **Axis** and **Angle** in the editor):

Lua

```
bob = PositionAction(scene)
bob:setTarget(coin)
bob:setAction(Vector3(0, 1, 0), Vector3(0, 1.4, 0), 0.8, true)
bob:setFunctionType(EaseType.SINE_IN_OUT)
bob.pingPong = true
bob:start()

spin = RotationAction(scene)
spin:setTarget(coin)
spin:setSpinAction(Quaternion(), Vector3(0, 1, 0), 360, 2.0, true)
spin:start()
```

C++

```
PositionAction bob(&scene);
bob.setTarget(&coin);
bob.setAction(Vector3(0, 1, 0), Vector3(0, 1.4f, 0), 0.8f, true);
bob.setFunctionType(EaseType::SINE_IN_OUT);
bob.setPingPong(true);
bob.start();

RotationAction spin(&scene);
spin.setTarget(&coin);
spin.setSpinAction(Quaternion(), Vector3(0, 1, 0), 360.0f, 2.0f, true);
spin.start();
```

Starting, restarting and phase

`start()` resumes a paused action and does nothing to a running one, so a `stop()` followed by `start()` in the same frame leaves an action running where it was. To play it again from the beginning, call `restart()`.

Copies of the same animation, such as the coins of a level, move in sync. Set an action's `startOffset` to start it some seconds in, or `randomStart` to start each copy at a random time of its duration (**Start offset** and **Random start** in the Properties window). Both apply when an action starts from stopped, not when it resumes from a pause, and an action inside an Animation keeps its own offset, so a random start on one track shifts only that track.

Easing curves

All `TimedAction` subclasses accept an `EaseType` (via `setFunctionType`) that shapes the interpolation curve. Names follow the `<CURVE>_<IN/OUT/IN_OUT>` pattern. Common choices:

EaseType	Best for
`LINEAR`	Constant-speed motion, debug
`STEP`	Held poses — stop-motion and stepped, pose-to-pose motion
`QUAD_IN_OUT`	UI transitions and camera moves
`BOUNCE_OUT`	Playful jumps and pop-in effects
`BACK_IN`	Anticipation before a jump
`ELASTIC_OUT`	Springing UI elements
`CUBIC_OUT`	Smooth deceleration into a stop

The full set covers `QUAD`, `CUBIC`, `QUART`, `QUINT`, `SINE`, `EXPO`, `CIRC`, `ELASTIC`, `BACK`, and `BOUNCE`, each with `_IN`, `_OUT`, and `_IN_OUT` variants, plus `STEP` (hold the start value and jump at the end) and `CUSTOM` for supplying your own curve function.

Sprite animation

`SpriteAnimation` cycles through a sequence of atlas frames by **frame index**. Frames must be registered on the sprite first — with `addFrame` in code or via the Sprite Slicer (p. 116). The interval is in **milliseconds**, and looping is the last argument of `setAnimation`.

Lua

```lua
sprite = Sprite(scene)
sprite:setTexture("characters/hero.png")
-- frames 0..7 previously registered with addFrame or the Sprite Slicer

anim = SpriteAnimation(scene)
anim:setTarget(sprite)
anim:setAnimation(0, 7, 80, true)  -- startFrame, endFrame, interval (ms), loop
anim:start()
```

C++

```cpp
Sprite hero(&scene);
hero.setTexture("characters/hero.png");

SpriteAnimation anim(&scene);
anim.setTarget(&hero);
anim.setAnimation(0, 7, 80, true);
anim.start();
```

For per-frame timing, pass explicit frame and time lists: `setAnimation({0, 1, 2, 5}, {100, 80, 80, 200}, true)`.

As a shortcut, `Sprite` can drive its own animation without a separate action object:

```lua
sprite:startAnimation(0, 7, 80, true)
-- ...
sprite:stopAnimation()
```

> **Lookup uses the sprite animation entity name**
>
> `SpriteAnimationComponent` has no separate name. Rename the action entity in the Structure panel (or call `setName`) and find it with `Scene:findEntity` when you need to look it up by name.

Skeletal animation from GLTF

A `Model` exposes the animation clips embedded in its GLTF file, addressable by name or index. How you obtain the `Model` depends on whether the model is **already in your scene** or is being **created at runtime** — these are two different things, and mixing them up is the most common animation mistake.

GLTF files that contain clips, or more than one skin, import the **full node tree** under the Model entity. Animation channels target those node entities — joints *and* transform-only helpers — so node animation authored in the DCC plays back, not only skeletal deformation. Models may use several skins; each skinned mesh keeps its own joint list and inverse-bind matrices, and its bone matrices live in a `SkinComponent` on the mesh entity. See 3D Graphics — GLTF node hierarchy (p. 249).

Playing a clip on a model already in the scene

This is the usual case: you imported the GLTF in the editor (so it appears as an entity in the Structure panel), and you want a script on that entity — or on a parent — to start its idle clip. **Do not reload the file.** The mesh, nodes, and clips are already loaded; the script only needs to *reference* the existing entity and start the clip.

The cleanest way to reach another entity is an entity reference property (p. 223): expose a `Model` property, then drag the model entity onto it in the Properties window.

Lua

```lua
-- player.lua — attached to a parent entity that has a child model
local player = {}

-- 'model' is filled in by dragging the child model onto the property
player.properties = {
    { name = "model", type = "Model" },
}

function player:init()
    if self.model then
        local idle = self.model:findAnimation("idle")
        idle.loop = true
        idle:start()
    end
end

return player
```

Don't have a property wired up? Look the child up by name instead. `Scene:findEntity` returns the first entity with that name (use the two-argument form to scope the search to children of a given parent), and the **two-argument** `Model(scene, entity)` constructor *wraps that existing entity* without creating or destroying anything:

Lua

```lua
function player:init()
    -- find the child named "model" under this script's entity
    local modelEntity = self.scene:findEntity("model", self.entity)
    if modelEntity ~= NULL_ENTITY then
        local model = Model(self.scene, modelEntity)
        model:getAnimation(0):start()   -- first clip, by index
    end
end
```

Creating a model at runtime

When you genuinely want to spawn a brand-new model from code (an enemy, a pickup), use the **single-argument** constructor and then `loadGLTF`. This *creates a new entity*.

Lua

```lua
-- store the handle so it lives as long as you need the entity
self.enemy = Model(self.scene)
self.enemy:loadGLTF("characters/hero.gltf")

local walk = self.enemy:findAnimation("Walk")
walk.loop = true
walk:start()
```

C++

```cpp
Model hero(&scene);
hero.loadGLTF("characters/hero.gltf");

Animation walk = hero.findAnimation("Walk");
walk.setLoop(true);
walk.start();
```

> **`Model(scene)` creates a new entity — it does not find an existing one**
>
> `Model(self.scene)` (one argument) **always creates a new entity** and that handle *owns* it. `Model(self.scene, entity)` (two arguments) **wraps an existing entity** and does not own it. So:
>
> - Never put `Model(...)` + `loadGLTF` inside `onUpdate` — that reloads the file and churns entities every frame. Load once in `init`.
> - To touch a model placed in the editor, reference it (property or `findEntity`) and use the two-argument constructor. Reloading the GLTF in code is **not** required.
> - When you do create a model at runtime, keep the handle in `self` (or another long-lived reference). A one-argument handle stored only in a local can have its entity destroyed when the local is garbage-collected.

GLTF can carry multiple named clips in a single file. You can play several clips simultaneously for layered animation blending.

> **Clip lookup uses the animation entity name**
>
> GLTF import initially names each animation entity from the authored clip name (often Mixamo-style, e.g. `mixamo.com`). `findAnimation` searches that entity name—the same name shown in the Structure panel and animation selector. If you rename the entity, use the new name in `findAnimation` and `playAnimation`. You can also address the clip by index with `getAnimation(0)`. Requesting a name or index that doesn't exist raises an error, so guard lookups you're unsure of.

Interpolation modes

Imported clips honor the GLTF sampler's `interpolation` setting, following the glTF 2.0 specification:

GLTF mode	Playback in Doriax
`LINEAR` (default)	Linear interpolation between keys; rotations use spherical linear interpolation (slerp)
`STEP`	Each key's value holds until the next key — imported as per-segment `STEP` easing
`CUBICSPLINE`	Cubic Hermite interpolation from the clip's per-key in/out tangents; interpolated rotations are normalized

`CUBICSPLINE` tangents are stored on the track components (`inTangents` / `outTangents`) and survive scene saves and code export. Editing an imported cubic track keeps the tangents aligned with its keys: a key added in the timeline or the Properties window gets zero tangents — its two neighboring segments flatten toward it

while the rest of the clip keeps its imported shape. If a track's tangents ever fall out of sync with its values, playback degrades safely to linear.

Smooth transitions (crossfading)

Starting a clip with `start()` snaps the skeleton straight into that clip's pose. For character motion you almost always want a **crossfade** instead — when the player stops running, the skeleton should ease from the run cycle into idle over a fraction of a second rather than pop. Doriax blends this for you: while a transition is in progress both clips play and each bone's pose is a weight-normalized average of them.

The simplest entry point is `Model:playAnimation` (p. 452), which fades out whatever is currently playing on the model and fades the requested clip in:

Lua

```lua
-- Hold looping clips; playAnimation manages the transition
self.model:findAnimation("Idle").loop = true
self.model:findAnimation("Run").loop = true
self.model:playAnimation("Idle")        -- uses each clip's Fade time

-- on input, later:
self.model:playAnimation("Run", 0.2)    -- crossfade over 0.2s
self.model:playAnimation("Die", 0.1)    -- quick blend into a one-shot
```

C++

```cpp
model.findAnimation("Idle").setLoop(true);
model.findAnimation("Run").setLoop(true);
model.playAnimation("Idle");

model.playAnimation("Run", 0.2f);
model.playAnimation("Die", 0.1f);
```

When the fade time is omitted, each clip's authored **Fade time** (`defaultFadeTime` (p. 574), set in the Properties window or in code) is used. Tune it per clip: ~0.2–0.3s for locomotion, shorter (~0.1s) for a snappy hit or death reaction. `playAnimation` works on clips of a single model; to crossfade arbitrary clips by hand, ramp their weights directly with `Animation:fadeIn` (p. 574) and `fadeOut`, or set `blendWeight` (p. 574) yourself for layered blending.

You can preview a transition without running the game — see Blending clips in the editor (p. 132).

Keyframe tracks (timeline-authored)

In the editor, the Animation Timeline writes keyframe data into component tracks. These are stored in the scene file and played back at runtime through the action system. The track types available are:

Track class	Property animated
`TranslateTracks`	Entity position over time
`RotateTracks`	Entity rotation over time
`ScaleTracks`	Entity scale over time
`MorphTracks`	Morph target weights for shape blending

Create track entities from the **Structure panel** (**Create entity → Animation → Translate/Rotate/Scale/Morph Tracks**) or from the Animation Timeline's **+ Add Action** menu. A track entity holds a shared key-time list plus one value per key, and animates the entity set as its action target. In the Animation Timeline, right-click empty track space to create and key transform tracks on demand, then use the camera **Snapshot** button to store the complete pose across existing tracks as one undo step. See Keying transforms (p. 131). `TranslateTracks` paths can also be edited visually in the scene view — see Editing movement paths (p. 87).

An action or animation with **Play on start** checked (ActionComponent) starts with the scene, so idle motion such as a spinning pickup needs no script. **Loop** on an AnimationComponent repeats the animation, and

Loop on a track's KeyframeTracks component repeats the track when it plays on its own, without an animation.

Relative tracks

Track values are the target's local position, rotation and scale. With **Relative** checked on the KeyframeTracks component (`setRelative(true)`), they are offsets from the pose the target has when the track starts instead: the position adds to it, the rotation is applied after it and the scale multiplies it. The same track then works wherever the target is placed, so one bob or wobble serves every copy of a bundle. A relative track that does not end where it began moves the target a bit further each time it is started again; a looping animation repeats it from the same starting pose.

A UI element with anchors is placed by its layout every frame, so an absolute track cannot move it; a relative track moves it through its layout offset, and the anchors keep working. Keying a relative track in the Animation Timeline stores the target's offset from where it was when the preview started.

Per-segment easing

Each segment between two consecutive keys can have its own easing curve: segment `i` shapes the interpolation from key `i` to key `i+1`, using any `EaseType` (p. 588). Unset segments are linear. `STEP` holds key `i`'s value for the whole segment — useful for stepped, pose-to-pose motion. GLTF-imported clips arrive with easing matching their sampler: `LINEAR` clips leave the list empty, `STEP` clips fill every segment with `STEP`, and `CUBICSPLINE` clips carry per-key tangents instead (see Interpolation modes). In the **Properties** window, **KeyframeTracks → Easing** is a collapsible, sparse list: **Add Ease** appends the next explicit segment entry, up to the number of key pairs, and the trash button removes an entry and shifts later entries to preceding segments. Missing trailing entries remain linear. You can also edit easing in code:

Lua

```lua
local path = TranslateTracks(scene)
path:setTarget(box.entity)
path:setTimes({0.0, 1.0, 2.5, 3.0})
path:setValues({Vector3(0,0,0), Vector3(5,0,0), Vector3(5,0,5), Vector3(0,0,0)})
path:setEasing(0, EaseType.QUAD_IN_OUT) -- key 0 -> key 1
path:setEasing(2, EaseType.BOUNCE_OUT)  -- key 2 -> key 3
path:start()
```

C++

```cpp
TranslateTracks path(&scene);
path.setTarget(box.getEntity());
path.setTimes({0.0f, 1.0f, 2.5f, 3.0f});
path.setValues({Vector3(0,0,0), Vector3(5,0,0), Vector3(5,0,5), Vector3(0,0,0)});
path.setEasing(0, EaseType::QUAD_IN_OUT); // key 0 -> key 1
path.setEasing(2, EaseType::BOUNCE_OUT);  // key 2 -> key 3
path.start();
```

Morph targets

Morph targets (blend shapes) are authored in a 3D tool such as Blender and exported through GLTF. Use `MorphTracks` on a Model entity to blend between shape keys at runtime.

```cpp
model.setMorphWeight("smile", 0.8f);
model.setMorphWeight("blink", 1.0f);
```

Best practices

- Name animation clips consistently (`idle`, `walk`, `run`, `jump`, `attack`).
- Keep looping clips separate from one-shot clips.

- Use runtime actions for UI and gameplay feedback; use the timeline for complex authored motion.
- Leave dynamic bodies to the simulation. A kinematic body (a moving platform) can be animated: it follows the animation with a velocity, so it carries and pushes dynamic bodies (see Moving a body (p. 320)).
- Test frame intervals in play mode; what looks good in the editor may feel wrong at the actual game frame rate.

See also

- Action (p. 569) — base action class
- TimedAction (p. 588) — duration-based base
- SpriteAnimation (p. 586)
- SpriteAnimationComponent (p. 690)
- Animation (p. 573)
- Particles (p. 724)
- Animation Timeline (p. 129) — editor authoring

Particles

Doriax includes a CPU-simulated particle system for effects such as fire, smoke, sparks, rain, explosions, and magic. A particle system has two parts:

Part	Role
`Particles` action	Drives the simulation — emission rate, lifetime, initializers, modifiers, bursts. Configuration lives in `ParticlesComponent`.
Render target entity	Displays the particles — a `Points` entity (textured point sprites) or a `Mesh` with instancing (each particle is a mesh instance).

`Particles` derives from `Action` (p. 569): create it, point it at the render target with `setTarget`, configure, and `start()`.

Creating particles in the editor

In the **Structure panel**, right-click a render-target entity (such as a Points entity) and choose **Create → Particles**. The editor creates an entity with `ActionComponent` + `ParticlesComponent` and wires its action target to the parent. Tune emission settings in the **Properties window**.

Creating particles in code

Lua

```lua
-- 1. A Points entity renders the particles as textured point sprites
points = Points(scene)
points:setTexture("effects/spark.png")

-- 2. The Particles action simulates and emits into the target
particles = Particles(scene)
particles:setTarget(points)

particles.rate = 50          -- particles emitted per second (property in Lua)
particles.loop = true

particles:setLifeInitializer(1.0, 2.0)                    -- min, max seconds
particles:setSpherePositionInitializer(0.5)               -- emit within a sphere
particles:setVelocityInitializer(Vector3(-1, 2, -1), Vector3(1, 4, 1))
particles:setAccelerationInitializer(Vector3(0, -9.8, 0))   -- gravity
particles:setSizeInitializer(10, 30)

-- Fade out between 1s and 2s of each particle's life
particles:setAlphaModifier(1.0, 2.0, 1.0, 0.0)

particles:start()
```

C++

```
Points points(&scene);
points.setTexture("effects/spark.png");

Particles particles(&scene);
particles.setTarget(&points);

particles.setRate(50);
particles.setLoop(true);

particles.setLifeInitializer(1.0f, 2.0f);
particles.setSpherePositionInitializer(0.5f);
particles.setVelocityInitializer(Vector3(-1, 2, -1), Vector3(1, 4, 1));
particles.setAccelerationInitializer(Vector3(0, -9.8f, 0));
particles.setSizeInitializer(10.0f, 30.0f);
particles.setAlphaModifier(1.0f, 2.0f, 1.0f, 0.0f);

particles.start();
```

Lua properties vs. C++ setters

Emitter settings (`rate`, `loop`, `emitter`, `localSpace`, `maxPerUpdate`, `emitterShape`) are **properties** in Lua and `set…()` methods in C++. The initializer/modifier calls use the same `set…` form in both languages.

Initializers and modifiers

Every particle property follows the same pattern:

- An **initializer** sets the value when the particle spawns. Passing two values picks a random value in that range per particle.
- A **modifier** interpolates the value between two points of the particle's lifetime (`fromTime` and `toTime`, in **seconds** of the particle's life), optionally shaped by an `EaseType` (p. 300).

Property	Initializer	Modifier
Lifetime	`setLifeInitializer(min, max)`	—
Position	`setPositionInitializer(...)` + shape helpers	`setPositionModifier(...)`
Velocity	`setVelocityInitializer(...)`	`setVelocityModifier(...)`
Acceleration	`setAccelerationInitializer(...)`	`setAccelerationModifier(...)`
Color	`setColorInitializer(...)`	`setColorModifier(...)` or color gradient
Alpha	`setAlphaInitializer(...)`	`setAlphaModifier(...)`
Size	`setSizeInitializer(...)`	`setSizeModifier(...)`
Rotation	`setRotationInitializer(...)`	`setRotationModifier(...)`
Scale	`setScaleInitializer(...)`	`setScaleModifier(...)` (instanced mesh targets)
Sprite frame	`setSpriteIntializer(...)`	`setSpriteModifier(...)`

Emitter shapes

Position initializers can emit from a volume or surface:

Shape	Helper
Box (min/max corners)	`setPositionInitializer(minPosition, maxPosition)`
Sphere	`setSpherePositionInitializer(radius [, innerRadius])`
Hemisphere	`setHemispherePositionInitializer(radius [, innerRadius])`
Circle	`setCirclePositionInitializer(radius [, innerRadius])`
Cone	`setConePositionInitializer(angle, height)`

Bursts

Bursts emit a batch of particles at a specific time, on top of (or instead of) the continuous `rate`:

```
particles:addBurst(0.0, 100)        -- 100 particles immediately
particles:addBurst(1.5, 20, 40)     -- 20–40 particles at 1.5 s
```

For a one-shot explosion, set `rate` to `0`, add a burst at time `0`, and disable looping.

Color gradients

Instead of a single color modifier, define a gradient over the particle's normalized lifetime (0 = spawn, 1 = death):

```
particles:addColorGradientStop(0.0, Vector3(1.0, 0.9, 0.3))  -- bright yellow at spawn
particles:addColorGradientStop(0.5, Vector3(1.0, 0.3, 0.1))  -- orange mid-life
particles:addColorGradientStop(1.0, Vector3(0.2, 0.2, 0.2))  -- dark smoke at death
```

Animated particle sprites

When the target texture is a sprite sheet (sliced with the Sprite Slicer (p. 116)), particles can spawn with random frames and animate through frames over their lifetime:

```
particles:setSpriteIntializer(0, 3)              -- random start frame 0–3
particles:setSpriteModifier(0.0, 2.0, {0, 1, 2, 3})
```

Local vs. world space

By default particles simulate in **world space** — they stay behind when the emitter moves (right for smoke trails and exhaust). Set `localSpace = true` to make particles follow the emitter's transform (right for auras and shields attached to a character).

Performance tips

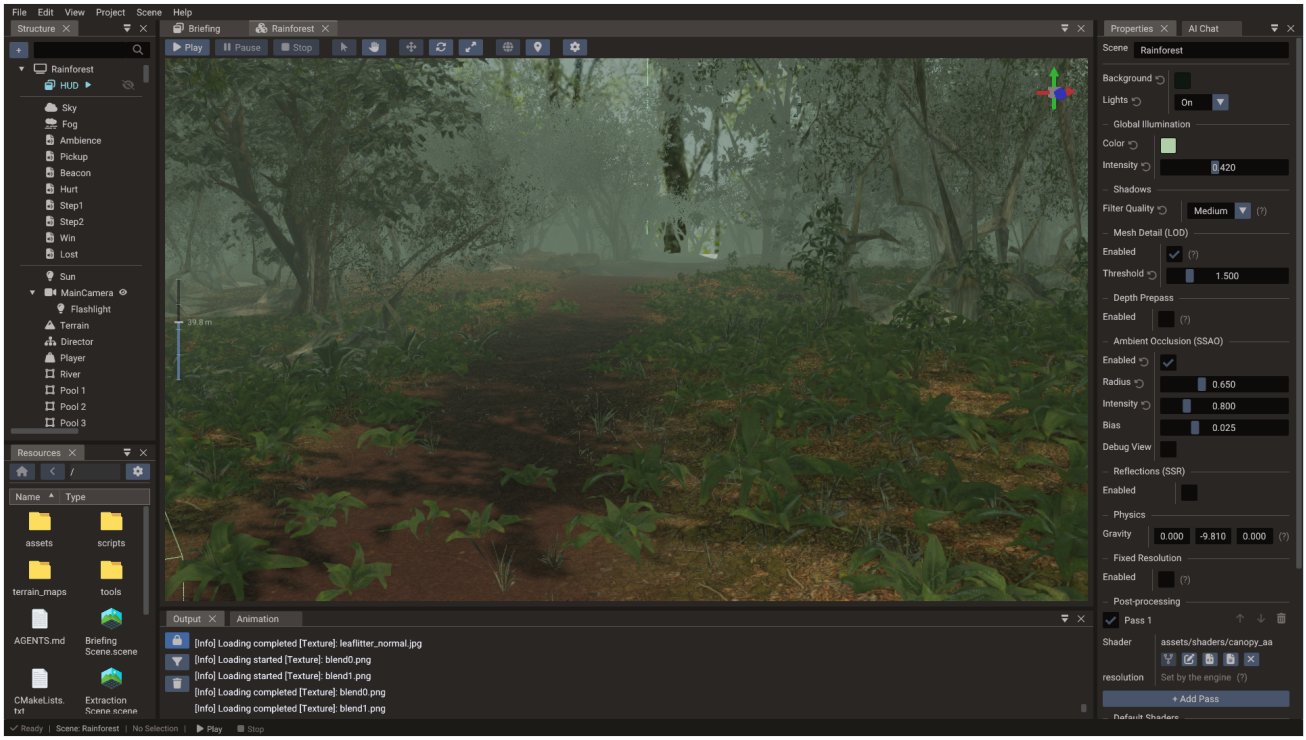
- Prefer `Points` targets for small, camera-facing particles — they are the cheapest to render. Use instanced-mesh targets only when particles need real geometry.
- Keep `maxParticles` as low as the effect allows; the simulation is CPU-side.
- Use `maxPerUpdate` to cap spawn spikes after frame hitches.
- Reuse one emitter with bursts rather than creating emitters per explosion.

See also

- Particles (p. 724) — full API reference
- Animation (p. 298) — the action system and easing curves
- 2D Graphics (p. 243), 3D Graphics (p. 248)

Terrain

Doriax renders large outdoor landscapes with a heightmap-based `Terrain` object. The terrain is a continuous-LOD quadtree, so the polygon count on screen stays roughly constant regardless of terrain size, with detail concentrated near the camera.



Terrain anatomy

Input	Purpose
Heightmap	Grayscale image that displaces the grid vertically (white = `maxHeight`)
Base texture	The ground texture tiled across the whole terrain
Blend maps	Up to three RGB masks; each channel weights one detail layer
Detail layers	Up to nine painted surfaces (grass, rock, path, ...) mixed over the base
Foliage layers	Meshes scattered over the surface from painted density maps

The heightmap and the blend maps are stretched once over the whole terrain, so they are sampled with clamp-to-edge wrapping: the outer row and column of texels define the border of the terrain instead of blending with the opposite edge. Only the base and detail textures tile.

Detail layers

Detail layers are numbered from 1 and grouped in threes: layers 1–3 are weighted by the RGB channels of the first blend map, 4–6 by the second, 7–9 by the third. Whatever weight the layers leave unclaimed goes to the base texture, so an unpainted terrain shows the base alone.

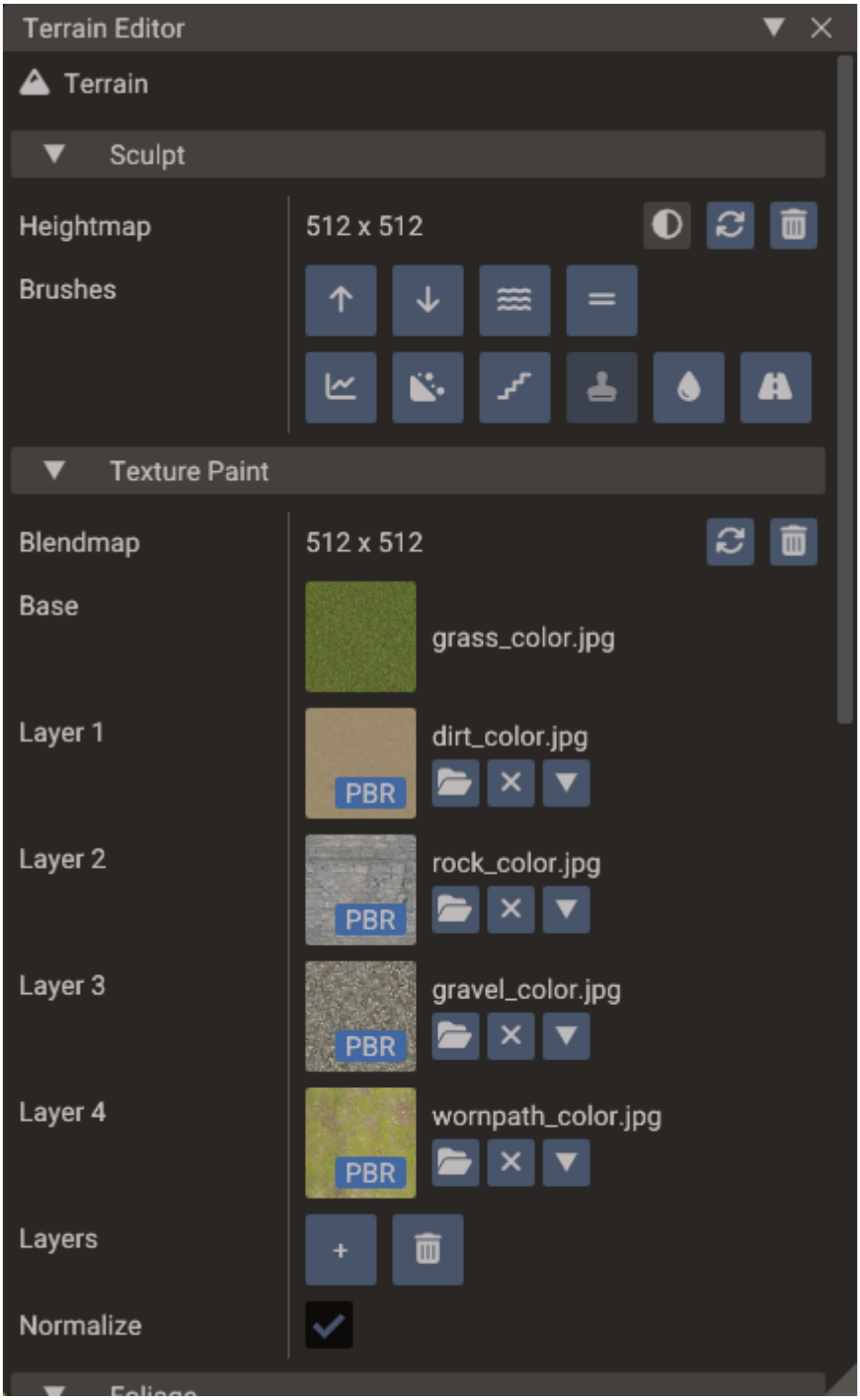
Every map of every layer is uploaded as a slice of one texture array, so the whole terrain costs a single sampler no matter how many layers and maps it uses. Sources are converted to RGBA and resized to a common size, capped at 2048 px — keep source maps at 1024 or below on a heavily layered terrain, or the array grows quickly.

Three things shape how the layers meet on screen:

- **Height blending.** A detail texture's alpha is read as *layer height*, not opacity. Where two layers overlap, the taller one takes the contact zone — gravel settles between cobbles instead of cross-fading into them. Fully opaque textures have no height and blend on their blend-map weight alone.
- **Triplanar projection on slopes.** A flat top-down projection stretches over a cliff, so as the surface turns vertical the layers fade into side projections taken along X and Z.
- **Distance tile break.** Past a few tiles of distance a second, coarser tiling rate fades in and mixes with the first, breaking up the repeating pattern of tiled ground.

PBR layers

A layer starts as a **color layer**: it blends its texture over the base and leaves every other surface property to the terrain material. Turning on **PBR** gives that layer its own surface instead.



Map	Read from	Without it
Color	RGB, decoded from sRGB and multiplied by the layer tint	White
Normal	Tangent-space RGB, projected with the layer	The terrain's own surface
Roughness	Green, or the only channel of a grayscale map	The roughness factor alone
Metallic	Blue, same fallback	The metallic factor alone
Occlusion	Red	Unoccluded
Height	Any channel; only biases blending, never geometry	The layer blends flat

Each map has a factor beside it, so a layer can be authored with factors alone and no maps at all. A packed ORM texture works by assigning the same file to Occlusion, Roughness and Metallic; separate grayscale files work just as well.

Every layer, PBR or not, also has a **UV scale and offset** that multiplies the shared detail tiling, so one layer can tile finer than its neighbours.

> **Color layers are left exactly as they were**
> A terrain with only plain color layers and default tiling renders through the same shader it always did — no added cost. The wider path switches on as soon as one layer turns on PBR or takes a custom tiling. In a mixed terrain, color layers keep blending their texture raw (and their alpha as height) rather than being re-interpreted, so adding a PBR layer never changes how the existing ones look.

Painted layers are written to the G-buffer as well, so screen-space reflections see the same normal, roughness, metallic and occlusion the lit surface shades with.

Foliage

A terrain can carry foliage layers: a mesh, a painted density map, and the rules for scattering it. Instances are not stored in the scene — they are resolved from the density map at runtime, in fixed cells batched into per-chunk instanced entities around the camera, and re-resolved when the camera moves. Each layer filters by ground slope and height, jitters scale and yaw, can lean instances along the surface normal, and has its own draw distance, with instances scaling in over the last quarter of it instead of popping. A **min draw distance** skips the instances nearer than it, so a copy of a layer with a simpler mesh can take over where the detailed one ends. **Cast shadows** (default on) puts those instances into the shadow cascades; turn it off on grass-scale scatter so the foliage still draws in the colour pass but skips the extra shadow geometry. **Wind sway** bends the instances toward the scene wind (p. 275), the tips by that much in 10 m/s, the base still.

Foliage entities belong to the engine, not to the authored scene: they are not listed in the editor's Structure panel and clicking one selects its terrain.

For props that need to be real entities — collision, scripts, animation, or just a hand-placed look — use the Terrain Editor's object placement brush instead, which parents ordinary models or bundles to the terrain.

Editing terrain in the editor

Select an entity with a Terrain component and click **Open Terrain Editor** in the Properties window. The window paints every map the terrain uses:

Section	Tools
Sculpt	Raise, Lower, Smooth, Flatten, Sharpen, Noise, Terrace, Stamp, Erode, Ramp
Texture Paint	Paint the base or any detail layer into the blend maps, optionally masked by slope and height
Foliage	Add layers and paint their density maps
Objects	Scatter models or bundles as children of the terrain, instanced or as individual entities

Brushes have configurable size, strength, shape (circle/square), falloff, and an optional grayscale mask. See Terrain Editor (p. 123) for the full window reference.

Creating terrain in code

Lua

```lua
terrain = Terrain(scene)
terrain:setHeightMap("terrain/heightmap.png")
terrain:setTexture("terrain/grass_base.png")

-- Layers 0-2 are weighted by the RGB channels of blend map 0
terrain:setBlendMap("terrain/blendmap.png")
terrain:setTextureLayer(0, "terrain/rock.png")
terrain:setTextureLayer(1, "terrain/grass_detail.png")
```

```lua
terrain:setTextureLayer(2, "terrain/path.png")

-- Layers 3-5 need a second blend map
terrain:setBlendMapIndex(1, "terrain/blendmap1.png")
terrain:setTextureLayer(3, "terrain/snow.png")

-- A layer with its own PBR surface
local mud = TerrainSurfaceLayer()
mud.pbr = true
mud.colorTexture = Texture("terrain/mud_color.png")
mud.normalTexture = Texture("terrain/mud_normal.png")
mud.roughnessFactor = 0.3
mud.uvScale = Vector2(2, 2)
terrain:setSurfaceLayer(4, mud)

-- Dimensions and LOD are properties in Lua
terrain.size = 2000          -- world units per side
terrain.maxHeight = 80       -- world height of a white heightmap pixel
terrain.resolution = 32      -- grid segments per LOD node (multiple of 4)
```

C++

```cpp
Terrain terrain(&scene);
terrain.setHeightMap("terrain/heightmap.png");
terrain.setTexture("terrain/grass_base.png");

// Layers 0-2 are weighted by the RGB channels of blend map 0
terrain.setBlendMap("terrain/blendmap.png");
terrain.setTextureLayer(0, "terrain/rock.png");
terrain.setTextureLayer(1, "terrain/grass_detail.png");
terrain.setTextureLayer(2, "terrain/path.png");

// Layers 3-5 need a second blend map
terrain.setBlendMap(1, "terrain/blendmap1.png");
terrain.setTextureLayer(3, "terrain/snow.png");

// A layer with its own PBR surface
TerrainSurfaceLayer mud;
mud.pbr = true;
mud.colorTexture = Texture("terrain/mud_color.png");
mud.normalTexture = Texture("terrain/mud_normal.png");
mud.roughnessFactor = 0.3f;
mud.uvScale = Vector2(2, 2);
terrain.setSurfaceLayer(4, mud);

terrain.setSize(2000.0f);
terrain.setMaxHeight(80.0f);
terrain.setResolution(32);
```

`Terrain` derives from `Mesh`, so material, shadow, and texture APIs from `Mesh` (p. 436) apply as well — `setTexture` sets the base ground texture through the material.

`setTextureLayer` sets only a layer's color, so it still does what it always did and leaves the rest of a PBR layer's surface alone. `setTextureDetailRed` / `Green` / `Blue` are layers 0, 1 and 2 under the names they had when a terrain had a single blend map. `setLayerFromMaterial` fills a layer from a Material (p. 734) in one call. Foliage layers have no runtime setters — they are authored in the editor and read back from the component. `setFoliageDistanceScale` scales the draw distances of all of them at once, for a quality setting.

LOD tuning

The terrain is a quadtree: `rootGridSize` × `rootGridSize` root nodes cover the whole terrain, and each node subdivides into four until `levels` is reached. Every node is drawn with the same grid of `resolution` × `resolution` segments, so a deeper node covers less ground with the same number of triangles — that is where the added detail comes from.

Property	Controls
`resolution`	Grid segments per side of every node — higher = denser geometry everywhere
`rootGridSize`	Root nodes per side of the terrain — the coarsest node is `size / rootGridSize`
`levels`	Quadtree depth — each level halves the node size, so the leaf node is `size / (rootGridSize × 2^(levels-1))`
`textureBaseTiles`	How many times the base texture repeats across the terrain
`textureDetailTiles`	Repeat count for the detail textures

Start with the defaults; raise `levels` when the ground near the camera looks too coarse, and `resolution` when it still does at the highest level. Both are paid for in nodes and triangles: the quadtree materializes `rootGridSize² × (4^levels - 1) / 3` nodes, so `levels` grows the node count exponentially and the engine refuses to build a terrain past its node budget of two million nodes (it logs an error asking for a lower **Levels** or **Root Grid Size**). `rootGridSize` also goes no higher than 4. Properties keeps **Root Grid Size** between 1 and 4 and **Levels** between 1 and 9, which always fit.

> **Resolution is rounded to a multiple of 4**
>
> Nodes stitch to their coarser neighbours by morphing only their odd vertices, and the internal half-resolution grid (`resolution / 2`) has to keep the same parity. The engine rounds `resolution` to the nearest multiple of 4 (minimum 4), writes the rounded value back to the component, and logs a warning. Set it in steps of 4 — the editor's **Resolution** field already drags that way — or the value you typed will not be the value you get.

LOD ranges

By default (`autoSetRanges`) the engine derives the distance ranges from the node sizes themselves: the first range is twice the leaf node size and each following level doubles it, so every level shows roughly the same amount of detail on screen no matter how large the terrain is. Only the coarsest range is stretched to the camera's far clip, so a terrain farther away than its own size still renders.

Terrain collision

For physics, add a `Body3D` with a **height field shape** to the terrain entity — see Physics (p. 315). Keep the visual resolution and collision resolution consistent so objects rest on the visible surface.

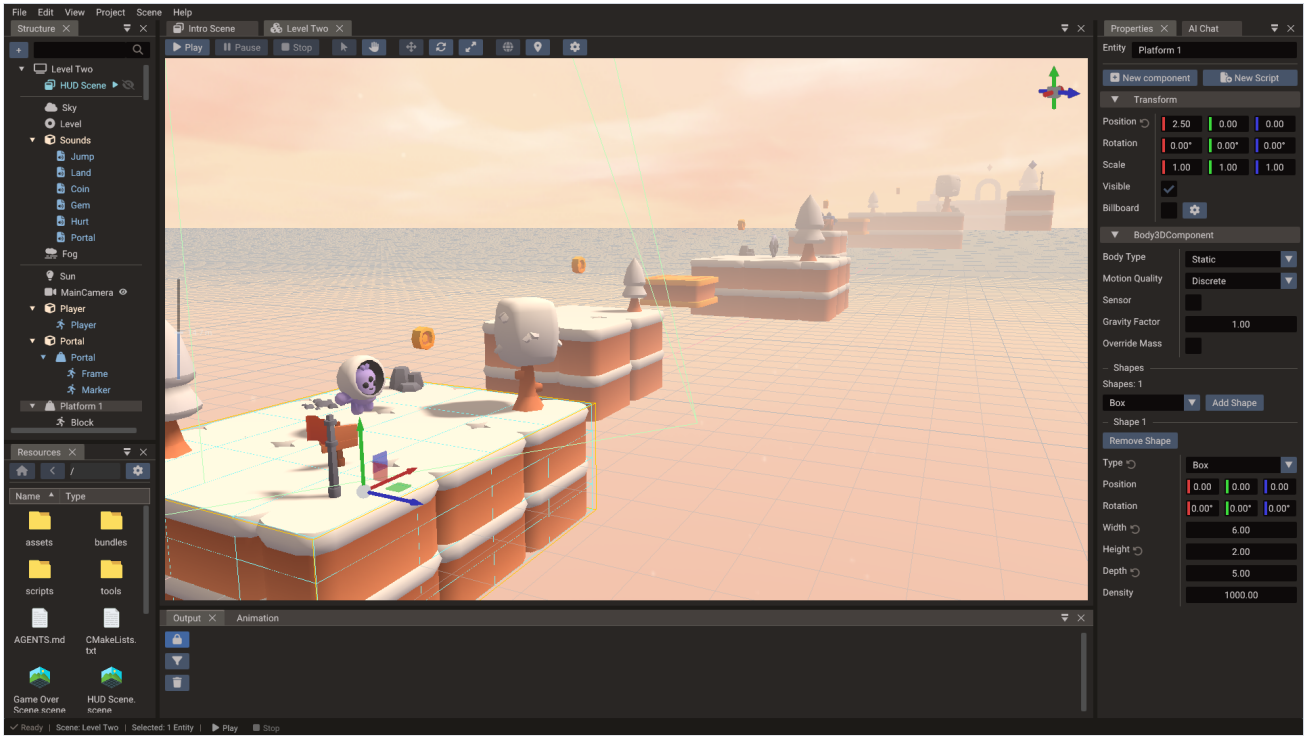
Samples Size sets how many heights the collision takes across the terrain, and **Bits Per Sample** how precisely it stores each one (8 by default, up to 16). Each block of samples keeps its own height range, so 8 bits are fine on gentle ground; raise the bits for steep or detailed terrain, at some memory cost.

See also

- Terrain Editor (p. 123) — sculpting, painting, foliage and object placement
- Terrain (p. 486) — full API reference
- 3D Graphics (p. 248) — lighting, materials, sky, and fog
- Physics (p. 315) — height field collision

Physics

Doriax includes integrated physics for both 2D and 3D games, so you can add realistic movement, collisions, and interactions without external libraries.



Physics backends

Dimension	Backend
2D	Box2D[11]
3D	Jolt Physics[12]

Both backends are integrated into the engine and exposed through the same ECS-based workflow. Each one can be left out of exported games that do not use it — see Project Settings → Physics backends (p. 59). The editor and Play always have both.

Core concepts

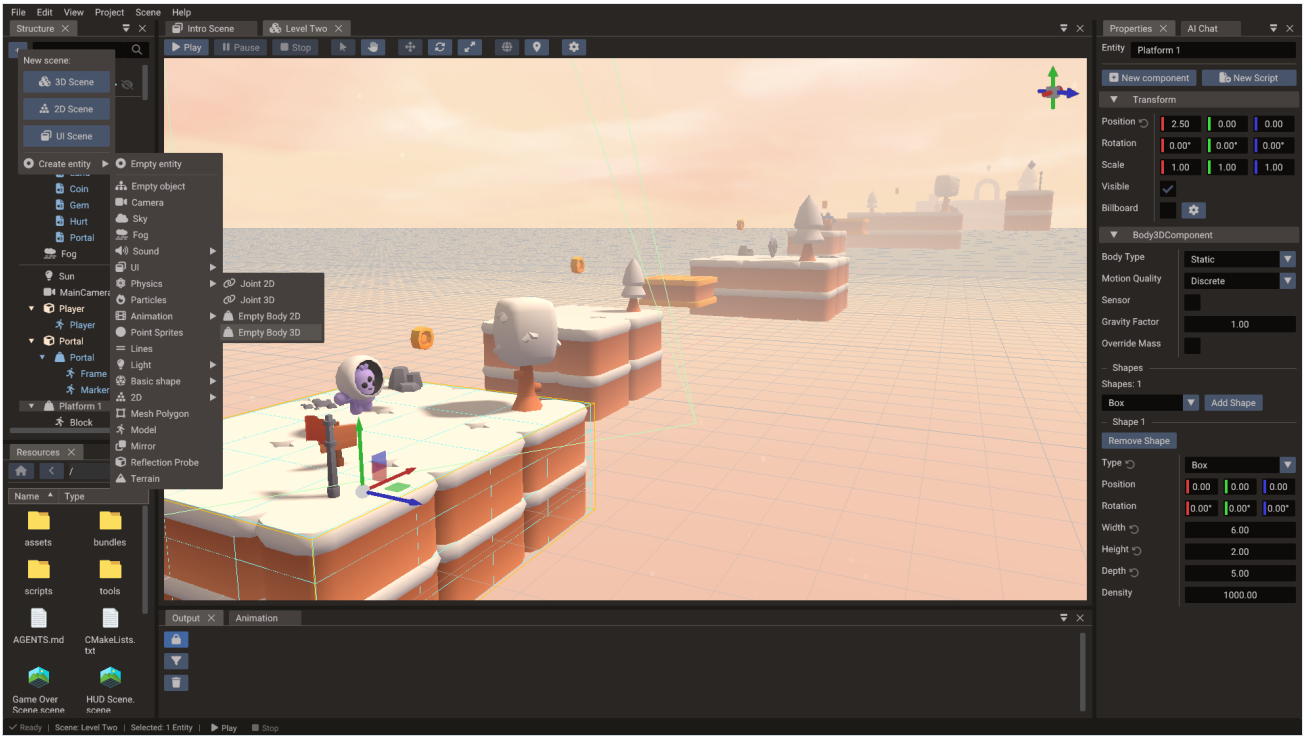
- **Rigid bodies** — give entities physical behavior so they respond to forces and gravity. Bodies can be static, kinematic, or dynamic.
- **Colliders / shapes** — define the volume used for collision detection (boxes, spheres, capsules, polygons, and more).
- **Joints** — constrain bodies together to model hinges, sliders, and other mechanical connections.
- **Collision detection** — the physics system detects overlaps and contacts between bodies each step.

Typical workflow

1. Add a physics body component to an entity (**New component → 2D Physics Body** or **3D Physics Body**), or create an empty body or a joint from **Create entity → Physics** in the Structure panel.

11. https://box2d.org/
12. https://github.com/jrouwe/JoltPhysics

2. Attach one or more collision shapes that match its geometry.
3. Configure mass, friction, restitution, and body type.
4. Let the physics system step the simulation each frame, updating transforms.



You can react to collisions in your game logic to trigger gameplay events such as damage, pickups, or sounds.

Gravity

Each scene has its own gravity, because each scene has its own physics worlds (see Physics across scenes). The 2D and 3D worlds are independent: `gravity2D` drives the Box2D world and `gravity3D` drives the Jolt world. Both default to `(0, -9.81)` m/s².

In the editor, select the scene in the Properties window and set **Gravity** in the **Physics** section — 2D scenes edit the 2D world, 3D scenes the 3D world. The value is saved with the scene and applied in exported projects.

C++

```
scene.setGravity2D(Vector2(0, -20));       // snappier platformer fall
scene.setGravity3D(Vector3(0, -3.7f, 0));  // Mars
```

Lua

```
scene.gravity2D = Vector2(0, -20)
scene.gravity3D = Vector3(0, -3.7, 0)
```

Scale the response per body with Body2D — gravityScale (p. 545) or Body3D — gravityFactor (p. 554). Changing gravity at runtime does not wake sleeping bodies — they pick up the new value when something wakes them.

2D physics

2D physics uses Box2D. A body can contain several shapes and each shape can have density, friction, restitution, rolling resistance, a tangent speed for conveyor belts, sensor state, and collision filtering. In

exported games a body holds at most `MAX_SHAPES_2D` shapes and a shape at most `MAX_SHAPE_POINTS_2D` points. They default to 4 and 8, and the export raises them when your scenes or bundles use more.

Shape	Use it for
Box/polygon	Platforms, crates, walls, characters with simple silhouettes
Circle	Balls, radial triggers, wheels
Capsule	Characters, rounded obstacles
Segment	Thin walls and edges that block from both sides
Chain	Terrain outlines and hollow containers; blocks from one side only

```
Body2D body = object.getBody2D();
body.createBoxShape(64, 32);
body.setType(BodyType::DYNAMIC);
body.setLinearVelocity(Vector2(200, 0)); // points per second
```

Chain shapes

A chain is a line of connected edges for level outlines: bodies slide along it without catching where edges meet. It collides on one side only, to the right of each edge going from one vertex to the next. Counter-clockwise vertices make a closed chain solid from the outside, like a platform. Clockwise vertices make it solid from the inside, like a ring that keeps a ball in. Rotating or mirroring the body doesn't change the side.

When a body is selected, ticks on its chain edges point to the colliding side, and **Collision Side → Reverse** in the shape's properties flips it. **Loop** closes the chain. An open chain doesn't collide on its first and last edges, so add one extra vertex at each end. Chains have no mass, so use them on static or kinematic bodies. See Body2D — createChainShape (p. 546) for code.

Tilemap collision

A Polygon shape can take its geometry from the Tilemap (p. 492) on the same entity. In the editor, set the shape's **Source** to **Entity Tilemap**; in code, call `createTilemapShape()`. Tiles that touch are joined into boxes, and tiles whose rect has **Collision** turned off are left out. However many boxes it makes, it counts as one shape toward `MAX_SHAPES_2D`.

The tiles are read when the body loads. After a script changes tiles, set `needUpdateShapes` on the Body2DComponent to rebuild the boxes.

C++

```
Body2D body = level.getBody2D(); // level is a Tilemap with its tiles placed
body.createTilemapShape();
```

Lua

```
local body = level:getBody2D() -- level is a Tilemap with its tiles placed
body:createTilemapShape()
```

See Tileset Slicer (p. 121) for the editor steps.

3D physics

3D physics uses Jolt Physics. Bodies can use primitives, compound shapes, mesh shapes, or height fields. Use simple primitives for dynamic bodies whenever possible, and reserve mesh shapes for static world geometry.

Shape	Use it for
Box/sphere/capsule/cylinder	Dynamic props and characters
Convex hull	Medium-complexity dynamic objects
Mesh	Static level collision

Shape	Use it for
Height field	Terrain collision

A mesh shape's **Build Quality** favors faster collision queries by default; **Favor Build Speed** makes a large mesh load sooner, at a cost on every query against it. A height field's **Bits Per Sample** (8 by default, up to 16) sets how precisely it stores each height: more bits follow steep terrain closer and use more memory. See Body3D — setShapeBuildQuality (p. 559) and setShapeBitsPerSample (p. 559).

```
Body3D body = object.getBody3D();
body.createCapsuleShape(0.8f, 0.25f);
body.setType(BodyType::DYNAMIC);
body.setAllowedDOFs2DPlane();
```

2D sensors

In 2D the flag lives on each **shape**, so one body can mix solid and trigger shapes (a character with a solid box and a foot sensor, a chest with a solid lid and a pickup zone). Tick **Sensor** on the shape in the Body2D component of the Properties window — the editor draws sensor shapes in orange — or set it from code with Body2D — setShapeSensor (p. 548). A sensor never pushes anything; its overlaps arrive through `beginSensorContact2D` / `endSensorContact2D` on the PhysicsSystem (p. 567), which name the sensor first and the visitor second. A sensor always reports its overlaps (its **sensor events** are implied), so give shapes a category bit so the callback can tell a coin from a spike:

C++

```
Body2D coin = coinSprite.getBody2D();
coin.createCircleShape(Vector2(0, 0), 22.0f);
coin.setShapeSensor(true);
coin.setCategoryBitsFilter(0x0010);

// on the hero
REGISTER_EVENT(scene->getSystem<PhysicsSystem>()->beginSensorContact2D, onSensor);

void Hero::onSensor(Body2D sensor, unsigned long shape, Body2D visitor, unsigned long) {
    if (visitor.getEntity() != getEntity()) return;
    if (sensor.getCategoryBitsFilter(shape) & 0x0010) collect(sensor.getEntity());
}
```

Lua

```
local coin = coinSprite:getBody2D()
coin:createCircleShape(Vector2(0, 0), 22)
coin.shapeSensor = true
coin:setCategoryBitsFilter(0x0010)
```

Two sensors never report each other, and ray casts ignore sensor shapes.

3D sensors

A whole 3D body can be turned into a trigger volume with the **Sensor** flag: it still reports contacts, but produces no collision response. Use it for checkpoints, pickup volumes, damage zones, and detection areas.

In the editor, tick **Sensor** in the Body3D component of the Properties window. The flag is stored on the component, so it is saved with the scene, applied in exported projects, and kept when the body is rebuilt (for example after changing shapes). From code, set it with Body3D — sensor (p. 555) before or after `load()`:

C++

```
Body3D trigger = checkpoint.getBody3D();
trigger.createBoxShape(2, 2, 2);
trigger.setIsSensor(true);
trigger.load();
```

Lua

```lua
local trigger = checkpoint:getBody3D()
trigger:createBoxShape(2, 2, 2)
trigger.sensor = true
trigger:load()
```

Sensor overlaps arrive through the regular 3D contact events below, and the Contact3D (p. 562) passed to them carries a `sensor` flag so trigger contacts can be told apart from solid ones. A static sensor only sees **awake** dynamic and kinematic bodies; make the sensor itself dynamic or kinematic when it must also detect sleeping bodies.

Buoyancy

A dynamic 3D body floats in a Water (p. 496) once its **buoyancy** is above `0`. Every fixed step, while the body's centre of mass is inside the water's area, the engine pushes it up by the part of its shape below the wave surface and tilts it with the waves, then drags it with the water. No script is needed.

In the editor, the Body3D component of a dynamic body has a **Buoyancy** section:

Field	Default	Purpose
Buoyancy	`0`	`1` is neutral, more floats and less sinks. `0` ignores the water.
Drag	`0.5`	How much the water slows the body down. Shown when Buoyancy is above `0`.
Angular Drag	`0.01`	How much the water slows the body rotation. Shown when Buoyancy is above `0`.

A box with a buoyancy of `2` floats about half under, `1.4` about 70% under, and `0.8` sinks slowly. The values are saved with the scene and applied in exported projects; from code they are Body3D — buoyancy (p. 554), `waterDrag`, and `waterAngularDrag`:

C++

```cpp
Body3D body = barrel.getBody3D();
body.createCylinderShape(0.5f, 0.4f);
body.setType(BodyType::DYNAMIC);
body.setBuoyancy(2.0f);
body.load();
```

Lua

```lua
local body = barrel:getBody3D()
body:createCylinderShape(0.5, 0.4)
body.type = BodyType.DYNAMIC
body.buoyancy = 2
body:load()
```

The water has no current: a floating body bobs and rocks with the waves but is not carried along by them. Bodies outside every water's area, and static or kinematic bodies, are not affected. Neither are bodies inside a Water Exclusion (p. 503), so cargo rests on the floor of a boat; the boat, which owns the volume, still floats. For an object without physics, read the surface with Water.getHeight (p. 502) instead.

To react to the water, such as a splash when a body enters it or a state when it sinks, ask how much of it is under with Body3D.getSubmergedVolume (p. 560). It takes a surface point and normal, so it also serves a body floated by your own forces.

The Water component has a **Buoyancy** section too:

Field	Default	Purpose
Buoyancy	On	Floats the bodies whose Buoyancy is above `0`. Turn it off for water that is only for show.
Depth	`0`	How far below the surface bodies still float. `0` has no bottom. Shown when Buoyancy is on.

Where waters overlap, a body floats in the one with the highest bottom that it is still above, so a pool with a **Depth** over a lake floats only the bodies inside it and the lake floats the rest. The bottom is not a floor: a body

that plunges below it sinks through, so give a pool a collider too. From code they are Water — buoyancy (p. 500) and `buoyancyDepth`.

Moving a body

A physics body owns its own pose. The engine syncs it with the entity's transform once per fixed step: **transform → body** before the step, **body → transform** after it. Which API you use depends on where your code runs.

Where	Continuous movement	Teleport
`onFixedUpdate`	`setLinearVelocity` / `applyForce`	`Body2D` / `Body3D` `setPosition`
`onUpdate`	not recommended for bodies	`Object::setPosition` also works

> **Transform positions and rotations written in onFixedUpdate are discarded**
>
> `Engine::onFixedUpdate` runs **between** those two syncs. A position or rotation written to the transform there is read back over by the post-step sync before anything renders it, so the write silently does nothing — no error, no warning, unless you also call `Object::updateTransform()` to refresh the world transform the pre-step sync reads. Moving a **parent** of a body-carrying entity has the same effect on the body. The body pose API below avoids the whole ordering question.
>
> The exact scope is worth knowing:
>
> - **3D bodies** — always. The post-step sync writes back every body each step.
> - **2D bodies** — only when Box2D reports the body as moved. A sleeping or static `Body2D` produces no move event, so a transform write there survives and reaches the body on the next sync. Do not rely on it: whether a body sleeps is the simulation's decision, not yours.
> - **Scale** — not affected. Neither path writes scale back, so `Object::setScale` persists. Only its effect is delayed: the collider is resized once the world scale refreshes in the next variable-timestep pass.

Use the body's own pose API instead. It writes to the simulation directly and updates the transform for you, so it works from any callback:

C++

```cpp
void Player::onFixedUpdate() {
    // Continuous movement: let the solver do the work
    body.setLinearVelocity(Vector3(input.x * speed, body.getLinearVelocity().y, input.z * speed));

    // Teleport: write the body pose, never the transform
    if (fellOffTheMap) {
        body.setPosition(spawnPoint);
        body.setLinearVelocity(Vector3::ZERO);
    }
}
```

Lua

```lua
function Player:onFixedUpdate()
    body.linearVelocity = Vector3(input.x * speed, body.linearVelocity.y, input.z * speed)

    if fellOffTheMap then
        body.position = spawnPoint
        body.linearVelocity = Vector3(0, 0, 0)
    end
end
```

Entities **without** a physics body have no such restriction — `Object::setPosition` works from `onFixedUpdate` as usual.

Prefer velocities and forces over teleporting whenever the motion is continuous. Teleporting skips collision detection between the old and new pose, so a body can pass straight through walls. Do not scale forces or torques by `Engine::getDeltatime()`: the solver already integrates them over the fixed step.

A **kinematic** body is the exception to teleporting: when its transform changes outside the fixed step — an animation (p. 303), a moving parent, or `Object::setPosition` in `onUpdate` — the body moves there with a velocity over the fixed steps of the frame. That velocity carries the bodies standing on a moving platform and pushes the ones in its way. To jump a kinematic body somewhere instead, use its `setPosition`.

Smooth motion between steps

The simulation advances in fixed steps, 60 per second by default, while frames come as fast as the display allows. So that a body does not jump once per step and stand still in between, its transform is **interpolated**: every frame it is drawn between the body's poses at the last two steps. The transform then trails the body by less than one step.

- Gameplay logic reads the body: `Body2D`/`Body3D` `getPosition` and `getRotation`.
- Whatever follows the body on screen, such as a camera, reads the transform (`Object::getWorldPosition`). Following the body pose brings the per-step jumps back.
- `setPosition` on the body or the object is a jump: the body is drawn there at once, not on its way there.
- A body moved by its transform, like an animated kinematic platform, is drawn where the transform puts it.

C++

```
// camera target, not Body3D(scene, entity).getPosition()
Vector3 focus = Object(scene, entity).getWorldPosition();
```

Lua

```
-- camera target, not body.position
local focus = Object(self.scene, self.entity):getWorldPosition()
```

To have the transform hold the body pose of the last step instead, turn it off with `PhysicsSystem::setInterpolation(false)` (`scene:getPhysicsSystem().interpolation = false` in Lua).

Contacts and filtering

The physics system exposes contact subscriptions so you can react to collisions in game logic. **2D bodies** use `beginContact2D`, `endContact2D`, hit, sensor, and `preSolve2D` events — use begin/end for gameplay state, hit for impacts, and pre-solve when a contact should be conditionally disabled. Shapes opt in to these: a pair reports begin/end, hit or pre-solve events only when one of its shapes has **Contact Events**, **Enable Hit Events** or **PreSolve Events** on, and all are off by default (see Body2D events (p. 548)). A 2D event whose body was destroyed or recreated before it was dispatched is dropped, so an end event does not come for a body that no longer exists. **3D bodies** instead use `onContactAdded3D`, `onContactPersisted3D`, and `onContactRemoved3D` (plus `onBodyActivated3D` / `onBodyDeactivated3D`); each added/persisted callback receives a Contact3D (p. 562) carrying the contact normal and points.

Collision filters use category and mask bits. Put broad gameplay groups into category bits, such as player, enemy, world, projectile, and trigger. Use masks to decide which groups interact.

What a 3D callback may do

3D contact events run *while Jolt is stepping*, unlike the 2D begin/end/sensor/hit events, which Box2D queues and the engine dispatches once the step is over (`preSolve2D` and `shouldCollide2D` are the 2D exceptions: they are filter hooks called mid-step). They run on Jolt's worker threads, but only one callback runs at a time, so callbacks can share script state without locks. Reading and moving bodies is fine there — velocities, positions, ray casts — but the broad phase is locked for the duration, so a body or joint cannot be **created**

from a callback: the call is refused and logged, and the next update retries it. Teardown is queued instead of refused, so removing a `Body3DComponent` from a callback still destroys its body, just after the step finishes.

`onBodyActivated3D` and `onBodyDeactivated3D` also run when a call outside the step wakes a body: `activate()`, a velocity or impulse on a sleeping body, or a joint motor. Jolt holds the body there too, so the callback can read and move bodies but must not wake a sleeping one.

`preSolve2D` and `shouldCollide2D` run mid-step with the Box2D world locked, so a 2D body can only be read there. Moving it, changing its settings, applying a force, casting a 2D ray or reading its contacts is refused and logged. A new body or joint is retried on the next update, shape settings apply after the step, and a removed body or joint goes when the step finishes.

Keep callbacks to recording what happened — a flag, a score, an entity to act on — and do the work from `onUpdate` or `onFixedUpdate`. `PhysicsSystem::isSteppingWorld2D()` and `isSteppingWorld3D()` (`physics.steppingWorld2D` and `physics.steppingWorld3D` in Lua) report the window for helpers called from both.

Raycasts and ground checks

A common need for character controllers is knowing whether a body is **standing on the ground**. Doriax has no built-in `isGrounded()` flag, but you can build a reliable check two ways.

Downward raycast (recommended)

Cast a short Ray (p. 649) straight down from the body's feet and test it against the 3D physics world with `RayFilter::BODY_3D`. Checking `normal.y` lets you reject steep walls so only near-horizontal surfaces count as ground.

```
bool isGrounded(Body3D& body, Scene* scene, float feetOffset, float probe = 0.15f) {
    // feetOffset = distance from the center of mass down to the soles
    // (for a capsule: halfHeight + radius)
    Vector3 com = body.getCenterOfMassPosition();
    Vector3 origin = com - Vector3(0, feetOffset - 0.05f, 0); // start just above the soles
    Ray ray(origin, Vector3(0, -(probe + 0.05f), 0));         // direction also sets the length

    // Pass the body so the ray continues through it instead of stopping at a self-hit
    RayReturn result = ray.intersects(scene, RayFilter::BODY_3D, body.getEntity());

    return result.hit
        && result.normal.y > 0.7f;           // ~45 degree slope limit
}
```

Pass `onlyStatic` or category/mask bits to restrict what counts as ground, for example `ray.intersects(scene, RayFilter::BODY_3D, true, groundCategory, groundMask, body.getEntity())`. The same `Ray` API is available in Lua (`RayFilter.BODY_2D` or `RayFilter.BODY_3D`). The returned RayReturn (p. 716) also carries the hit `distance`, which is handy for step snapping or coyote-time.

A game without a physics backend can still cast against mesh bounds with `RayFilter::BOUNDS`. It tests each mesh's axis-aligned box rather than a collider, so it suits picking and line-of-sight more than ground checks — see RayFilter (p. 651).

Contact normal (event-driven)

If you already subscribe to contact events, inspect the contact normal instead of raycasting. Read Contact3D (p. 562) `::getWorldSpaceNormal()` inside `onContactAdded3D` / `onContactPersisted3D`. Jolt's normal points **from body 1 toward body 2**, and Jolt — not your code — decides which body is which, so flip the sign when your character is body 1:

```
Vector3 n = contact.getWorldSpaceNormal();
Vector3 up = (bodyA.getEntity() == characterEntity) ? n * -1.0f : n;
bool grounded = up.y > 0.7f;
```

Track grounded state with a contact counter (increment on `onContactAdded3D`, decrement on `onContactRemoved3D`) rather than a single bool, so multiple simultaneous contacts are handled correctly.

Joints

Doriax exposes 2D and 3D joint wrappers for constrained motion. 2D joint types are distance, revolute, prismatic, mouse, wheel, weld, and motor joints. With **Rope** enabled, a distance joint lets the bodies move closer but never farther apart than when it was created. In 3D, use constraints and allowed degrees of freedom to lock or limit movement.

A 3D **hinge** or **prismatic** joint can also drive its bodies with a motor: at a speed, to a target angle or position, or both, with an optional force limit. Use it for doors, lifts, cranks, and wheels. In the editor it is the **Motor** section of the joint; from code see Joint3D — Motors (p. 564).

Both bodies of a joint must be in the joint's own scene.

Physics across scenes

Every scene owns its own Box2D and Jolt worlds. Only the worker threads and temporary memory are shared between them. When a scene stack (p. 184) runs several scenes at once, for example a gameplay scene with a HUD or a world made of chunk scenes, those worlds stay apart:

- A body in one scene never collides with a body in another scene, and never triggers its sensors.
- A ray cast against a `Scene*` tests only that scene's world. The returned `RayReturn::body` is an entity ID that only means something in that scene.
- Contact events report only the bodies of their own scene.
- A joint's `bodyA` and `bodyB` are looked up in the joint's scene. The editor accepts only same-scene entities in those fields.
- Only scenes on the engine are stepped. A scene removed with `SceneManager.removeChildScene` keeps its bodies, frozen, until it is added again.

There is no API that spans worlds. The approaches below go from simplest to most work.

Keep interacting bodies in one scene

This is the recommended setup. Put everything that has to touch physically in one scene, and use child scenes for the parts that don't take part in gameplay physics: HUD, menus, lighting and sky. A gameplay scene with a UI overlay needs nothing more.

Ray cast several scenes

Cast the same ray against each scene and keep the closest hit. Keep the scene next to the hit, because the entity ID needs it. `distance` is a fraction of the ray's length, so hits from different scenes compare directly.

C++

```cpp
struct SceneHit {
    Scene* scene = nullptr;
    RayReturn hit = Ray::NO_HIT;
};

SceneHit raycastScenes(const Ray& ray, RayFilter filter) {
    SceneHit best;
    for (Scene* scene : Engine::getScenesSnapshot()) {
        RayReturn hit = ray.intersects(scene, filter);
        if (hit.hit && (!best.scene || hit.distance < best.hit.distance)) {
            best.scene = scene;
            best.hit = hit;
        }
    }
    return best;
}
```

Lua

```lua
-- Lua has no getScenesSnapshot: pass the IDs of the scenes to test
local function raycastScenes(ray, filter, sceneIds)
    local bestScene, bestHit = nil, nil
    for _, id in ipairs(sceneIds) do
        local scene = SceneManager.getScenePtr(id)
        if scene then
            local hit = ray:intersects(scene, filter)
            if hit.hit and (bestHit == nil or hit.distance < bestHit.distance) then
                bestScene, bestHit = scene, hit
            end
        end
    end
    return bestScene, bestHit
end

local ids = { SceneManager.getSceneId("Level"), SceneManager.getSceneId("Props") }
local scene, hit = raycastScenes(ray, RayFilter.BODY_3D, ids)
```

This only works when the scenes share one coordinate space. An overlay drawn with its own camera needs its own ray, built with that scene's `Camera::screenToRay`. To test a single known body, `ray.intersects(body)` works for a body of any scene.

Collide across scenes with a proxy body

To make a body in scene A push bodies in scene B, give scene B a **kinematic** stand-in with the same shape, and move it to follow the real body.

Move the proxy from `onFixedUpdate`, which runs once per fixed step before any scene's world steps. Set its velocity toward the target rather than its position. Setting the position teleports it, so it won't push other bodies properly.

C++

```cpp
// player: Body3D in scene A, proxy: kinematic Body3D in scene B
Vector3 toTarget = player.getPosition() - proxy.getPosition();
proxy.setLinearVelocity(toTarget * (1.0f / Engine::getUpdateTime()));
```

Lua

```lua
local toTarget = self.player.position - self.proxy.position
self.proxy.linearVelocity = toTarget * (1 / Engine.updateTime)
```

This works one way only: the proxy pushes bodies in B, but nothing in B pushes the real body back. To get a response, handle the proxy's contacts in scene B (`onContactAdded3D`) and apply impulses to the real body in scene A yourself. If you need full two-way physics, put both bodies in the same scene.

Move a body to another scene

Doriax has no transfer API. Recreate the entity in the target scene instead:

1. Read the position, rotation, `linearVelocity` and `angularVelocity` of the old body.
2. Create the entity in the target scene, or instance a bundle there, and set those values on its new body.
3. Destroy the old entity.

Don't do this inside a contact callback, while `isSteppingWorld3D()` is `true` (see What a 3D callback may do). Note what needs to move in the callback, then move it in `onUpdate` or `onFixedUpdate`.

The new entity has a new ID, so update anything that refers to it. Contacts, sleep state and joints don't carry over.

Practical guidance

- Prefer primitive collision shapes for dynamic entities.
- Keep visual meshes and collision meshes separate.
- Use sensors for triggers, pickups, and detection volumes.
- Use fixed update logic for physics-driven gameplay.
- Tune gravity and meter scale before authoring a large scene.

Next steps

Bring everything together and ship your game in Export Window (p. 155).

Audio

Doriax uses **SoLoud** for audio playback and exposes sound behavior through the `Sound` object class, `SoundComponent`, `SoundPool`, and `AudioSystem`. Audio entities are regular ECS entities — you position them in the world just like any other object, and the audio system reads their Transform to compute spatial attenuation.

Adding a sound to a scene

Lua

```lua
sound = Sound(scene)
sound:loadSound("audio/jump.ogg")
sound.volume = 0.8
sound:play()
```

C++

```cpp
Sound jump(&scene);
jump.loadSound("audio/jump.ogg");
jump.setVolume(0.8f);
jump.play();
```

> **Lua uses properties, C++ uses setters**
> Most sound parameters are exposed to Lua as **properties** (`sound.volume = 0.8`), while C++ uses setter methods (`sound.setVolume(0.8f)`). The property/method names below list both forms.

Core features

Feature	Lua property	C++ method	Notes
Playback control	`play()`, `pause()`, `stop()`	same	Transport controls (call `play()` again to resume a paused sound)
Looping	`sound.looping`	`setLooping(bool)`	Loop music or ambient sound
Volume	`sound.volume`	`setVolume(float)`	Per-sound level from 0.0 to 1.0+
Pitch / speed	`sound.speed`	`setSpeed(float)`	1.0 = normal, 2.0 = double speed
Pan	`sound.pan`	`setPan(float)`	–1.0 = full left, 0 = center, +1.0 = full right
Seek	`seek(seconds)`	same	Jump to a position in the audio
3D mode	`sound.sound3D`	`setSound3D(bool)`	Enable spatialization (or pass `true` to the constructor)
Attenuation	`sound.attenuationModel`, `sound.minDistance`, `sound.maxDistance`	`setAttenuationModel()`, `setMinDistance()`, `setMaxDistance()`	Distance-based volume falloff
Doppler effect	`sound.dopplerFactor`	`setDopplerFactor(float)`	Pitch shift from relative motion

3D spatial audio

Attach a sound to a moving entity to get automatic position-based spatialization. The audio system reads the entity's world transform every frame.

Pass `true` to the `Sound` constructor (or set `sound.sound3D = true`) to make a sound spatial. A 3D sound is positioned by its entity's Transform, so parent it under the moving entity in the editor's Structure panel (or in the scene hierarchy).

Lua

```lua
-- 3D footstep sound (the `true` enables spatialization)
footstep = Sound(scene, true)
footstep:loadSound("audio/footstep.ogg")
footstep.attenuationModel = SoundAttenuation.INVERSE_DISTANCE
footstep.minDistance = 1.0
footstep.maxDistance = 20.0
footstep:play()
```

C++

```cpp
Sound footstep(&scene, true);
footstep.loadSound("audio/footstep.ogg");
footstep.setAttenuationModel(SoundAttenuation::INVERSE_DISTANCE);
footstep.setMinDistance(1.0f);
footstep.setMaxDistance(20.0f);
footstep.play();
```

Attenuation models

SoundAttenuation	Behavior
`NO_ATTENUATION`	No distance attenuation — volume is constant regardless of distance
`INVERSE_DISTANCE`	Volume decreases inversely with distance (realistic for point sources)
`LINEAR_DISTANCE`	Volume falls off linearly between min and max distance
`EXPONENTIAL_DISTANCE`	Exponential falloff, louder near the source

Global controls

`AudioSystem` provides scene-wide controls:

Lua

```lua
AudioSystem.globalVolume = 0.5     -- halve all audio (property, not a setter)
AudioSystem.pauseAll()
AudioSystem.resumeAll()
AudioSystem.stopAll()
```

C++

```cpp
AudioSystem::setGlobalVolume(0.5f);
AudioSystem::pauseAll();
AudioSystem::resumeAll();
AudioSystem::stopAll();
```

SoundPool

For sounds that play frequently (gunshots, footsteps, coins), the engine caches decoded audio data in an internal `SoundPool` so the same file is not loaded multiple times. Loading the same path from several `Sound` instances reuses the cached data automatically — you do not need to manage the pool yourself.

```
// Each Sound that loads the same path shares the cached audio data.
Sound coin1(&scene);
coin1.loadSound("audio/coin.ogg");

Sound coin2(&scene);
coin2.loadSound("audio/coin.ogg");  // served from the shared cache
```

A file that fails to load logs an error and sets `SoundComponent.loadFailed`. It is not tried again until a scene that uses it loads or `loadSound` is called for it.

Background music

Music typically loops, has no 3D attenuation, and plays at a lower priority than sound effects:

Lua

```
music = Sound(scene)
music:loadSound("audio/music_level1.ogg")
music.looping = true
music.volume = 0.4
music:play()
```

A scene's sounds stop when the scene is deleted, looping ones included, so each level can own its music and a scene switch ends it. Music meant to carry on across scenes goes in a scene that stays, such as a child scene the next stack keeps.

Audio workflow checklist

1. Add audio files to the project resources folder (OGG or WAV recommended).
2. Create a `Sound` entity, set the resource path, and configure volume and loop.
3. For 3D audio: parent the sound to a spatial entity and configure attenuation.
4. For music: mark the sound as looping; keep it non-spatial.
5. Use `AudioSystem` for master volume and global pause.
6. Test on target hardware — mobile speakers and headphones differ significantly.

Practical tips

- Keep sound effect files short and compressed; use OGG/Vorbis for best size-to-quality ratio.
- Use WAV for very short, frequently-triggered sounds where decoding latency matters.
- Non-spatial sounds (UI clicks, music) should not be 3D — leave `sound3D` off (the default) so they play at full volume regardless of listener position.
- Tune attenuation `minDistance` and `maxDistance` in the actual scene scale so the falloff feels natural.
- Separate music, ambient, and SFX into named volume categories so players can control them independently in settings.

See also

- Sound (p. 475) — full API reference
- AudioSystem (p. 591) — global controls

Math & Utilities

Doriax includes a complete set of math and utility types that are shared across rendering, physics, animation, input, and gameplay code. Understanding these types helps you write efficient, readable Doriax scripts.

Vector types

Type	Dimensions	Typical uses
`Vector2` (p. 656)	2D (x, y)	Positions, sizes, velocities, UV coordinates
`Vector3` (p. 660)	3D (x, y, z)	Positions, directions, scales, normals
`Vector4` (p. 664)	4D (x, y, z, w)	Colors (RGBA), homogeneous coordinates, packed data

```lua
-- Create vectors
local pos = Vector3(10, 5, 0)
local dir = Vector3(0, 1, 0)
local target = Vector3(0, 5, 0)

-- Arithmetic
local newPos = pos + dir * 3.0   -- 3.0 * dir works too
local dist   = pos:length()
local unit   = dir:normalized()
local dot    = pos:dotProduct(dir)

-- Step toward a point without passing it, or blend toward it
local stepped = pos:moveTowards(target, 4 * Engine.deltatime)
local halfway = pos:lerp(target, 0.5)
```

All three vector types have `moveTowards` and `lerp`. `moveTowards` moves at a fixed speed and stops exactly on the target, which suits movement driven by `deltatime`. `lerp` covers a fraction of the remaining distance, so it slows down near the target.

Common static constants

Constant	Value
`Vector3.ZERO`	(0, 0, 0)
`Vector3.UNIT_SCALE`	(1, 1, 1)
`Vector3.UNIT_X`	(1, 0, 0)
`Vector3.UNIT_Y`	(0, 1, 0)
`Vector3.UNIT_Z`	(0, 0, 1)

Changing one component of a property

Properties such as `position`, `scale`, `rotation`, `color` or `linearVelocity` return a copy of the value. Changing a component of that copy does not reach the object, so `self.object.position.y = 2` runs without error but leaves the object where it was. Assign the whole value instead:

```lua
local pos = self.object.position
pos.y = 2                    -- only the copy changes
self.object.position = pos   -- now the object moves

-- the same, building a new vector
self.object.position = Vector3(pos.x, 2, pos.z)

-- or with a setter that takes the components
self.object:setPosition(pos.x, 2, pos.z)
```

Not every property has such a setter: `setScale` takes one uniform factor, and bodies have none for `position` or `linearVelocity`.

The same holds for methods that change a vector in place, such as `normalize()`, for values returned by methods like `object:getWorldPosition()`, and for list properties such as `scene.postProcessPasses`: they act on a copy. Code Editor (p. 136) completion marks properties that return a copy, e.g. `Object.position -> Vector3 (copy)`.

Component fields are the exception: Lua gets the field itself, not a copy, so `image:getUIComponent().color.x = 1` changes the component. Some fields also need their `needUpdate…` flag set; see Writing components directly (p. 191).

Quaternion

`Quaternion` (p. 644) represents 3D rotation. Prefer quaternions over Euler angles for rotation interpolation and composition.

```
-- Create from Euler angles (ZYX order; degrees if useDegrees is enabled)
local q = Quaternion(0, 90, 0)

-- Interpolate (interpolation factor comes first)
local result = Quaternion.slerp(0.5, q1, q2)

-- Rotate a direction
local rotated = q * Vector3(1, 0, 0)  -- same as q:xAxis()

-- Face a target with the local +Z axis
obj.rotation = Quaternion.lookRotation(target - obj.position)
```

Matrix types

Type	Use
`Matrix3` (p. 637)	3×3 transforms, normal matrix
`Matrix4` (p. 638)	4×4 transforms, projection, view matrix

Matrices are used internally by the renderer and physics, but you can construct them for custom transforms or pass them to shader uniforms.

Rect

`Rect` (p. 652) stores a 2D rectangle as (x, y, width, height). Use it for UI bounds, sprite atlas frames, and screen-space regions.

```
local frame = Rect(64, 0, 32, 32)   -- x=64, y=0, w=32, h=32
local contains = frame:contains(Vector2(70, 10))
```

Bounds and spatial queries

Class	Use
`AABB` (p. 632)	Axis-aligned bounding box — fast broad-phase tests, frustum culling
`OBB` (p. 639)	Oriented bounding box — rotated objects that AABB over-approximates
`Sphere` (p. 654)	Sphere bounds — broad-phase tests, shadow radius, audio range
`Ray` (p. 649)	Ray from origin in a direction — picking, raycasts, line-of-sight
`Plane` (p. 642)	Infinite plane defined by normal and distance — frustum planes, clip tests

```lua
-- Cast a ray through the cursor and test it against the scene's 3D physics bodies
local pos = Input.getMousePosition()
local ray = camera:screenToRay(pos.x, pos.y)
local result = ray:intersects(scene, RayFilter.BODY_3D)
if result.hit then
    print("Hit entity:", result.body, "at distance", result.distance)
end
```

`intersects` also accepts individual `AABB`, `OBB`, `Plane`, `Body2D`, and `Body3D` targets, and `RayFilter.BOUNDS` tests every mesh's bounding box without any physics. Pass an entity or a list of entities to skip those bodies so the ray continues through them (`ray:intersects(scene, RayFilter.BODY_3D, self.entity)`). The returned `RayReturn` carries `hit`, `distance`, `point`, `normal`, `body` (the entity), and `shapeIndex`.

Angles

The `Angle` (p. 636) utility class provides explicit degree/radian conversions. The engine can be configured to use degrees everywhere with `Engine::setUseDegrees(true)`.

```lua
-- Convert explicitly regardless of engine setting
local rad = Angle.degToRad(90)
local deg = Angle.radToDeg(math.pi / 2)
```

Color utilities

The static `Color` (p. 709) class provides sRGB ↔ linear conversions. Rendering expects linear color space internally.

```lua
-- Convert from sRGB to linear for the renderer
local linearRed = Color.sRGBToLinear(Vector3(1, 0, 0))

-- And back for display values
local srgb = Color.linearTosRGB(linearRed)
```

Logging

`Log` (p. 387) provides formatted console output at different severity levels.

Lua

```lua
-- The Lua functions take a single string; use .. for formatting
Log.print("Scene loaded")
Log.debug("Player position: " .. tostring(obj.position))
Log.warn("Optional resource missing: " .. path)
Log.error("Physics body creation failed")
```

C++

```cpp
// The C++ functions accept printf-style format arguments
Log::print("Scene loaded");
Log::debug("Player position: %s", position.toString().c_str());
Log::warn("Optional resource missing: %s", path.c_str());
Log::error("Physics body creation failed");
```

Use `Log::debug` for development-only output. Release builds can strip debug logs by setting the appropriate log level.

Object utility

The `Object` (p. 455) class wraps an entity handle and provides transform, visibility, parenting, and basic physics convenience APIs. Use it when you need to access transform data from a script without knowing the specific object type.

```
local Mover = {
    properties = {
        { name = "speed", displayName = "Speed", type = "float", default = 50.0 }
    }
}

function Mover:init()
    RegisterEngineEvent(self, "onUpdate")
end

function Mover:onUpdate()
    local obj = Object(self.scene, self.entity)
    local dt = Engine.deltatime
    obj.position = obj.position + Vector3(0, self.speed * dt, 0)
end

return Mover
```

Base64

The `Base64` (p. 708) utility class encodes and decodes binary data as base64 strings. Useful for embedding small binary data in JSON or YAML, or for network/save serialization. **C++ only** — `encode` / `decode` are not exposed to Lua yet.

See also

- Vector2 (p. 656), Vector3 (p. 660), Vector4 (p. 664)
- Quaternion (p. 644)
- Rect (p. 652)
- AABB (p. 632), OBB (p. 639), Ray (p. 649)
- Log (p. 387)
- Color (p. 709), Angle (p. 636)

Threading & Async Loading

Doriax includes thread utilities and async loading support for resource-heavy work such as loading large models, generating terrain, or pre-warming audio pools in the background. Threading support depends on the target platform and build options.

Why async loading?

Synchronous loading freezes the game loop for the duration of the load. For large assets (multi-megabyte models, terrain heightmaps, audio files), this produces noticeable hitches. Async loading moves the heavy work to a worker thread while the main thread continues rendering a loading screen or animating a progress indicator.

Automatic async resource loading

The simplest way to use threading is to enable **async loading** and let the engine handle the rest. When enabled, the resource pools (textures, models, sounds, shaders) load data on a worker thread pool automatically:

Lua

```lua
Engine.asyncLoading = true   -- property in Lua
Engine.setMaxResourceLoadingThreads(4)
```

C++

```cpp
Engine::setAsyncLoading(true);
Engine::setMaxResourceLoadingThreads(4);
```

While resources load, the entity is created immediately and the resource appears when its data is ready. Query `Engine.getQueuedResourceCount()` to see how many resources are still pending.

Async thread scope (manual background work, C++)

For your own background work, bracket it with `Engine::AsyncThreadScope` (an RAII helper around `Engine::startAsyncThread()` / `Engine::endAsyncThread()`). While inside an async scope, GPU resource creation is **deferred to a commit queue** instead of touching the graphics API from the wrong thread. When the outermost scope ends, the queue is flushed automatically (you can also flush manually with `Engine::commitThreadQueue()` from the main thread):

C++

```cpp
#include "Engine.h"

void loadLevelInBackground() {
    Engine::AsyncThreadScope asyncScope;

    // Heavy work on the worker thread. Engine objects created here defer
    // their GPU-side resources to the commit queue.
    model.loadGLTF("models/level.gltf");

} // scope ends → queued GPU work is committed
```

Use `Engine::isAsyncThread()` to check whether the current code runs inside an async scope.

Thread pool

`ThreadPoolManager` is the worker pool behind async resource loading. In C++ you can also dispatch your own tasks to it. `enqueue` is an instance method — get the singleton first:

C++

```
ThreadPoolManager::getInstance().enqueue([]() {
    // parse data, decompress files, run a simulation step, etc.
});
```

The pool is initialized on demand with one thread per hardware core by default; `Engine::setMaxResourceLoadingThreads(n)` re-initializes it with a specific worker count. The pool is shared across the engine; avoid blocking workers with long synchronous operations. (In Lua only `initialize`, `shutdown`, and `getQueueSize` are exposed — task dispatch is C++-only.)

Tracking loading progress

`ResourceProgress` is a static tracker that reports per-resource and overall build progress while async loads are running. Read the overall progress to drive a loading bar:

Lua

```
function LoadingScreen:onUpdate()
    if ResourceProgress.hasActiveBuilds() then
        local overall = ResourceProgress.getOverallProgress()
        progressBar.value = overall.totalProgress * 100
        statusText.text = overall.currentBuildName
    end
end
```

C++

```
if (ResourceProgress::hasActiveBuilds()) {
    OverallBuildProgress overall = ResourceProgress::getOverallProgress();
    progressBar.setValue(overall.totalProgress * 100.0f);
}
```

`OverallBuildProgress` carries `totalProgress` (0–1), `totalBuilds`, `completedBuilds`, `currentBuildName`, `currentBuildType`, and `hasActiveBuilds`. Your own loaders can participate by calling `ResourceProgress.startBuild(id, type, name)`, `updateProgress(id, value)`, and `completeBuild(id)` / `failBuild(id)`.

Platform caveats

Platform	Threading behavior
Desktop (Windows, Linux, macOS)	Full `std::thread` support; thread pool fully functional
Android	Full thread support via POSIX threads
iOS	Full thread support via POSIX threads
HTML5 (Emscripten)	Threads require `SharedArrayBuffer` and COOP/COEP headers on the server. Without pthreads, the engine defines `NO_THREAD_SUPPORT` and runs single-threaded

For web, always test async loading with pthreads enabled in your deployment environment.

Practical rules

- Prefer the built-in async loading (`Engine.asyncLoading = true`) over manual threads — the resource pools already handle the thread-safety details.

- For manual background work, wrap it in `Engine::AsyncThreadScope` so GPU resource creation is deferred to the commit queue instead of running on the wrong thread.
- Spawn physics bodies and mutate scene hierarchies **only on the main thread**.
- Do not hold scene references or entity handles on worker threads without synchronization.
- Test async loading on the slowest target device — the loading time and race conditions may not reproduce on a fast desktop machine.

Loading screens

`SceneManager` shows a loading screen for you. Pick a UI scene as the **Loading Scene** in Project Settings → General → Loading (p. 56), turn on **Async Loading** there, and every `loadScene` goes through it:

1. The loading scene appears on top of the running scene, which keeps running below it but no longer gets UI input.
2. After the loading delay, the old scene is replaced behind it. The new scene's entities and scripts are built on the main thread, so the game freezes for that moment with the loading scene on screen.
3. With async loading on, the new scene's textures, sounds and models stream in on worker threads while the loading scene keeps animating. Once they are loaded, `Engine.onSceneLoaded` fires and the loading scene is removed, or fades out first if a script holds the load (p. 606).

The start scene goes through it too, with nothing running below the loading scene yet.

The same settings can be made from code, at startup:

Lua

```lua
Engine.asyncLoading = true
SceneManager.setLoadingScene("Loading")
SceneManager.loadingDelay = 0.3

-- LoadingScreen.lua, on the Loading scene
function LoadingScreen:onUpdate()
    if not SceneManager.loading then return end
    self.bar.width = 360 * SceneManager.loadingProgress
end
```

C++

```cpp
Engine::setAsyncLoading(true);
SceneManager::setLoadingScene("Loading");
SceneManager::setLoadingDelay(0.3f);

// LoadingScreen.cpp, on the Loading scene
void LoadingScreen::onUpdate() {
    if (!SceneManager::isLoading()) return;
    bar->setWidth((unsigned int)(360 * SceneManager::getLoadingProgress()));
}
```

A level script starts gameplay in `Engine.onSceneLoaded` (p. 373): the level is loaded, and the loading scene still covers it while it places the player and camera. See SceneManager.setLoadingScene (p. 605).

Preloading the next scene

A loading screen hides the wait; preloading removes most of it. `SceneManager.preloadScene` (p. 607) loads the next scene's textures, sounds and models on worker threads while the player is still in the current one. When `loadScene` runs later, only the scene's entities are built, and the loading screen closes almost at once.

Lua

```lua
SceneManager.preloadScene("Level2")          -- near the end of Level1
if SceneManager.getPreloadProgress("Level2") == 1 then
    -- everything is loaded
end
```

C++

```cpp
SceneManager::preloadScene("Level2");        // near the end of Level1
if (SceneManager::getPreloadProgress("Level2") == 1.0f) {
    // everything is loaded
}
```

See also

- ThreadPoolManager (p. 719)
- ResourceProgress (p. 717)
- Data (p. 703)
- File (p. 705)

Monetization

Doriax has three service classes for earning from a game. They are static classes that any script can call, and each one talks to a platform SDK that an export includes only when it is enabled in Project Settings (p. 60):

Class	Service	Platforms	Enabled by
`AdMob` (p. 610)	Google Mobile Ads: banner, interstitial, rewarded, rewarded interstitial and app open ads, with Google's consent form	Android, iOS	**Google AdMob**, in the Android and iOS sections
`InAppPurchase` (p. 618)	Google Play Billing and the App Store (StoreKit 2): one-time products and subscriptions	Android, iOS	**Google Play Billing** in the Android section, **App Store Purchases** in the iOS section
`WebPortal` (p. 626)	The SDK of one web game portal: ads, gameplay and loading events, cloud saves	Web	**Game Portal**, in the Web section

The classes exist on every platform, so the same scripts build and run everywhere. Where a service is missing (desktop, the editor's Play, an export with the setting off), calls that expect an answer report "not available" through their usual event, and the first call logs a warning naming the setting to turn on.

How results arrive

Every result arrives as an event, never as a return value: an ad loaded, a purchase went through, a portal finished starting. The platform answers on its own thread. The engine queues each answer and runs the subscribers at the start of the next frame, on the game thread and before the scenes update, so a handler can use the scene like any other script code.

Subscribe with the same helpers as any other event (p. 221):

C++

```
// Ads.h
#include "ScriptBase.h"
#include "AdMob.h"

class Ads : public doriax::ScriptBase {
public:
    Ads(doriax::Scene* scene, doriax::Entity entity);
    ~Ads();

    void onAdLoaded(doriax::AdMobFormat format);
};

// Ads.cpp
Ads::Ads(Scene* scene, Entity entity): ScriptBase(scene, entity) {
    REGISTER_EVENT(AdMob::onAdLoaded, onAdLoaded);
}

Ads::~Ads() {
    UNREGISTER_EVENT(AdMob::onAdLoaded, onAdLoaded);
}

void Ads::onAdLoaded(AdMobFormat format) {
}
```

Lua

```lua
function Ads:init()
    RegisterEvent(self, AdMob.onAdLoaded, "onAdLoaded")
end

function Ads:onAdLoaded(format)
end
```

- A C++ handler takes the parameters exactly as the event declares them, by value (`std::string`, not `const std::string&`), or the macro does not compile.
- The Code Editor's Add event (p. 137) menu lists these events under **AdMob**, **In-App Purchase** and **Web Portal**, and writes the registration and the handler for you.
- The events belong to the class, not to a scene, so they keep firing across scene changes. A Lua script's `RegisterEvent` subscriptions are removed with the script, and a C++ script removes its own with `UNREGISTER_EVENT`. A function assigned straight to an event (`AdMob.onAdLoaded = function(format) ... end`) stays until the game shuts down.

Ads pause the game

While a full-screen ad covers the game, the engine pauses just as when the app goes to the background: sounds pause, the update loop stops, and `Engine::onPause` (p. 373) fires. `Engine::onResume` (p. 373) follows when the ad closes. The same happens while a banner's click opens a page over the game, and on the web while a portal ad plays or a portal pauses the game itself.

On Android the whole app pauses while a full-screen ad shows, so the events of that ad (`onAdShown`, `onAdImpression`, `onUserEarnedReward`, ...) arrive together when it closes.

AdMob

Setup

1. Create the app and its ad units in the AdMob console[13].
2. In **Project Settings → Platforms**, open the **Android** and **iOS** sections, tick **Google AdMob**, and paste the **AdMob App ID** (`ca-app-pub-…~…`). Left empty, the export uses Google's sample app, which only serves test ads.
3. On Android, declare in the Play Console that the app contains ads and uses the advertising ID. Exports raise **Min SDK** to 24 for AdMob.
4. On iOS, fill in **Tracking Description** when your consent message asks for tracking permission.
5. Export as **Source Code** and build with Android Studio or Xcode. Ads only run on a device, emulator or simulator; the editor has no AdMob.

See Android (p. 62) and iOS (p. 62) settings for what each field writes.

Consent, then initialize

Google's policies require a consent message for users in the EEA, the UK and Switzerland, and privacy messages for some US states. `AdMob` shows both through Google's User Messaging Platform, so a game asks first and starts Google Mobile Ads after:

1. Call `requestConsent()` (p. 612) at every launch. It updates the consent information and shows Google's consent form when the user still has to answer.

13. https://admob.google.com

2. `onConsentUpdated` (p. 615) reports the result (`errorCode` 0 on success). When `canRequestAds()` (p. 613) is true, call `initialize()` (p. 612). Consent given in an earlier launch counts, even if this request failed offline.
3. `onInitialized` fires once Google Mobile Ads is ready. Load the first ads there.

Audience settings such as `setAgeRestriction` (p. 612) and `setMaxAdContentRating` (p. 612) go before `requestConsent()`.

C++

```
// Ads.h
#include "ScriptBase.h"
#include "AdMob.h"

class Ads : public doriax::ScriptBase {
public:
    Ads(doriax::Scene* scene, doriax::Entity entity);
    ~Ads();

    void showBreak();

    void onConsentUpdated(int errorCode, std::string message);
    void onInitialized();
    void onAdDismissed(doriax::AdMobFormat format);
};

// Ads.cpp
#include "Ads.h"

using namespace doriax;

static const char* interstitialId = "ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx";

Ads::Ads(Scene* scene, Entity entity): ScriptBase(scene, entity) {
    REGISTER_EVENT(AdMob::onConsentUpdated, onConsentUpdated);
    REGISTER_EVENT(AdMob::onInitialized, onInitialized);
    REGISTER_EVENT(AdMob::onAdDismissed, onAdDismissed);

    AdMob::requestConsent();
}

Ads::~Ads() {
    UNREGISTER_EVENT(AdMob::onConsentUpdated, onConsentUpdated);
    UNREGISTER_EVENT(AdMob::onInitialized, onInitialized);
    UNREGISTER_EVENT(AdMob::onAdDismissed, onAdDismissed);
}

void Ads::onConsentUpdated(int errorCode, std::string message) {
    if (errorCode != 0) {
        Log::warn("Consent: %s", message.c_str());
    }
    if (AdMob::canRequestAds() && !AdMob::isInitialized()) {
        AdMob::initialize();
    }
}

void Ads::onInitialized() {
    AdMob::loadInterstitialAd(interstitialId);
}

// At a natural break, like the end of a level
void Ads::showBreak() {
    if (AdMob::isInterstitialAdLoaded()) {
        AdMob::showInterstitialAd();
    }
}

// Each ad shows once, so load the next one
```

```
void Ads::onAdDismissed(AdMobFormat format) {
    if (format == AdMobFormat::INTERSTITIAL) {
        AdMob::loadInterstitialAd(interstitialId);
    }
}
```

Lua

```lua
local Ads = {}

local INTERSTITIAL_ID = "ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx"

function Ads:init()
    RegisterEvent(self, AdMob.onConsentUpdated, "onConsentUpdated")
    RegisterEvent(self, AdMob.onInitialized, "onInitialized")
    RegisterEvent(self, AdMob.onAdDismissed, "onAdDismissed")

    AdMob.requestConsent()
end

function Ads:onConsentUpdated(errorCode, message)
    if errorCode ~= 0 then
        Log.warn("Consent: " .. message)
    end
    if AdMob.canRequestAds() and not AdMob.isInitialized() then
        AdMob.initialize()
    end
end

function Ads:onInitialized()
    AdMob.loadInterstitialAd(INTERSTITIAL_ID)
end

-- At a natural break, like the end of a level
function Ads:showBreak()
    if AdMob.isInterstitialAdLoaded() then
        AdMob.showInterstitialAd()
    end
end

-- Each ad shows once, so load the next one
function Ads:onAdDismissed(format)
    if format == AdMobFormat.INTERSTITIAL then
        AdMob.loadInterstitialAd(INTERSTITIAL_ID)
    end
end

return Ads
```

When `isPrivacyOptionsRequired()` (p. 613) is true, Google requires a way back to the consent form, such as a **Privacy** button in the settings menu that calls `showPrivacyOptionsForm()`.

Full-screen ads

Interstitial, rewarded, rewarded interstitial and app open ads work the same way:

Step	Call	Result
Load	`loadInterstitialAd(adUnitId)`, `loadRewardedAd`, `loadRewardedInterstitialAd`, `loadAppOpenAd`	`onAdLoaded(format)` or `onAdFailedToLoad(format, errorCode, message)`
Check	`isInterstitialAdLoaded()`, …	`true` between `onAdLoaded` and the show
Show	`showInterstitialAd()`, …	`onAdShown`, `onAdImpression`, then `onAdDismissed`; or `onAdFailedToShow`

- Loading a format again replaces its ad, and only the latest load reports.

- An ad shows once: `is...Loaded()` turns false when it is shown, so load the next one, usually in `onAdDismissed`.
- Showing an ad that is not loaded reports `onAdFailedToShow` with `AdMob::errorNotLoaded` (p. 615).
- A failed load reports Google's error code. Retry after a delay rather than at once: Google advises against reloading in a loop.

Rewarded and **rewarded interstitial** ads grant their reward through `onUserEarnedReward(format, type, amount)`, with the type and amount set on the ad unit. Grant it there, not in `onAdDismissed`: a player who closes the ad early earns nothing. For rewards checked by your server, call `setServerSideVerificationOptions` (p. 614) before showing.

App open ads cover the app at launch or when it returns from the background. Google stops serving one four hours after it loaded, so `isAppOpenAdLoaded()` turns false by then, and the ad needs loading again.

> **`Engine::onResume` also follows every ad**
>
> Since full-screen ads pause the game, `onResume` fires after each of them, and after anything else that briefly covers the app, like Google Play's purchase screen. A game that shows an app open ad from `onResume` needs a flag, set before showing its other ads, that skips those resumes.

Banners

`loadBannerAd(adUnitId, size, position)` (p. 613) places a banner over the game, at the bottom by default. It shows as soon as it loads, replacing any banner already there.

C++

```
AdMob::loadBannerAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx", AdMobBannerSize::ADAPTIVE,
AdMobBannerPosition::TOP);
```

Lua

```
AdMob.loadBannerAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx", AdMobBannerSize.ADAPTIVE,
AdMobBannerPosition.TOP)
```

- `ADAPTIVE`, the default size, takes the screen width when it loads and lets Google pick the height. Load it again after the screen rotates.
- `hideBannerAd()` and `showBannerAd()` toggle it, `setBannerAdPosition()` moves it, and `removeBannerAd()` destroys it.
- `getBannerAdWidth()` and `getBannerAdHeight()` give its size in screen pixels once loaded, to keep game UI clear of it. On Android the banner also stays out of display cutouts and visible system bars.
- Banners refresh at the rate set on their ad unit. A refresh that fails reports `onAdFailedToLoad(BANNER, ...)`, but the last ad keeps showing.

Audience and privacy

Call	Use
`setAgeRestriction(CHILD)`	The game is for children. Requests get Google's child treatment, and `requestConsent()` asks as for a user under the age of consent
`setAgeRestriction(TEEN)`	The game is for teens. Requests get Google's teen treatment
`setMaxAdContentRating(rating)`	Highest content rating of ads: `GENERAL`, `PARENTAL_GUIDANCE`, `TEEN` or `MATURE_AUDIENCE`
`setPersonalization(DISABLED)`	Only non-personalized ads

These apply to the requests made after them. See Request settings (p. 612).

Testing

Clicking live ads from your own devices breaks AdMob's policies, so use test ads while developing:

- Use Google's demo ad units, listed for Android[14] and iOS[15], or
- register your devices with `setTestDeviceIds` (p. 612). Google Mobile Ads prints the device's hashed ID to the device log (Logcat or the Xcode console) on the first request.

On a test device, `setConsentDebugGeography(AdMobDebugGeography::EEA)` makes the consent form appear as it would in Europe, and `resetConsent()` forgets the last answer. `openAdInspector()` opens Google's Ad Inspector on a test device, to check each ad unit's requests and mediation results.

Revenue and volume

`onAdPaid(format, valueMicros, currencyCode, precision)` reports what each ad earned, in millionths of the currency, for your own analytics. `setAppVolume(volume)` and `setAppMuted(muted)` tell Google how loud video ads should play relative to the game, so they match a game whose volume is turned down.

In-app purchases

`InAppPurchase` sells one-time products and subscriptions through Google Play Billing on Android and StoreKit 2 on iOS, with one API: products are queried, bought and settled the same way on both. The App Store differences are listed at the end.

Setup

Android

1. Tick **Google Play Billing** in **Project Settings → Platforms → Android**. Exports raise **Min SDK** to 23 for it.
2. In the Play Console, create the products: **one-time products** (consumable or not) and **subscriptions** with their base plans and offers.
3. Upload a build to a testing track (internal testing is enough) with the same package name, and add your Google accounts as license testers so their purchases are not charged. Google Play only sells the products of an app it knows.

iOS

1. Tick **App Store Purchases** in **Project Settings → Platforms → iOS**. Building it needs Xcode 16.3 or later.
2. In App Store Connect, create the in-app purchases (consumable, non-consumable or non-renewing subscription) and the auto-renewable subscriptions, which belong to a subscription group.
3. Test with a Sandbox Apple Account, or with a StoreKit configuration file selected in the Xcode scheme, which works before anything is set up in App Store Connect.

On every other platform, and in builds without the setting, the calls report `BillingResponse::BILLING_UNAVAILABLE`.

The purchase flow

1. `initialize()` (p. 619) connects to the store. `onInitialized(response, message)` reports `OK` when it is ready; calls made before that fail with `SERVICE_DISCONNECTED`. On Android, a lost connection (`onDisconnected`) comes back by itself on the next call.
2. `queryProducts(ids, type)` (p. 619) fetches the localized names and prices. After `onProductsQueried`, read them with `getProducts()` or `getProduct(id)` to fill the store screen.

14. https://developers.google.com/admob/android/test-ads
15. https://developers.google.com/admob/ios/test-ads

3. `purchase(productId)` (p. 620) opens the store's purchase screen for a queried product. The result is `onPurchaseUpdated(purchase)`, or `onPurchaseFailed(productId, response, message)` when the player cancels (`USER_CANCELED`) or something goes wrong.
4. In `onPurchaseUpdated`, grant the product only when `purchase.state` is `PURCHASED`. A `PENDING` purchase waits for something outside the app, like a cash payment at a store or a parent approving it with Ask to Buy, and the event fires again when it completes.
5. Settle the purchase: `consumePurchase(token)` (p. 620) for consumables, which can then be bought again, and `acknowledgePurchase(token)` (p. 620) for everything else. Google Play refunds a purchase left unsettled for three days, and the App Store delivers it again at every launch.

> **Grant a consumable before consuming it, once per token**
>
> The same purchase can be reported more than once: by the purchase screen, by a query, again at the next launch. Once consumed, it is never reported again. So grant and save a consumable *before* consuming it, and remember its `purchaseToken` so that a second report does not grant it twice.

Purchases also complete outside the purchase screen: a pending payment clears, a promo code is redeemed, the player buys on another device. Call `queryPurchases(type)` (p. 621) for each product type at startup and on `Engine::onResume`. It reports every owned purchase through `onPurchaseUpdated` (with `restored` set), then `onPurchasesQueried`, and drops refunded and expired ones from `getPurchases()` and `isPurchased(productId)` (p. 621).

Give the store screen a **Restore Purchases** button too, which calls `restorePurchases()` (p. 621): it queries both types, and on iOS first syncs the player's App Store account, which may ask them to sign in. Apple's review guidelines ask for a way to restore purchases that can be restored.

C++

```
// Store.h (declarations)
void buyCoins();
void restore();

void onStoreReady(doriax::BillingResponse response, std::string message);
void onResume();
void onPurchaseUpdated(doriax::PurchaseDetails purchase);
void onPurchaseFailed(std::string productId, doriax::BillingResponse response, std::string message);

// Store.cpp
static const std::string coinsId = "coins_100";    // consumable
static const std::string noAdsId = "remove_ads";   // bought once

Store::Store(Scene* scene, Entity entity): ScriptBase(scene, entity) {
    REGISTER_EVENT(InAppPurchase::onInitialized, onStoreReady);
    REGISTER_EVENT(InAppPurchase::onPurchaseUpdated, onPurchaseUpdated);
    REGISTER_EVENT(InAppPurchase::onPurchaseFailed, onPurchaseFailed);
    REGISTER_ENGINE_EVENT(onResume);

    InAppPurchase::initialize();
}

void Store::onStoreReady(BillingResponse response, std::string message) {
    if (response != BillingResponse::OK) {
        Log::warn("Store unavailable: %s", message.c_str());
        return;
    }
    InAppPurchase::queryProducts({coinsId, noAdsId}, ProductType::INAPP);
    InAppPurchase::queryPurchases(ProductType::INAPP);
}

void Store::onResume() {
    if (InAppPurchase::isReady()) {
        InAppPurchase::queryPurchases(ProductType::INAPP);
    }
```

```
}

// From a Buy button
void Store::buyCoins() {
    InAppPurchase::purchase(coinsId);
}

// From a Restore Purchases button
void Store::restore() {
    InAppPurchase::restorePurchases();
}

void Store::onPurchaseUpdated(PurchaseDetails purchase) {
    if (purchase.state != PurchaseState::PURCHASED) return;

    if (purchase.productId == coinsId) {
        // Grant and save once per token, then consume
        const std::string granted = "granted_" + purchase.purchaseToken;
        if (!UserSettings::getBoolForKey(granted.c_str(), false)) {
            UserSettings::setIntegerForKey("coins", UserSettings::getIntegerForKey("coins", 0) + 100);
            UserSettings::setBoolForKey(granted.c_str(), true);
        }
        InAppPurchase::consumePurchase(purchase.purchaseToken);
    } else if (!purchase.acknowledged) {
        InAppPurchase::acknowledgePurchase(purchase.purchaseToken);
    }
    // remove_ads is checked with InAppPurchase::isPurchased(noAdsId)
}

void Store::onPurchaseFailed(std::string productId, BillingResponse response, std::string message) {
    if (response != BillingResponse::USER_CANCELED) {
        Log::warn("Purchase failed: %s", message.c_str());
    }
}
```

Lua

```
local Store = {}

local COINS = "coins_100"     -- consumable
local NO_ADS = "remove_ads"   -- bought once

function Store:init()
    RegisterEvent(self, InAppPurchase.onInitialized, "onStoreReady")
    RegisterEvent(self, InAppPurchase.onPurchaseUpdated, "onPurchaseUpdated")
    RegisterEvent(self, InAppPurchase.onPurchaseFailed, "onPurchaseFailed")
    RegisterEngineEvent(self, "onResume")

    InAppPurchase.initialize()
end

function Store:onStoreReady(response, message)
    if response ~= BillingResponse.OK then
        Log.warn("Store unavailable: " .. message)
        return
    end
    InAppPurchase.queryProducts({COINS, NO_ADS}, ProductType.INAPP)
    InAppPurchase.queryPurchases(ProductType.INAPP)
end

function Store:onResume()
    if InAppPurchase.isReady() then
        InAppPurchase.queryPurchases(ProductType.INAPP)
    end
end

-- From a Buy button
function Store:buyCoins()
    InAppPurchase.purchase(COINS)
```

```
end

-- From a Restore Purchases button
function Store:restore()
    InAppPurchase.restorePurchases()
end

function Store:onPurchaseUpdated(purchase)
    if purchase.state ~= PurchaseState.PURCHASED then
        return
    end
    if purchase.productId == COINS then
        -- Grant and save once per token, then consume
        local granted = "granted_" .. purchase.purchaseToken
        if not UserSettings.getBoolForKey(granted, false) then
            UserSettings.setIntegerForKey("coins", UserSettings.getIntegerForKey("coins", 0) + 100)
            UserSettings.setBoolForKey(granted, true)
        end
        InAppPurchase.consumePurchase(purchase.purchaseToken)
    elseif not purchase.acknowledged then
        InAppPurchase.acknowledgePurchase(purchase.purchaseToken)
    end
    -- remove_ads is checked with InAppPurchase.isPurchased(NO_ADS)
end

function Store:onPurchaseFailed(productId, response, message)
    if response ~= BillingResponse.USER_CANCELED then
        Log.warn("Purchase failed: " .. message)
    end
end

return Store
```

A consumable left unconsumed, because the app closed before the consume finished, comes back through `queryPurchases` or at the next launch; its saved token keeps it from being granted twice, and it is consumed then.

Subscriptions

Query subscriptions with `ProductType::SUBS`, both with `queryProducts` and with `queryPurchases`. On Google Play, each product lists its base plans and offers in `offers` (p. 622), each with an `offerToken` and its `pricingPhases` (p. 623), such as a free week followed by a monthly price. `purchase(productId, offerToken)` buys that plan; without a token it buys the first base plan. An App Store subscription has one plan, so `offers` holds a single entry, with no token, whose first phase is the introductory offer when the player is eligible for it.

- Acknowledge a subscription like any non-consumable. `isPurchased(productId)` is true while it is active.
- A subscription on hold for a payment problem arrives with `suspended` set, and `isPurchased` turns false. `showInAppMessages()` lets Google Play prompt the player to fix the payment, which iOS does by itself, and `openSubscriptionManagement(productId)` opens the store's subscription management.
- `changeSubscription(productId, offerToken, oldPurchaseToken, mode)` upgrades or downgrades an owned subscription. On Google Play, the `SubscriptionReplacementMode` (p. 624) decides how the time left on the old plan is billed. On the App Store, it buys the new product, and the App Store replaces the subscription of the same group.

Security

Purchase data comes from the player's device, which they can tamper with. For anything of value, send the purchase to your server and verify it there before granting: the `purchaseToken` with the Google Play Developer API, or on iOS the `signature`, which is the transaction's JWS, with Apple's App Store Server Library or API. `setObfuscatedAccountId` attaches an id of your own user account to the next purchases, which helps

the store detect fraud and helps your server match a purchase to an account. Use a hash on Google Play; the App Store takes it only as a UUID.

App Store differences

	Google Play	App Store
`purchaseToken`	Google Play's purchase token	The original transaction id, which renewals keep
`orderId`	The order id	The transaction id
`originalJson`, `signature`	The signed purchase data and its signature	The transaction JSON and its JWS
Offers	Base plans and offers, chosen with an `offerToken`	One plan per product. `offerToken` is ignored, and the introductory offer applies by itself
Unsettled purchases	Refunded after three days	Delivered again at every launch
`changeSubscription`	Applies the replacement mode	Buys the new product; `mode` is ignored
`setObfuscatedAccountId`	Any id, ideally a hash	A UUID, kept as the transaction's `appAccountToken`. Anything else is left out with a warning, and the profile id is not used
Pending purchases	Payments made outside the app	Ask to Buy: reported as `PENDING` with no `purchaseToken`, then as a new purchase once approved
Refunds and expirations	Gone after the next `queryPurchases`	Gone after the next `queryPurchases`. Those StoreKit announces while the game runs also arrive through `onPurchaseUpdated`, with state `UNSPECIFIED`
`showInAppMessages()`	Shows Google Play's messages	Does nothing: iOS shows them by itself
`openSubscriptionManagement()`	Opens Google Play's subscription center	Shows Apple's subscription sheet over the game, then queries the subscriptions again
Non-renewing subscriptions	—	`INAPP` products that the store never expires: work out the end from `purchaseTime`

Web portals

Web game portals pay through ads and expect games to report what the player is doing, so the portal can choose the moment for an ad. `WebPortal` gives one API for five portals:

Portal	Game Portal value	Notes
CrazyGames[16]	CrazyGames	SDK v3
Poki[17]	Poki	SDK v2
GameDistribution[18]	GameDistribution	Needs the **Portal Game ID** from the GameDistribution dashboard
Yandex Games[19]	Yandex Games	The SDK only starts inside Yandex Games
YouTube Playables[20]	YouTube Playables	Adds cloud saves and a platform mute

Each export builds in one portal, chosen with **Game Portal** in Project Settings → Platforms → Web (p. 60), so a game sent to several portals needs one export each. `WebPortal::getPortal()` tells which one a build has.

16. https://docs.crazygames.com
17. https://sdk.poki.com
18. https://gamedistribution.com
19. https://yandex.com/dev/games/doc/en/
20. https://developers.google.com/youtube/gaming/playables

Using the portal

1. Call `initialize()` (p. 627) once, at startup. It loads the portal's SDK, and `onInitialized(environment)` reports where the game runs. Calls made after `initialize()` wait for the SDK; calls made before it are dropped, and ads and saves report an error.
2. Call `loadingStop()` once the game can be played. Yandex Games and YouTube Playables require it. Bracket later loading screens with `loadingStart()` and `loadingStop()`.
3. Call `gameplayStart()` when play starts or resumes, and `gameplayStop()` at every break: menus, pauses, the end of a level.
4. Request ads with `requestAd()` (p. 627): `MIDGAME` at natural breaks and `REWARDED` when the player chooses to watch one. The result is `onAdStarted` then `onAdFinished`, or `onAdError(type, code, message)`.
5. Call `happytime()` at happy moments, like a level completed.

C++

```
// Portal.h (declarations)
void startLevel();
void levelComplete();
void watchAdForCoins();

void onAdFinished(doriax::WebPortalAdType type);
void onAdError(doriax::WebPortalAdType type, std::string code, std::string message);

// Portal.cpp
Portal::Portal(Scene* scene, Entity entity): ScriptBase(scene, entity) {
    REGISTER_EVENT(WebPortal::onAdFinished, onAdFinished);
    REGISTER_EVENT(WebPortal::onAdError, onAdError);

    WebPortal::initialize();
    WebPortal::loadingStop();   // waits for the SDK
}

void Portal::startLevel() {
    WebPortal::gameplayStart();
}

void Portal::levelComplete() {
    WebPortal::happytime();
    WebPortal::gameplayStop();
    WebPortal::requestAd(WebPortalAdType::MIDGAME);   // the portal may show none
}

void Portal::watchAdForCoins() {
    WebPortal::requestAd(WebPortalAdType::REWARDED);
}

void Portal::onAdFinished(WebPortalAdType type) {
    if (type == WebPortalAdType::REWARDED) {
        // add the coins
    }
}

void Portal::onAdError(WebPortalAdType type, std::string code, std::string message) {
    if (type == WebPortalAdType::REWARDED) {
        // tell the player no ad could be shown
    }
}
```

Lua

```
local Portal = {}

function Portal:init()
    RegisterEvent(self, WebPortal.onAdFinished, "onAdFinished")
    RegisterEvent(self, WebPortal.onAdError, "onAdError")

    WebPortal.initialize()
```

```
    WebPortal.loadingStop()   -- waits for the SDK
end

function Portal:startLevel()
    WebPortal.gameplayStart()
end

function Portal:levelComplete()
    WebPortal.happytime()
    WebPortal.gameplayStop()
    WebPortal.requestAd(WebPortalAdType.MIDGAME)   -- the portal may show none
end

function Portal:watchAdForCoins()
    WebPortal.requestAd(WebPortalAdType.REWARDED)
end

function Portal:onAdFinished(adType)
    if adType == WebPortalAdType.REWARDED then
        -- add the coins
    end
end

function Portal:onAdError(adType, code, message)
    if adType == WebPortalAdType.REWARDED then
        -- tell the player no ad could be shown
    end
end

return Portal
```

Portals decide when a midgame ad really plays, so a request often shows none and reports `onAdError` with the code `unfilled`. The codes are:

Code	Meaning
`unfilled`	No ad played: the portal had none, chose not to show one now, or another ad is playing
`unavailable`	No portal SDK: a build without a portal, `initialize()` not called, or the portal refused to start here
`other`	The ad started but failed, or a rewarded ad was closed before its reward
A CrazyGames code	CrazyGames passes its own codes, like `adCooldown`

A rewarded ad grants its reward only through `onAdFinished`.

Environments

`onInitialized` and `getEnvironment()` report a `WebPortalEnvironment` (p. 629):

Value	Meaning
`UNINITIALIZED`	`initialize()` has not finished yet
`DISABLED`	No portal in this build, its SDK failed to load, or the portal does not run on this host (CrazyGames outside its site, Yandex Games outside its frame, YouTube outside Playables). Calls do nothing, and ads report `unavailable`
`LOCAL`	`localhost`, `127.0.0.1` or `[::1]`, where most portals show test ads
`PORTAL`	Any other host, normally the portal's

`isAvailable()` is true for `LOCAL` and `PORTAL`.

Pauses and mute

The engine pauses while one of its ads plays. GameDistribution, Yandex Games and YouTube Playables also pause the game themselves: for an ad they start on their own, a purchase window, or the player switching away. The engine stays paused until both the ad and the portal let it go, and `Engine::onResume` follows.

YouTube Playables can mute a game from its own interface. The engine follows it: audio stays silent while YouTube has it off, whatever the game's global volume (p. 591) is.

Cloud saves

YouTube Playables saves progress in the player's account. Call `loadData()` at startup and read the save in `onDataLoaded(data)`, which is empty when nothing was saved yet. `saveData(data)` stores one string, like a JSON document, and reports only failures, through `onDataSaveFailed`. YouTube refuses saves made before a `loadData()`, and its rules require progress to be saved this way.

Other portals have no cloud save, so `loadData()` reports `onDataLoadFailed`. Save there with `UserSettings` (p. 707), which the web stores in the browser.

What each portal supports

Call or event	CrazyGames	Poki	GameDistribution	Yandex Games	YouTube Playables
`requestAd(MIDGAME)`	Midgame ad	Commercial break	Interstitial	Fullscreen ad	Interstitial
`requestAd(REWARDED)`	Yes	Yes	Yes	Yes	Yes
`onAdStarted`	Yes	Yes	Yes	Yes	No: YouTube pauses the game instead
`gameplayStart` / `gameplayStop`	Yes	Yes	No	Yes	No
`loadingStart`	Yes	Yes	No	No	No
`loadingStop`	Yes	Yes	No	Required	Required
`happytime`	Yes	Yes	No	No	No
`loadData` / `saveData`	No	No	No	No	Yes
Pauses the game itself	No	No	Yes	Yes	Yes

A call a portal does not support does nothing.

Testing on your machine

Serve the exported folder from `localhost` (see Export Window → Web mode (p. 162)) and the portal SDKs act as follows:

- **CrazyGames** shows demo ads on `localhost`. On another host outside CrazyGames, add `?useLocalSdk=true` to the URL for the same.
- **Poki** runs in debug mode on `localhost`, with test ads.
- **GameDistribution** needs the **Portal Game ID** even locally. For test ads, set the `gd_debug_ex` and `gd_tag` keys of the page's `localStorage` to `true`.
- **Yandex Games** loads its SDK from `/sdk.js`, which Yandex's hosting serves; on other hosts the portal is `DISABLED`. Yandex's `@yandex-games/sdk-dev-proxy`[21] serves a mock SDK with placeholder ads (run it with `--dev-mode=true`).
- **YouTube Playables** runs only inside YouTube, and is `DISABLED` anywhere else. The export adds YouTube's SDK `<script>` to the page head, which YouTube requires to load before the game. Test it with YouTube's Playables test suite.

Migrating from the System methods

Earlier versions had AdMob and CrazyGames methods on `System` (p. 407). They were replaced by the classes on this page:

21. https://www.npmjs.com/package/@yandex-games/sdk-dev-proxy

Before	Now
`System::initializeAdMob(childDirected, underAge)`	`AdMob::setAgeRestriction(AdMobAgeRestriction::CHILD)` for a children's game, then `AdMob::requestConsent()` and `AdMob::initialize()`, as in Consent, then initialize
`System::setMaxAdContentRating(AdMobRating::General)`	`AdMob::setMaxAdContentRating(AdMobRating::GENERAL)`. The values are upper case now, with `UNSPECIFIED` added
`System::loadInterstitialAd(id)`, `isInterstitialAdLoaded()`, `showInterstitialAd()`	`AdMob::loadInterstitialAd(id)`, `AdMob::isInterstitialAdLoaded()`, `AdMob::showInterstitialAd()`
`System::initializeCrazyGamesSDK()`	**Game Portal** set to CrazyGames, then `WebPortal::initialize()`
`System::showCrazyGamesAd("midgame")`, `("rewarded")`	`WebPortal::requestAd(WebPortalAdType::MIDGAME)`, `(WebPortalAdType::REWARDED)`
`System::happytimeCrazyGames()`	`WebPortal::happytime()`
`System::gameplayStartCrazyGames()`, `gameplayStopCrazyGames()`	`WebPortal::gameplayStart()`, `WebPortal::gameplayStop()`
`System::loadingStartCrazyGames()`, `loadingStopCrazyGames()`	`WebPortal::loadingStart()`, `WebPortal::loadingStop()`

Lua scripts make the same change: `System.showInterstitialAd()` becomes `AdMob.showInterstitialAd()`. Ads now also need **Google AdMob** or **Game Portal** in the project settings, since exports only include the SDKs that are enabled.

Reference

- AdMob (p. 610)
- InAppPurchase (p. 618)
- WebPortal (p. 626)
- Project Settings → Platforms (p. 60)

PART 5

Reference

API Index 352
Enumerations 355
Build Options 357
Core 361
Objects 413
Physics 541
Actions 569
Systems 590
Managers 597
Services 610
Scripting 630
Math 632
Components 666
I/O 703
Other 708
Particles 720
Textures 733

API Reference

Each class page documents the **C++ and Lua** API together, in the style of the Supernova Engine docs[22].

Pages with detailed method descriptions: **Engine**, **Scene**, **Object**, **Input**, **Camera**, **Body2D**, **PhysicsSystem**, and related core types. Other classes list the full API surface; expand descriptions over time by editing the markdown files under `docs/reference/classes/`.

Use `doriax/generate_api_suggestions.py` in the engine repo to inventory bindings when adding new APIs.

See also Enumerations (p. 355) and Build Options (p. 357).

Actions

- Action (p. 569)
- AlphaAction (p. 572)
- Animation (p. 573)
- ColorAction (p. 577)
- KeyframeTracks (p. 578)
- MorphTracks (p. 579)
- PositionAction (p. 580)
- RotateTracks (p. 581)
- RotationAction (p. 582)
- ScaleAction (p. 584)
- ScaleTracks (p. 585)
- SpriteAnimation (p. 586)
- TimedAction (p. 588)
- TranslateTracks (p. 590)

Components

- ActionComponent (p. 666)
- AlphaActionComponent (p. 666)
- AnimationComponent (p. 667)
- Body2DComponent (p. 667)
- Body3DComponent (p. 668)
- BoneComponent (p. 668)
- ButtonComponent (p. 669)
- CameraComponent (p. 670)
- ColorActionComponent (p. 671)
- FogComponent (p. 671)
- ImageComponent (p. 672)
- InstancedMeshComponent (p. 672)
- Joint2DComponent (p. 673)
- Joint3DComponent (p. 674)
- KeyframeTracksComponent (p. 675)
- Light2DComponent (p. 676)
- LightComponent (p. 677)
- LinesComponent (p. 678)
- MeshComponent (p. 679)
- MeshPolygonComponent (p. 680)
- MirrorComponent (p. 680)
- ModelComponent (p. 681)
- MorphTracksComponent (p. 681)
- Occluder2DComponent (p. 682)
- PanelComponent (p. 682)
- ParticlesComponent (p. 683)
- PointsComponent (p. 684)
- PolygonComponent (p. 684)
- PositionActionComponent (p. 685)
- ProgressbarComponent (p. 685)
- ReflectionProbeComponent (p. 686)
- RotateTracksComponent (p. 686)
- RotationActionComponent (p. 687)
- ScaleActionComponent (p. 687)
- ScaleTracksComponent (p. 687)
- ScrollbarComponent (p. 688)
- SkyComponent (p. 688)
- SoundComponent (p. 689)
- SpriteAnimationComponent (p. 690)
- SpriteComponent (p. 691)
- TerrainComponent (p. 692)
- TextComponent (p. 693)
- TextEditComponent (p. 694)
- TilemapComponent (p. 695)
- TimedActionComponent (p. 695)
- Transform (p. 696)
- TranslateTracksComponent (p. 697)
- UIComponent (p. 698)
- UIContainerComponent (p. 699)
- UILayoutComponent (p. 700)
- WaterComponent (p. 701)
- WaterExclusionComponent (p. 702)

Core

- Engine (p. 361)
- EntityManager (p. 378)
- EntityRegistry (p. 379)
- Input (p. 380)
- Log (p. 387)
- Scene (p. 390)
- System (p. 407)

I/O

- Data (p. 703)
- File (p. 705)
- FileData (p. 706)
- UserSettings (p. 707)

22. https://github.com/supernovaengine/supernova-docs

Managers

- BundleManager (p. 597)
- SceneManager (p. 601)

Math

- AABB (p. 632)
- Angle (p. 636)
- Matrix3 (p. 637)
- Matrix4 (p. 638)
- OBB (p. 639)
- Plane (p. 642)
- Quaternion (p. 644)
- Ray (p. 649)
- Rect (p. 652)
- Sphere (p. 654)
- Vector2 (p. 656)
- Vector3 (p. 660)
- Vector4 (p. 664)

Objects

- Bone (p. 413)
- Camera (p. 414)
- Fog (p. 422)
- Light (p. 424)
- Light2D (p. 430)
- Lines (p. 433)
- Mesh (p. 436)
- Mirror (p. 448)
- Model (p. 450)
- Object (p. 455)
- Occluder2D (p. 462)
- Points (p. 464)
- ReflectionProbe (p. 468)
- Shape (p. 471)
- SkyBox (p. 472)
- Sound (p. 475)
- Sprite (p. 480)
- Terrain (p. 486)
- Tilemap (p. 492)
- Water (p. 496)
- WaterExclusion (p. 503)

UI

- Button (p. 506)
- Container (p. 510)
- Image (p. 513)
- Panel (p. 518)
- Polygon (p. 521)
- Progressbar (p. 523)
- Scrollbar (p. 525)
- Text (p. 528)
- TextEdit (p. 533)
- UILayout (p. 537)

Other

- ActionFrame (p. 708)
- Base64 (p. 708)
- Color (p. 709)
- EntityHandle (p. 711)
- FramebufferRender (p. 712)
- InstanceData (p. 713)
- LineData (p. 713)
- MeshPolygon (p. 713)
- OverallBuildProgress (p. 714)
- PointData (p. 714)
- PolygonPoint (p. 714)
- PostProcessPass (p. 715)
- RayReturn (p. 716)
- ResourceBuildInfo (p. 717)
- ResourceProgress (p. 717)
- TerrainFoliageLayer (p. 718)
- TextureLoadResult (p. 718)
- TextureRender (p. 718)
- ThreadPoolManager (p. 719)
- TileData (p. 719)
- TileRectData (p. 719)

Particles

- ParticleAccelerationInitializer (p. 720)
- ParticleAccelerationModifier (p. 720)
- ParticleAlphaInitializer (p. 720)
- ParticleAlphaModifier (p. 720)
- ParticleBurst (p. 721)
- ParticleColorGradient (p. 721)
- ParticleColorGradientStop (p. 721)
- ParticleColorInitializer (p. 721)
- ParticleColorModifier (p. 722)
- ParticleLifeInitializer (p. 722)
- ParticlePositionInitializer (p. 722)
- ParticlePositionModifier (p. 723)
- ParticleRotationInitializer (p. 723)
- ParticleRotationModifier (p. 723)
- Particles (p. 724)
- ParticleScaleInitializer (p. 731)
- ParticleScaleModifier (p. 731)
- ParticleSizeInitializer (p. 731)
- ParticleSizeModifier (p. 732)
- ParticleSpriteInitializer (p. 732)
- ParticleSpriteModifier (p. 732)
- ParticleVelocityInitializer (p. 732)
- ParticleVelocityModifier (p. 733)

Physics

- Body2D (p. 541)
- Body3D (p. 551)
- CollideShapeResult3D (p. 561)
- Contact2D (p. 562)
- Contact3D (p. 562)
- Joint2D (p. 563)
- Joint3D (p. 564)
- Manifold2D (p. 566)
- PhysicsSystem (p. 567)

Scripting

- ScriptBase (p. 630)

Services

- AdMob (p. 610)
- InAppPurchase (p. 618)
  - PricingPhase (p. 621)
  - ProductOffer (p. 622)
  - ProductDetails (p. 623)
  - PurchaseDetails (p. 623)
- WebPortal (p. 626)

Systems

- ActionSystem (p. 590)
- AudioSystem (p. 591)
- MeshSystem (p. 592)
- RenderSystem (p. 594)
- SubSystem (p. 594)
- UISystem (p. 595)

Textures

- Framebuffer (p. 733)
- Material (p. 734)
- Texture (p. 738)
- TextureData (p. 744)

Enumerations

Most enums are exposed as Lua namespaces (`Scaling.FITWIDTH`) and C++ `enum class` types. C++-only enums are noted below. See Engine (p. 361) for `Scaling`, `Platform`, and `GraphicBackend`, and AdMob (p. 616), InAppPurchase (p. 624) and WebPortal (p. 628) for the values of their enums.

Enum	Typical use
`Scaling`	Canvas scaling mode
`TextureStrategy`	Non-power-of-two texture handling
`Platform`	Runtime OS
`GraphicBackend`	Graphics API
`BodyType`	`STATIC`, `KINEMATIC`, `DYNAMIC`
`CameraType`	UI, orthographic, perspective
`LightType`	Directional, point, spot
`Occluder2DShape`	2D shadow occluder outline (`AUTO_QUAD`, `POLYGON`)
`ShadowQuality`	Shadow PCF filter, 3D and 2D (`NONE`, `LOW`, `MEDIUM`, `HIGH`)
`Shape2DType` / `Shape3DType`	Physics shapes
`Shape3DBuildQuality`	Mesh shape collision tree (`FAVOR_RUNTIME_PERFORMANCE`, `FAVOR_BUILD_SPEED`)
`Joint2DType` / `Joint3DType`	Constraints
`Joint3DMotorState`	Hinge and prismatic motor (`OFF`, `VELOCITY`, `POSITION`, `POSITION_AND_VELOCITY`)
`AnchorPreset` / `PivotPreset`	UI layout
`EaseType`	Animation easing
`SoundState` / `SoundAttenuation`	Audio
`ActionState`	Action playback
`FogType`	Fog mode
`ReflectionProbeMode`	Reflection probe source (`STATIC`, `DYNAMIC`)
`ReflectionProbeUpdateMode`	Dynamic probe capture policy (`ON_LOAD`, `ON_MOVE`, `INTERVAL`, `MANUAL`)
`WaterExclusionShape`	Volume keeping water out (`BOX`, `SPHERE`, `HULL`)
`LightState` / `UIEventState`	Scene flags
`ShaderType`	Scene default shader types for Scene.setDefaultShaderUniform (p. 405) (`MESH`, `UI`, `SKYBOX`, `POINTS`, `LINES`, `WATER`)
`PrimitiveType` / `CullingMode` / `WindingOrder`	Rendering
`MaterialAlphaMode`	Material alpha handling (`AUTO`, `ALPHA_OPAQUE`, `MASK`, `BLEND`)
`TextureFilter` / `TextureWrap` / `TextureType`	Textures
`CursorType`	Mouse cursor
`NavigationDirection`	Keyboard and gamepad UI focus
`ResourceLoadState`	Async loading
`AdMobFormat` / `AdMobBannerSize` / `AdMobBannerPosition`	AdMob ad formats and banner layout
`AdMobRating` / `AdMobAgeRestriction` / `AdMobPersonalization`	AdMob request settings
`AdMobConsentStatus` / `AdMobDebugGeography` / `AdMobPrecision`	AdMob consent and paid-event precision
`ProductType` / `PurchaseState` / `BillingResponse`	In-app purchases
`RecurrenceMode` / `SubscriptionReplacementMode`	Subscription pricing and plan changes

Enum	Typical use
`WebPortalType` / `WebPortalAdType` / `WebPortalEnvironment`	Web game portals

Run `doriax/generate_api_suggestions.py` against the engine binding folder for the authoritative member list when documenting new enum values.

Build Options

Doriax uses CMake for desktop editor/runtime builds and platform tooling for mobile, Apple, and web exports.

Common requirements

Requirement	Notes
CMake	3.20 or newer
C++ compiler	MSVC, GCC, Clang, or equivalent, capable of the selected `DORIAX_CXX_STANDARD` (C++17 by default)
Python 3	Used by code generation scripts
Platform SDK	Required for Android, iOS, macOS, and web builds

Editor build

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release --target doriax-editor
cmake --install build --config Release --strip
```

`DORIAX_CXX_STANDARD` applies here too, building the editor and engine with it. It is independent of the standard a project selects: the editor compiles script plugins with the project's value regardless of how the editor itself was built.

Leave `DORIAX_PHYSICS_2D` and `DORIAX_PHYSICS_3D` on for the editor: it authors every component, and Play links against its engine library, so the editor needs both backends. They are meant for runtime builds.

On Windows the build also provides `doriax-editor-cmd`, the same editor linked as a console application for automation and terminal output. It is excluded from the default build, so request it by name:

```
cmake --build build --config Release --target doriax-editor-cmd
```

See Command-Line Tools (p. 167).

Runtime build

```
cmake -S engine -B build -DCMAKE_BUILD_TYPE=Debug
cmake --build build --config Debug
```

Scripting entry options

Option	Effect
`NO_CPP_INIT`	Disable the C++ startup entry point
`NO_LUA_INIT`	Disable the Lua startup entry point

Engine/library options

Option	Effect
`DORIAX_SHARED`	Build the engine as a shared library when enabled
`ENABLE_OPT`	Enable SPIR-V optimizer support when available
`EMSCRIPTEN_THREAD_SUPPORT`	Enable pthread support for Emscripten builds

Runtime project options

Option	Default	Effect
`DORIAX_VSYNC_ENABLED`	`ON`	Synchronize supported desktop presentation backends to the display refresh rate. Set to `OFF` to request an uncapped build. macOS Metal remains synchronized.
`DORIAX_WINDOW_WIDTH`	`960`	Initial window width in pixels for desktop builds
`DORIAX_WINDOW_HEIGHT`	`540`	Initial window height in pixels for desktop builds
`DORIAX_WINDOW_MODE`	`0`	Initial window state: `0` windowed, `1` maximized, `2` fullscreen. The Sokol app backend has no maximized support and treats `1` as windowed.
`DORIAX_WINDOW_RESIZABLE`	`ON`	Allow resizing the window. Ignored by the Sokol app backend, whose windows are always resizable.
`DORIAX_WINDOW_TITLE`	`"Doriax"`	Window title-bar text
`DORIAX_CXX_STANDARD`	`17`	C++ standard for the engine and the game target. Accepts `17`, `20` or `23` only — configuration fails on any other value.
`DORIAX_PHYSICS_2D`	`ON`	Build and link Box2D. `OFF` leaves out 2D bodies, joints and their Lua API. When on, the engine target also defines the `DORIAX_PHYSICS_2D` macro for the game.
`DORIAX_PHYSICS_3D`	`ON`	Build and link Jolt. `OFF` leaves out 3D bodies, joints and their Lua API. When on, the engine target also defines the `DORIAX_PHYSICS_3D` macro for the game.
`DORIAX_ADMOB`	`ON`	iOS only: link Google Mobile Ads and compile the AdMob adapter, defining the `DORIAX_ADMOB` macro. `OFF` leaves `AdMob` (p. 610) unavailable.
`DORIAX_STOREKIT`	`ON`	iOS only: compile the StoreKit 2 adapter, a Swift source that needs Swift 6.1 (Xcode 16.3 or later), defining the `DORIAX_STOREKIT` macro. `OFF` leaves `InAppPurchase` (p. 618) unavailable on iOS.
`DORIAX_WEB_PORTAL`	`none`	Web only: the game portal SDK built in for `WebPortal` (p. 626): `none`, `crazygames`, `poki`, `gamedistribution`, `yandex` or `youtube`. Any other value fails configuration. A portal also defines `DORIAX_WEB_PORTAL_<NAME>`, like `DORIAX_WEB_PORTAL_POKI`.
`DORIAX_WEB_PORTAL_GAME_ID`	`""`	Web only: the game ID of portals that need one, which is GameDistribution.

The editor writes these options into exported CMake projects from **Project Settings** (VSync, the Window settings, the C++ standard, the physics backends, iOS **Google AdMob** and **App Store Purchases**, and the web **Game Portal**). For a manually configured standalone runtime, override them at configure time:

```
cmake -S engine -B build-runtime \
  -DPROJECT_ROOT=/path/to/project \
  -DDORIAX_VSYNC_ENABLED=OFF \
  -DDORIAX_WINDOW_MODE=2 \
  -DDORIAX_WINDOW_TITLE="My Game" \
  -DDORIAX_CXX_STANDARD=23
```

Vulkan treats `OFF` as a request for Immediate presentation, with Mailbox and then FIFO as fallbacks according to surface support.

Graphics backend

Set `GRAPHIC_BACKEND` when configuring a standalone runtime project. The editor's Desktop export selector passes the corresponding value automatically; Web exports use `gles3`.

Option value	Target
`glcore`	Desktop OpenGL Core
`gles3`	OpenGL ES 3 / WebGL-style targets
`d3d11`	Windows Direct3D 11
`metal`	macOS and iOS Metal
`vulkan`	Vulkan (Linux and Windows)

On macOS, `vulkan` builds through MoltenVK but cannot run yet — see Vulkan backend (p. 755).

The root **editor** build reads the same variable, but accepts only `vulkan` or `glcore` (Linux and Windows, defaulting to `glcore`); macOS is forced to `metal`. There it is a cached variable, so the build directory keeps the backend it was configured with until `-DGRAPHIC_BACKEND=` is passed again.

App backend

The target system selects the app backend, which handles the window, input, and main loop. Configure prints it as `APP_BACKEND`, but the value is informational: setting it has no effect.

Target system	APP_BACKEND	Platform code
Windows	`win`	`platform/win`: native Win32 window with a WGL (OpenGL), Vulkan, or D3D11 context
Linux, FreeBSD	`linux`	`platform/linux`: native X11 window with a GLX (OpenGL) or Vulkan context
macOS	`apple`	`platform/mac`: native Cocoa executable. With the Xcode generator and Metal, `platform/apple` builds an `.app` bundle instead
iOS	`apple`	`platform/apple`
Android	`android`	`platform/android`: GameActivity
Emscripten	`emscripten`	`platform/emscripten`

Generated artifacts

Artifact	Generated from
`shaders.h`	Shader sources processed by CMake shader generation
`engine_api_suggestions.h`	Lua binding files parsed by `generate_api_suggestions.py`
Copied engine directory	Editor export/build support

Engine capacity macros

Exported CMake projects and their Xcode workspace receive compiler defines for the largest capacities the editor measured in the project's scenes and bundles, each floored at the engine default in `engine/core/Engine.h`. Growing past the default lets large authored content compile; the floor keeps dynamically spawned content inside the known-safe baseline.

Macro	Default	Controls
`MAX_SUBMESHES`	4	Submeshes per mesh
`MAX_TILEMAP_TILESRECT`	64	Tilemap rectangle tiles
`MAX_TILEMAP_TILES`	512	Tiles per tilemap
`MAX_SPRITE_FRAMES`	32	Atlas frames per sprite or `Points` object, and frames per sprite animation
`MAX_EXTERNAL_BUFFERS`	20	External vertex/index buffers per mesh
`MAX_BONES`	96	Bone matrices per skinned mesh, held in its `SkinComponent`
`MAX_SHAPES_2D`	4	Shapes per 2D body
`MAX_SHAPE_POINTS_2D`	8	Points per 2D polygon or chain shape
`MAX_SHAPES_3D`	4	Shapes per 3D body
`MAX_SHAPE_VERTICES_3D`	64	Vertices per 3D convex hull or mesh shape
`MAX_SHAPE_INDICES_3D`	192	Indices per 3D mesh shape

`MAX_EXTERNAL_BUFFERS` is counted from the glTF files the models and foliage layers use (an FBX through its imported `.glb`), including the buffers the loader generates, so it is right even in a command line (p. 167) export that never loads them.

`MAX_BONES` is the size of the matrix array in each skinned mesh's `SkinComponent`; meshes without a skin don't carry it. Skinned shaders bind those matrices through a storage buffer or a bone texture and do not embed this count. The editor can load skins larger than 96; export writes the raised value so the standalone build matches. If you load a larger skin only from code, edit the generated `add_definitions("-DMAX_BONES=N")` line in the exported CMakeLists, or `MAX_BONES` in the project-level **Preprocessor Macros** of the exported Xcode workspace (or pass the same compiler define in a standalone engine build). The engine and the game must be built with the same value.

Export measures only the macros above. These other limits keep their engine values in every build:

Macro	Value	Controls
`MAX_MESH_LODS`	4	Detail levels per submesh, the source plus three simplified ones (see Mesh detail (p. 275))
`MAX_MORPHTARGETS`	8	Morph targets per mesh, fewer when they carry normals or tangents (see GLTF limits (p. 249))
`MAX_TEXT_FONTS`	4	Font slots per text, the main font and three fallbacks
`MAX_BROADPHASELAYER_3D`	6	3D broad-phase layers
`MAX_TERRAINGRID`	16	Root nodes per terrain, so `rootGridSize` goes up to 4
`MAX_CONTAINER_BOXES`	30	Children laid out by one UI container
`MAX_WATER_EXCLUSIONS`	8	Water exclusions per water draw, the ones nearest the camera

The light and shadow limits are listed under Shadow maps (p. 260) and 2D lighting and shadows (p. 261) in Rendering Pipeline. The editor compiles them into the shaders it builds, so an exported project can't change them.

Platform guides

See Building Overview (p. 746), then use the guide for your target platform.

Core

Engine

Description

Control engine properties and define defaults used across the whole project. `Engine` is entirely static — never instantiated. It manages the scene stack, canvas size and scaling, the update loop, platform queries, and all frame/input callback events.

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Scene scene;

DORIAX_INIT void init() {
    Engine::setCanvasSize(1000, 480);
    Engine::setScalingMode(Scaling::FITWIDTH);
    Engine::setScene(&scene);
}
```

Lua

```lua
scene = Scene()
Engine.setCanvasSize(1000, 480)
Engine.scalingMode = Scaling.FITWIDTH
Engine.setScene(scene)
```

Properties

Type	Name	Default	Languages
static Scaling	scalingMode	`FITWIDTH`	C++ | Lua
static TextureStrategy	textureStrategy	`NONE`	C++ | Lua
static bool	callMouseInTouchEvent	`false`	C++ | Lua
static bool	callTouchInMouseEvent	`false`	C++ | Lua
static bool	useDegrees	`true`	C++ | Lua
static bool	allowEventsOutCanvas	`false`	C++ | Lua
static bool	ignoreEventsHandledByUI	`true`	C++ | Lua
static float	updateTime	`0.03`	C++ | Lua
static double	interpolationAlpha	—	C++ | Lua
static int	canvasWidth	—	C++ | Lua
static int	canvasHeight	—	C++ | Lua
static int	preferredCanvasWidth	—	C++ | Lua
static int	preferredCanvasHeight	—	C++ | Lua
static Rect	viewRect	—	C++ | Lua
static float	deltatime	—	C++ | Lua
static float	maxDeltatime	`0.25`	C++ | Lua
static float	framerate	—	C++ | Lua
static FrameStats	frameStats	—	C++
static double	systemTime	—	C++ | Lua
static Platform	platform	—	C++ | Lua

Type	Name	Default	Languages
static GraphicBackend	graphicBackend	—	C++ | Lua
static bool	openGL	—	C++ | Lua
static bool	asyncLoading	`false`	C++ | Lua
static CursorType	mouseCursor	`ARROW`	C++ | Lua
static MouseMode	mouseMode	`NORMAL`	C++ | Lua
static Framebuffer*	framebuffer	—	C++ | Lua

Methods

Returns	Name	Languages
static void	setScene	C++ | Lua
static Scene*	getScene	C++ | Lua
static void	addSceneLayer	C++ | Lua
static void	executeSceneOnce	C++ | Lua
static void	removeScene	C++ | Lua
static void	removeAllSceneLayers	C++ | Lua
static void	removeAllScenes	C++ | Lua
static bool	isSceneRunning	C++ | Lua
static bool	hasScenesToExecuteOnce	C++ | Lua
static Scene*	getMainScene	C++ | Lua
static Scene*	getLastScene	C++ | Lua
static void	pauseGameEvents	C++ | Lua
static void	setCanvasSize	C++ | Lua
static Rect	getViewRect	C++
static void	setMousePosition	C++ | Lua
static void	setUpdateTimeMS	C++ | Lua
static bool	isUIEventReceived	C++ | Lua
static bool	isViewLoaded	C++ | Lua
static const FrameStats&	getFrameStats	C++
static void	setMaxResourceLoadingThreads	C++ | Lua
static size_t	getQueuedResourceCount	C++ | Lua
static void	clearPools	C++
static void	clearUnusedPools	C++
static void	clearAllSubscriptions	C++ | Lua
static void	clearComponentSubscriptions	C++ | Lua
static void	startAsyncThread	C++ | Lua
static void	commitThreadQueue	C++ | Lua
static void	endAsyncThread	C++ | Lua
static bool	isAsyncThread	C++ | Lua

Callback events

Callback	Name	Languages
void()	onViewLoaded	C++ | Lua
void()	onCanvasChanged	C++ | Lua
void()	onViewDestroyed	C++ | Lua

Callback	Name	Languages
void()	onDraw	C++ | Lua
void()	onUpdate	C++ | Lua
void()	onFixedUpdate	C++ | Lua
void()	onPostUpdate	C++ | Lua
void()	onPause	C++ | Lua
void()	onResume	C++ | Lua
void()	onShutdown	C++ | Lua
void()	onSceneLoaded	C++ | Lua
void(int,float,float)	onTouchStart	C++ | Lua
void(int,float,float)	onTouchEnd	C++ | Lua
void(int,float,float)	onTouchMove	C++ | Lua
void(int,float,float)	onTouchCancel	C++ | Lua
void(int,float,float,int)	onMouseDown	C++ | Lua
void(int,float,float,int)	onMouseUp	C++ | Lua
void(float,float,int)	onMouseMove	C++ | Lua
void(float,float,int)	onMouseScroll	C++ | Lua
void()	onMouseEnter	C++ | Lua
void()	onMouseLeave	C++ | Lua
void(int,bool,int)	onKeyDown	C++ | Lua
void(int,bool,int)	onKeyUp	C++ | Lua
void(wchar_t)	onCharInput	C++ | Lua
void(int)	onGamepadConnect	C++ | Lua
void(int)	onGamepadDisconnect	C++ | Lua
void(int,int)	onGamepadButtonDown	C++ | Lua
void(int,int)	onGamepadButtonUp	C++ | Lua
void(int,int,float)	onGamepadAxisMove	C++ | Lua

Enumerations

Scaling

- **FITWIDTH** — Keeps canvas width fixed; height floats to match aspect ratio.
- **FITHEIGHT** — Keeps canvas height fixed; width floats.
- **LETTERBOX** — Keeps both dimensions; empty bars appear on screen edges.
- **CROP** — Keeps both dimensions; parts of the canvas may be cropped off screen.
- **STRETCH** — Stretches to fill the screen; objects may deform.
- **NATIVE** — Uses the native window resolution as-is.

TextureStrategy

- **FIT** — Pads non-power-of-two textures to the next power of two, without scaling them.
- **RESIZE** — Scales textures to the next power of two. The resampling blurs the edges of atlas tiles, which can show as seams in tilemaps.
- **NONE** — Keeps textures at their size. Every supported backend handles non-power-of-two textures.

Platform

- **MacOS** — Running on macOS.
- **iOS** — Running on iOS.
- **Web** — Running in WebGL/HTML5.
- **Android** — Running on Android.
- **Linux** — Running on Linux.
- **Windows** — Running on Windows.

GraphicBackend

- **GLCORE** — OpenGL Core (desktop).
- **GLES3** — OpenGL ES 3 (mobile/web).
- **D3D11** — Direct3D 11 (Windows).
- **METAL** — Metal (Apple platforms).
- **WGPU** — WebGPU.
- **VULKAN** — Vulkan (Linux and Windows; macOS builds through MoltenVK).

CursorType

`ARROW`, `IBEAM`, `CROSSHAIR`, `POINTING_HAND`, `RESIZE_EW`, `RESIZE_NS`, `RESIZE_NWSE`, `RESIZE_NESW`, `RESIZE_ALL`, `NOT_ALLOWED`

MouseMode

Value	Cursor	Movement
`NORMAL`	visible	moves freely (default)
`HIDDEN`	hidden	moves freely; absolute position still reported
`CAPTURED`	hidden	locked to the window; relative motion only (mouse-look)
`CONFINED`	visible	cannot leave the window bounds

NavigationDirection

`NONE`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `NEXT`, `PREVIOUS`

Where UISystem.eventOnNavigate (p. 595) moves the button focus. `NEXT` and `PREVIOUS` follow reading order, like Tab and Shift+Tab.

Property details

scalingMode

- *Setter:* `static void setScalingMode(Scaling scalingMode)`
- *Getter:* `static Scaling getScalingMode()`

Controls how the logical canvas is mapped to the physical window. See Scaling for each mode. Like setCanvasSize, a change applies immediately to the running scenes.

textureStrategy

- *Setter:* `static void setTextureStrategy(TextureStrategy textureStrategy)`
- *Getter:* `static TextureStrategy getTextureStrategy()`

Controls automatic resizing of non-power-of-two textures on hardware that requires it. It applies to textures loaded after it is set.

callMouseInTouchEvent

- *Setter:* `static void setCallMouseInTouchEvent(bool callMouseInTouchEvent)`
- *Getter:* `static bool isCallMouseInTouchEvent()`

When `true`, receiving a touch event also fires the equivalent mouse event. Useful for cross-platform testing.

callTouchInMouseEvent

- *Setter:* `static void setCallTouchInMouseEvent(bool callTouchInMouseEvent)`
- *Getter:* `static bool isCallTouchInMouseEvent()`

When `true`, receiving a mouse event also fires the equivalent touch event.

useDegrees

- *Setter:* `static void setUseDegrees(bool useDegrees)`
- *Getter:* `static bool isUseDegrees()`

When `true` (default), angle parameters in the API expect degrees. Set to `false` to use radians throughout.

allowEventsOutCanvas

- *Setter:* `static void setAllowEventsOutCanvas(bool allowEventsOutCanvas)`
- *Getter:* `static bool isAllowEventsOutCanvas()`

When `true`, mouse move and scroll events are delivered even when the cursor is outside the canvas bounds.

ignoreEventsHandledByUI

- *Setter:* `static void setIgnoreEventsHandledByUI(bool ignoreEventsHandledByUI)`
- *Getter:* `static bool isIgnoreEventsHandledByUI()`

When `true`, gameplay input callbacks are skipped for events already consumed by a UI widget. That covers pointer and touch events, and the keys, gamepad buttons, and left stick that move and press the focused button (p. 295).

updateTime

- *Setter:* `static void setUpdateTime(float updateTime)`
- *Getter:* `static float getUpdateTime()`

Fixed-update interval in **seconds**. Used by `onFixedUpdate` and the physics simulation step. Also accepted in milliseconds via setUpdateTimeMS.

interpolationAlpha

- *Getter:* `static double getInterpolationAlpha()`

Value in `[0, 1)` representing how far the current rendered frame is between the previous and next fixed-update steps. Physics bodies are already drawn with it (see PhysicsSystem interpolation (p. 567)); use it to interpolate anything else you move from `onFixedUpdate`.

canvasWidth / canvasHeight

- *Getter:* `static int getCanvasWidth()` / `static int getCanvasHeight()`

Logical canvas dimensions after scaling is applied. These may differ from preferredCanvasWidth when a non-NATIVE scaling mode is active.

---

preferredCanvasWidth / preferredCanvasHeight

- *Getter:* `static int getPreferredCanvasWidth()` / `static int getPreferredCanvasHeight()`

The dimensions originally passed to setCanvasSize, unaffected by scaling.

---

viewRect

- *Getter:* `static Rect getViewRect()`

Viewport rectangle computed from canvas size, screen size, and scaling mode.

---

deltatime

- *Getter:* `static float getDeltatime()`

Time elapsed in **seconds** since the last draw frame. Multiply movement speeds by `deltatime` inside `onUpdate` to make motion frame-rate independent.

The value is **clamped** to maxDeltatime. After a long stall — the first frame of a scene (which includes load time), a debugger break, or an alt-tab — the real elapsed time can be several seconds. Returning that unclamped would teleport anything driven by `deltatime` in a single step (and could trigger a physics "spiral of death"). Clamping makes the simulation skip the lost time instead. This matches Unity's `Time.deltaTime` / `Time.maximumDeltaTime` and Godot's max-step behaviour.

> **Tip:** Even with clamping, prefer `moveTowards` (p. 662) over `direction * speed * deltatime` for "move to a target" logic. It clamps the step to the remaining distance, so it can never overshoot, whatever the frame time.

---

maxDeltatime

- *Getter:* `static float getMaxDeltatime()`
- *Setter:* `static void setMaxDeltatime(float seconds)`
- *Default:* `0.25`

Upper bound, in **seconds**, for the value returned by deltatime and used by the update loop. Lower it for stricter clamping; raise it to allow larger single steps. Must be greater than `0`. Equivalent to Unity's `Time.maximumDeltaTime`.

---

framerate

- *Getter:* `static float getFramerate()`

Current frames-per-second estimate. Computed from the **raw** (unclamped) frame time, so it stays accurate even on stalled frames where deltatime is clamped.

---

frameStats

- *Getter:* `static const FrameStats& getFrameStats()`

Draw counters for the last completed `systemDraw`: GPU submissions, not CPU work. C++ only — Lua can read framerate but not these fields.

Field	Type	Meaning
`drawCalls`	`uint32_t`	`draw` submissions issued that frame
`instances`	`uint32_t`	Sum of the instance counts passed to those draws
`triangles`	`uint64_t`	Triangle estimate from the primitive type and the index/vertex count, multiplied by the instance count (`1` when the draw is not instanced)

The editor footer shows `drawCalls` and `triangles` next to FPS. The `benchmark` (p. 169) command averages the same counters over a measured run.

C++

```
const Engine::FrameStats& stats = Engine::getFrameStats();
Log::info("draws %u  tris %llu", stats.drawCalls,
          (unsigned long long)stats.triangles);
```

systemTime

- *Getter:* `static double getSystemTime()`

Monotonic wall-clock time in **seconds**, independent of scene pausing and update loop throttling.

platform

- *Getter:* `static Platform getPlatform()`

Returns the Platform the engine is currently running on.

graphicBackend

- *Getter:* `static GraphicBackend getGraphicBackend()`

Returns the active GraphicBackend.

openGL

- *Getter:* `static bool isOpenGL()`

Returns `true` when any OpenGL backend (GLCORE, GLES3) is active.

asyncLoading

- *Setter:* `static void setAsyncLoading(bool enable)`
- *Getter:* `static bool isAsyncLoading()`

Enable background resource loading. When active, GPU resource creation must be committed on the main thread via commitThreadQueue.

The editor always loads asynchronously, Play included, while an exported game starts with it off. Turn it on from a script so both behave the same and a loading screen (p. 605) keeps animating.

mouseCursor

- *Setter:* `static void setMouseCursor(CursorType type)`
- *Getter:* `static CursorType getMouseCursor()`

Changes the OS mouse cursor shape. See CursorType.

mouseMode

- *Setter:* `static void setMouseMode(MouseMode mode)`
- *Getter:* `static MouseMode getMouseMode()`

Sets how the mouse cursor behaves — a single state that covers both its visibility and whether it is free, locked, or confined. See MouseMode for every value.

Use `CAPTURED` for first-person cameras, free-look controls, and any interaction that uses mouse deltas instead of an absolute pointer: the pointer is hidden and locked to the window, so it cannot escape while moving. While captured, mouse move events keep firing even when the virtual mouse position moves outside the canvas bounds. Use `CONFINED` to keep a visible cursor from leaving the window without switching to relative motion.

Lua

```
Engine.mouseMode = MouseMode.CAPTURED
```

C++

```
Engine::setMouseMode(MouseMode::CAPTURED);
```

> **Note**
> `CONFINED` needs cursor confinement from the platform (desktop GLFW 3.4+); where it isn't available it falls back to `NORMAL`. Browser builds implement `CAPTURED` with the Pointer Lock API, so it may only take effect after a user gesture such as a click.

framebuffer

- *Setter:* `static void setFramebuffer(Framebuffer* framebuffer)`
- *Getter:* `static Framebuffer* getFramebuffer()`

Off-screen render target for the full frame output. When set, the engine renders to this framebuffer instead of the window surface.

Method details

setScene

- `static void setScene(Scene* scene)`

Sets the main scene. Pass `nullptr` (C++) or `nil` (Lua) to clear the main scene.

getScene

- `static Scene* getScene()`

Returns the current main scene, or `nullptr`/`nil` if none is set.

addSceneLayer

- `static void addSceneLayer(Scene* scene)`

Renders an additional scene as a layer on top of the main scene. Commonly used for HUDs, overlays, or pause menus.

executeSceneOnce

- `static void executeSceneOnce(Scene* scene)`

Adds a scene that runs for a single draw, then is automatically removed. Useful for rendering a scene to a texture on demand, such as a thumbnail or a preview image. Passing `nullptr` has no effect.

The scene keeps drawing on every frame until its resources finish loading, so with asynchronous loading it may take several frames before it is removed. Use hasScenesToExecuteOnce to tell whether any are still pending.

removeScene

- `static void removeScene(Scene* scene)`

Removes a specific scene from the active scene stack. If it was queued with executeSceneOnce, the pending one-shot execution is cancelled so the scene can be destroyed safely after this method returns.

removeAllSceneLayers

- `static void removeAllSceneLayers(bool removeOneTimeScenes)`

Removes all layered scenes. When `removeOneTimeScenes` is `true`, one-time scenes added via executeSceneOnce are also removed.

removeAllScenes

- `static void removeAllScenes()`

Removes all scenes, including the main scene and all layers.

isSceneRunning

- `static bool isSceneRunning(Scene* scene)`

Returns `true` if the given scene is currently in the active scene stack.

hasScenesToExecuteOnce

- `static bool hasScenesToExecuteOnce()`

Returns `true` while any scene queued with executeSceneOnce is still waiting to be drawn. Useful when the draw loop only runs on demand: a renderer that skips idle frames must keep drawing while this is `true`, or the queued scenes never render.

C++

```cpp
Engine::executeSceneOnce(&previewScene);
if (Engine::hasScenesToExecuteOnce()) {
    // Keep the draw loop running this frame.
}
```

Lua

```lua
Engine.executeSceneOnce(previewScene)
if Engine.hasScenesToExecuteOnce() then
    -- Keep the draw loop running this frame.
end
```

getMainScene

- `static Scene* getMainScene()`

Returns the scene set with setScene.

getLastScene

- `static Scene* getLastScene()`

Returns the top-most scene in the stack (the most recently added layer, or the main scene if no layers exist).

pauseGameEvents

- `static void pauseGameEvents(bool pause)`

Pauses delivery of gameplay events (onUpdate, onFixedUpdate, input callbacks) without stopping the render loop. The editor uses this when the project is not in play mode.

setCanvasSize

- `static void setCanvasSize(int canvasWidth, int canvasHeight)`

Sets the preferred logical canvas dimensions. The actual canvasWidth/canvasHeight reported at runtime may differ depending on the active scalingMode. It applies immediately, also while a scene runs: UI and auto-resized cameras follow the new size.

getViewRect

- `static Rect getViewRect()`

Returns the viewport rectangle calculated from canvas size, window size, and scaling mode.

setMousePosition

- `static void setMousePosition(float x, float y)`

Sets the mouse position in logical canvas coordinates and updates `Input.getMousePosition()`.

On desktop GLFW builds this also moves the OS cursor. On platforms that do not expose cursor warping, the engine still updates its internal input position, and the next real mouse event may replace it with the platform-reported position.

Lua

```lua
Engine.setMousePosition(Engine.canvasWidth * 0.5, Engine.canvasHeight * 0.5)
```

C++

```cpp
Engine::setMousePosition(
    Engine::getCanvasWidth() * 0.5f,
    Engine::getCanvasHeight() * 0.5f
);
```

setUpdateTimeMS

- `static void setUpdateTimeMS(unsigned int updateTimeMS)`

Same as updateTime but accepts the interval in **milliseconds**.

isUIEventReceived

- `static bool isUIEventReceived()`

Returns `true` inside an input callback when a UI widget consumed that event. With ignoreEventsHandledByUI turned off, gameplay code can use it to skip UI clicks and menu keys.

isViewLoaded

- `static bool isViewLoaded()`

Returns `true` after the graphics surface is ready (after onViewLoaded fires).

clearAllSubscriptions

- `static void clearAllSubscriptions(bool includeLifecycle)`

Removes all registered callbacks from all engine events. When `includeLifecycle` is `false`, lifecycle events such as `onViewLoaded` are preserved.

clearComponentSubscriptions

- `static void clearComponentSubscriptions(Scene* scene)`

Removes the callbacks registered on a scene's components — Button (p. 506) `onPress`, UI (p. 698) pointer and focus events, panel, scrollbar, text edit, sound, and action events. clearAllSubscriptions covers the engine's own events; this one covers the per-component events.

Callbacks a script adds directly, as in `getComponent<ButtonComponent>().onPress.add("pressed", ...)`, stay on the component after the script is gone, so call this before the code behind them is unloaded — the editor does it when a scene stops and the project library is released.

Lua

```
Engine.clearComponentSubscriptions(scene)
```

C++

```
Engine::clearComponentSubscriptions(&scene);
```

setMaxResourceLoadingThreads

- `static void setMaxResourceLoadingThreads(size_t maxThreads)`

Sets the maximum number of worker threads used for background asset loading.

getQueuedResourceCount

- `static size_t getQueuedResourceCount()`

Returns the number of resources still waiting to be loaded on background threads.

clearPools

- `static void clearPools()`
- C++ only

Fully clears the texture, texture-data, model, shader, sound, and font caches. This also destroys pooled GPU textures and shaders, including entries that are still referenced elsewhere. Use it for engine or graphics-view teardown, not routine level unloading.

Before clearing during a project or runtime-context switch, stop work that can still write into the pools, such as pending asynchronous model loads.

clearUnusedPools

- `static void clearUnusedPools()`
- C++ only

Removes cache entries that are no longer referenced outside their pool while keeping resources that active scenes or engine objects still own. Pending sound and decoded texture-data work is allowed to finish before those caches are swept. Shader missing/failed state and custom-shader registrations are also reset.

This is the safer cleanup option after destroying a scene or switching projects.

startAsyncThread / commitThreadQueue / endAsyncThread / isAsyncThread

Used to safely create GPU resources from background threads:

1. Call `startAsyncThread()` at the beginning of the worker function.
2. Create textures, meshes, etc.
3. Call `commitThreadQueue()` periodically (or at the end) to flush pending GPU uploads on the main thread.
4. Call `endAsyncThread()` when the worker is finished.

`AsyncThreadScope` (C++ RAII) wraps start/end automatically.

Callback event details

onViewLoaded

- `static FunctionSubscribe<void()> onViewLoaded`

Called once when the graphics backend surface is ready for rendering.

onCanvasChanged

- `static FunctionSubscribe<void()> onCanvasChanged`
- Callback: `void()`

Called when the canvas or window size changes. Re-query canvasWidth/canvasHeight or reposition UI elements here.

onViewDestroyed

- `static FunctionSubscribe<void()> onViewDestroyed`
- Callback: `void()`

Called when the graphics surface is being destroyed (e.g. app going to background on mobile).

onDraw

- `static FunctionSubscribe<void()> onDraw`
- Callback: `void()`

Called every frame after the updates, right before the scenes are drawn. A minimized Windows, Linux, or macOS game keeps updating without drawing and skips it, as does a macOS game whose window can't be seen, so keep game logic in onUpdate.

onUpdate

- `static FunctionSubscribe<void()> onUpdate`
- Callback: `void()`

Called every frame for variable-step gameplay logic. Multiply movement by deltatime to ensure frame-rate independence.

onFixedUpdate

- `static FunctionSubscribe<void()> onFixedUpdate`
- Callback: `void()`

Called at a fixed interval defined by updateTime. Use for physics, deterministic simulation, and anything that must not depend on frame rate.

onPostUpdate

- `static FunctionSubscribe<void()> onPostUpdate`
- Callback: `void()`

Called after the regular update and fixed-update passes each frame. Useful for late-stage logic such as camera follow that must run after all objects have moved.

onPause

- `static FunctionSubscribe<void()> onPause`
- Callback: `void()`

Called when the system suspends the application (e.g. phone call received, app moved to background). It also fires while a full-screen ad covers the game and while a web portal pauses it; see Ads pause the game (p. 338).

onResume

- `static FunctionSubscribe<void()> onResume`
- Callback: `void()`

Called when the application returns to the foreground after a pause, including after each full-screen ad.

onShutdown

- `static FunctionSubscribe<void()> onShutdown`
- Callback: `void()`

Called when the application is closing. Use to flush saves or clean up persistent state.

onSceneLoaded

- `static FunctionSubscribe<void()> onSceneLoaded`
- Callback: `void()`

Called once a scene stack loaded by `SceneManager::loadScene` (p. 603) has loaded its resources, and once for the start scene, both in exported games and in the editor's Play. It fires before the loading scene (p. 605) closes, so a level can place its player and camera behind it. `SceneManager::isLoading()` is still `true` here, and a loading scene can hold (p. 606) the load to fade out. `addChildScene` does not fire it.

Lua

```lua
function Level:init()
    RegisterEngineEvent(self, "onSceneLoaded")
end

function Level:onSceneLoaded()
    self:startLevel()
end
```

C++

```cpp
Level::Level(Scene* scene, Entity entity): ScriptBase(scene, entity) {
    REGISTER_ENGINE_EVENT(onSceneLoaded);
}

Level::~Level() {
    UNREGISTER_ENGINE_EVENT(onSceneLoaded);
}

void Level::onSceneLoaded() {
    startLevel();
}
```

onTouchStart

- `static FunctionSubscribe<void(int,float,float)> onTouchStart`
- Callback: `void(int pointer, float x, float y)`
  - **pointer** — Touch identifier for multitouch tracking.
  - **x** — Horizontal position in canvas coordinates.
  - **y** — Vertical position in canvas coordinates.

Called when a new touch contact begins.

onTouchEnd

- `static FunctionSubscribe<void(int,float,float)> onTouchEnd`
- Callback: `void(int pointer, float x, float y)`
  - **pointer** — Touch identifier.
  - **x** — Final horizontal position in canvas coordinates.
  - **y** — Final vertical position in canvas coordinates.

Called when a touch contact is lifted.

onTouchMove

- `static FunctionSubscribe<void(int,float,float)> onTouchMove`
- Callback: `void(int pointer, float x, float y)`
  - **pointer** — Touch identifier.
  - **x** — Updated horizontal position.
  - **y** — Updated vertical position.

Called whenever a touch contact moves.

onTouchCancel

- `static FunctionSubscribe<void(int,float,float)> onTouchCancel`
- Callback: `void(int pointer, float x, float y)`
  - **pointer** — Identifier of the canceled touch.
  - **x** — Last recorded horizontal position in logical canvas coordinates.
  - **y** — Last recorded vertical position in logical canvas coordinates.

Called when a tracked touch is interrupted, after it has been removed from `Input`. A per-pointer cancellation preserves other active touches. A full cancellation removes all tracked touches before emitting this callback once per touch. There is no separate global callback, and a full cancellation with no tracked touches emits nothing. Canceling an unknown or already removed identifier also emits nothing. Use `Input::numTouches()` (`Input.numTouches()` in Lua) to check whether any fingers remain; it is zero during every full-cancel callback.

Use this alongside `onTouchEnd` to clear actions associated with a finger. Cancellation does not call UI pointer-up, `onClick`, or `onTouchEnd`. Notifications follow `pauseGameEvents`, but do not depend on whether UI consumed the original touch or whether its position is inside the canvas.

C++ scripts subscribe with `REGISTER_ENGINE_EVENT(onTouchCancel)` and unsubscribe with `UNREGISTER_ENGINE_EVENT(onTouchCancel)`. Lua scripts use `RegisterEngineEvent(self, "onTouchCancel")` with a handler named `function Player:onTouchCancel(pointer, x, y)`.

---

onMouseDown

- `static FunctionSubscribe<void(int,float,float,int)> onMouseDown`
- Callback: `void(int button, float x, float y, int mods)`
  - **button** — Mouse button index; see Input (p. 381) constants.
  - **x** — Horizontal position in canvas coordinates.
  - **y** — Vertical position in canvas coordinates.
  - **mods** — Modifier key bitmask; see Input (p. 382).

Called when a mouse button is pressed.

---

onMouseUp

- `static FunctionSubscribe<void(int,float,float,int)> onMouseUp`
- Callback: `void(int button, float x, float y, int mods)`
  - **button** — Mouse button index.
  - **x** — Horizontal position.
  - **y** — Vertical position.
  - **mods** — Modifier key bitmask.

Called when a mouse button is released.

---

onMouseMove

- `static FunctionSubscribe<void(float,float,int)> onMouseMove`
- Callback: `void(float x, float y, int mods)`
  - **x** — Horizontal position in canvas coordinates.
  - **y** — Vertical position in canvas coordinates.
  - **mods** — Modifier key bitmask.

Called when the mouse cursor moves. Also called outside the canvas when allowEventsOutCanvas is `true`.

When mouseMode is `CAPTURED`, this event keeps firing from relative mouse movement, and `x/y` can represent a virtual position outside the normal canvas range.

onMouseScroll

- `static FunctionSubscribe<void(float,float,int)> onMouseScroll`
- Callback: `void(float xoffset, float yoffset, int mods)`
  - **xoffset** — Horizontal scroll amount.
  - **yoffset** — Vertical scroll amount (positive = scroll up).
  - **mods** — Modifier key bitmask.

Called when the mouse scroll wheel changes.

onMouseEnter

- `static FunctionSubscribe<void()> onMouseEnter`
- Callback: `void()`

Called when the mouse cursor enters the canvas boundary.

onMouseLeave

- `static FunctionSubscribe<void()> onMouseLeave`
- Callback: `void()`

Called when the mouse cursor leaves the canvas boundary.

onKeyDown

- `static FunctionSubscribe<void(int,bool,int)> onKeyDown`
- Callback: `void(int key, bool repeat, int mods)`
  - **key** — Key code; see Input (p. 381) constants (`D_KEY_*` in C++, `Input.KEY_*` in Lua).
  - **repeat** — `true` if the key is being held and this is a repeat event.
  - **mods** — Modifier key bitmask.

Called when any keyboard key is pressed.

onKeyUp

- `static FunctionSubscribe<void(int,bool,int)> onKeyUp`
- Callback: `void(int key, bool repeat, int mods)`
  - **key** — Key code.
  - **repeat** — Always `false` on key-up.
  - **mods** — Modifier key bitmask.

Called when any keyboard key is released.

onCharInput

- `static FunctionSubscribe<void(wchar_t)> onCharInput`
- Callback: `void(wchar_t codepoint)`
  - **codepoint** — Unicode code point of the typed character, including composed/accented characters. Lua receives it as a one-character UTF-8 string.

Called for text input. Prefer this over onKeyDown when implementing text fields.

On Windows, macOS, and iOS, a character past U+FFFF, such as an emoji, arrives as two calls carrying its UTF-16 halves. A TextEdit (p. 533) joins them into one character, while Lua receives each half as U+FFFD.

onGamepadConnect

- `static FunctionSubscribe<void(int)> onGamepadConnect`
- Callback: `void(int gamepad)`
  - **gamepad** — Id of the controller that connected.

Called when a controller is connected (and once for each controller already connected at startup). Query its name with Input.getGamepadName (p. 385).

onGamepadDisconnect

- `static FunctionSubscribe<void(int)> onGamepadDisconnect`
- Callback: `void(int gamepad)`
  - **gamepad** — Id of the controller that disconnected.

Called when a controller is disconnected. After this fires, Input.isGamepadConnected (p. 384) returns `false` for that id.

onGamepadButtonDown

- `static FunctionSubscribe<void(int,int)> onGamepadButtonDown`
- Callback: `void(int gamepad, int button)`
  - **gamepad** — Id of the controller.
  - **button** — Button index; see Input (p. 382) constants (`D_GAMEPAD_BUTTON_*` in C++, `Input.GAMEPAD_BUTTON_*` in Lua).

Called when a controller button is pressed.

onGamepadButtonUp

- `static FunctionSubscribe<void(int,int)> onGamepadButtonUp`
- Callback: `void(int gamepad, int button)`
  - **gamepad** — Id of the controller.
  - **button** — Button index.

Called when a controller button is released.

onGamepadAxisMove

- `static FunctionSubscribe<void(int,int,float)> onGamepadAxisMove`
- Callback: `void(int gamepad, int axis, float value)`
  - **gamepad** — Id of the controller.
  - **axis** — Axis index; see Input (p. 382) constants (`D_GAMEPAD_AXIS_*` in C++, `Input.GAMEPAD_AXIS_*` in Lua).
  - **value** — Axis value in `-1.0`…`1.0`. Sticks are down-positive on the Y axis; triggers rest at `-1` and reach `+1` when fully pressed.

Called when an analog axis (stick or trigger) changes. For always-on movement, polling Input.getGamepadAxis (p. 385) in `onUpdate` is usually simpler than tracking this event.

EntityManager

Low-level entity ID allocation.

Methods

Name	Languages
`createUserEntity`	C++ | Lua
`createSystemEntity`	C++ | Lua
`destroy`	C++ | Lua
`setSignature`	C++ | Lua
`getSignature`	C++ | Lua
`setName`	C++ | Lua
`getName`	C++ | Lua
`isCreated`	C++ | Lua
`recreateEntity`	C++ | Lua
`getLastUserEntity`	C++ | Lua

EntityRegistry

Base of Scene (p. 390). Entity/component CRUD and hierarchy.

Methods

Name	Description
`createEntity` / `createUserEntity` / `createSystemEntity`	Allocate ids
`destroyEntity`	Remove entity and components
`addEntityChild`	Parent/child link
`setEntityName` / `getEntityName`	Debug/editor names
`findEntity`	Look up an existing entity by name
`findComponent` / `addComponent` / `removeComponent`	ECS data access
`getEntityList`	All entity ids

User entities start at id 1001; system entities use 1–1000.

findEntity

```
Entity findEntity(name)
Entity findEntity(name, parent)
```

Returns the first entity whose name (as shown in the editor Structure panel) matches `name`, or `NULL_ENTITY` when there is no match. The two-argument form restricts the search to descendants of `parent`, so you can resolve a named child without relying on globally unique names.

Use it to reference an entity that already exists in the scene — for example a child model you want to animate — and then wrap it with the matching object handle:

```lua
local modelEntity = self.scene:findEntity("model", self.entity)
if modelEntity ~= NULL_ENTITY then
    local model = Model(self.scene, modelEntity)   -- wraps; does not create
    model:getAnimation(0):start()
end
```

> **Wrap, don't recreate**
>
> Pass the found entity to the **two-argument** handle constructor (`Model(scene, entity)`). The one-argument form (`Model(scene)`) creates a brand-new entity instead of referencing the existing one. See Referencing other entities (p. 203).

The name lookup is linear over the entity list; cache the result in `init()` rather than calling it every frame.

Input

Description

`Input` is a static polling class that provides the current keyboard, mouse, touch, and gamepad state at any point during a frame. Use it inside `onUpdate` or `onFixedUpdate` to check whether a key or button is held down. For event-based input (press/release and controller connect/disconnect notifications), subscribe to the Engine (p. 361) callback events instead.

> **Constant naming differs by language**
> Key, mouse-button, gamepad, and modifier constants are C++ preprocessor macros prefixed `D_` (e.g. `D_KEY_SPACE`, `D_MOUSE_BUTTON_LEFT`, `D_GAMEPAD_BUTTON_A`). In Lua they are static properties on the `Input` class with the `D_` prefix dropped (e.g. `Input.KEY_SPACE`, `Input.MOUSE_BUTTON_LEFT`, `Input.GAMEPAD_BUTTON_A`).

C++

```cpp
#include "Doriax.h"
using namespace doriax;

// Inside an update callback:
if (Input::isKeyPressed(D_KEY_SPACE)) {
    player.jump();
}
```

Lua

```lua
-- Inside update:
if Input.isKeyPressed(Input.KEY_SPACE) then
    player:jump()
end
```

Methods

Returns	Name	Languages
static bool	isKeyPressed	C++ | Lua
static bool	isMousePressed	C++ | Lua
static bool	isTouch	C++ | Lua
static bool	isMouseEntered	C++ | Lua
static Vector2	getMousePosition	C++ | Lua
static Vector2	getMouseScroll	C++ | Lua
static Vector2	getTouchPosition	C++ | Lua
static std::vector<Touch>	getTouches	C++ | Lua
static size_t	numTouches	C++ | Lua
static bool	isGamepadConnected	C++ | Lua
static std::string	getGamepadName	C++ | Lua
static bool	isGamepadButtonPressed	C++ | Lua
static float	getGamepadAxis	C++ | Lua
static std::vector<Gamepad>	getGamepads	C++
static size_t	numGamepads	C++ | Lua
static int	getGamepadId	C++ | Lua
static int	getModifiers	C++ | Lua
static size_t	findTouchIndex	C++ | Lua

Returns	Name	Languages
static size_t	findGamepadIndex	C++

Keys

Key constants are preprocessor `#define` values in C++ (prefix `D_KEY_`) and static properties on `Input` in Lua (prefix `Input.KEY_`). The table below lists the C++ form; for Lua, replace `D_KEY_` with `Input.KEY_`.

Constant	Value	Description
`D_KEY_SPACE`	32	Space bar
`D_KEY_A` … `D_KEY_Z`	65–90	Letter keys
`D_KEY_0` … `D_KEY_9`	48–57	Number row
`D_KEY_ESCAPE`	256	Escape
`D_KEY_ENTER`	257	Enter / Return
`D_KEY_TAB`	258	Tab
`D_KEY_BACKSPACE`	259	Backspace
`D_KEY_INSERT`	260	Insert
`D_KEY_DELETE`	261	Delete
`D_KEY_RIGHT`	262	Arrow right
`D_KEY_LEFT`	263	Arrow left
`D_KEY_DOWN`	264	Arrow down
`D_KEY_UP`	265	Arrow up
`D_KEY_PAGE_UP`	266	Page Up
`D_KEY_PAGE_DOWN`	267	Page Down
`D_KEY_HOME`	268	Home
`D_KEY_END`	269	End
`D_KEY_CAPS_LOCK`	280	Caps Lock
`D_KEY_F1` … `D_KEY_F25`	290–314	Function keys
`D_KEY_KP_0` … `D_KEY_KP_9`	320–329	Numpad digits
`D_KEY_KP_ENTER`	335	Numpad Enter
`D_KEY_LEFT_SHIFT`	340	Left Shift
`D_KEY_LEFT_CONTROL`	341	Left Ctrl
`D_KEY_LEFT_ALT`	342	Left Alt
`D_KEY_LEFT_SUPER`	343	Left Super (Win/Cmd)
`D_KEY_RIGHT_SHIFT`	344	Right Shift
`D_KEY_RIGHT_CONTROL`	345	Right Ctrl
`D_KEY_RIGHT_ALT`	346	Right Alt
`D_KEY_RIGHT_SUPER`	347	Right Super
`D_KEY_MENU`	348	Menu key

Mouse buttons

In Lua, replace the `D_MOUSE_BUTTON_` prefix with `Input.MOUSE_BUTTON_`.

C++ constant	Lua property	Value	Description
`D_MOUSE_BUTTON_LEFT`	`Input.MOUSE_BUTTON_LEFT`	0	Left button
`D_MOUSE_BUTTON_RIGHT`	`Input.MOUSE_BUTTON_RIGHT`	1	Right button
`D_MOUSE_BUTTON_MIDDLE`	`Input.MOUSE_BUTTON_MIDDLE`	2	Middle button

C++ constant	Lua property	Value	Description
`D_MOUSE_BUTTON_1` … `D_MOUSE_BUTTON_8`	`Input.MOUSE_BUTTON_1` … `Input.MOUSE_BUTTON_8`	0–7	All buttons

Modifiers

Modifier bits returned by getModifiers and passed to key/mouse callbacks. In Lua, replace the `D_MODIFIER_` prefix with `Input.MODIFIER_`.

C++ constant	Lua property	Bit	Description
`D_MODIFIER_SHIFT`	`Input.MODIFIER_SHIFT`	`0x0001`	Shift key held
`D_MODIFIER_CONTROL`	`Input.MODIFIER_CONTROL`	`0x0002`	Ctrl key held
`D_MODIFIER_ALT`	`Input.MODIFIER_ALT`	`0x0004`	Alt key held
`D_MODIFIER_SUPER`	`Input.MODIFIER_SUPER`	`0x0008`	Super (Win/Cmd) held
`D_MODIFIER_CAPS_LOCK`	`Input.MODIFIER_CAPS_LOCK`	`0x0010`	Caps Lock active
`D_MODIFIER_NUM_LOCK`	`Input.MODIFIER_NUM_LOCK`	`0x0020`	Num Lock active

Gamepad buttons

Button indices follow the standard Xbox-style layout. In Lua, replace the `D_GAMEPAD_BUTTON_` prefix with `Input.GAMEPAD_BUTTON_`. The PlayStation-style aliases are the same underlying values.

C++ constant	Lua property	Value	Description
`D_GAMEPAD_BUTTON_A`	`Input.GAMEPAD_BUTTON_A`	0	A (alias `CROSS`)
`D_GAMEPAD_BUTTON_B`	`Input.GAMEPAD_BUTTON_B`	1	B (alias `CIRCLE`)
`D_GAMEPAD_BUTTON_X`	`Input.GAMEPAD_BUTTON_X`	2	X (alias `SQUARE`)
`D_GAMEPAD_BUTTON_Y`	`Input.GAMEPAD_BUTTON_Y`	3	Y (alias `TRIANGLE`)
`D_GAMEPAD_BUTTON_LEFT_BUMPER`	`Input.GAMEPAD_BUTTON_LEFT_BUMPER`	4	Left bumper / L1
`D_GAMEPAD_BUTTON_RIGHT_BUMPER`	`Input.GAMEPAD_BUTTON_RIGHT_BUMPER`	5	Right bumper / R1
`D_GAMEPAD_BUTTON_BACK`	`Input.GAMEPAD_BUTTON_BACK`	6	View / Back / Share
`D_GAMEPAD_BUTTON_START`	`Input.GAMEPAD_BUTTON_START`	7	Menu / Start / Options
`D_GAMEPAD_BUTTON_GUIDE`	`Input.GAMEPAD_BUTTON_GUIDE`	8	Guide / PS button
`D_GAMEPAD_BUTTON_LEFT_THUMB`	`Input.GAMEPAD_BUTTON_LEFT_THUMB`	9	Left stick click / L3
`D_GAMEPAD_BUTTON_RIGHT_THUMB`	`Input.GAMEPAD_BUTTON_RIGHT_THUMB`	10	Right stick click / R3
`D_GAMEPAD_BUTTON_DPAD_UP`	`Input.GAMEPAD_BUTTON_DPAD_UP`	11	D-pad up
`D_GAMEPAD_BUTTON_DPAD_RIGHT`	`Input.GAMEPAD_BUTTON_DPAD_RIGHT`	12	D-pad right
`D_GAMEPAD_BUTTON_DPAD_DOWN`	`Input.GAMEPAD_BUTTON_DPAD_DOWN`	13	D-pad down
`D_GAMEPAD_BUTTON_DPAD_LEFT`	`Input.GAMEPAD_BUTTON_DPAD_LEFT`	14	D-pad left

The aliases `D_GAMEPAD_BUTTON_CROSS`, `CIRCLE`, `SQUARE`, and `TRIANGLE` (Lua `Input.GAMEPAD_BUTTON_CROSS`, …) resolve to `A`, `B`, `X`, and `Y` respectively.

Gamepad axes

Axis values range from `-1.0` to `1.0`. Sticks are **down-positive** on the Y axis (up is `-1`), and triggers **rest at `-1`**, reaching `+1` when fully pressed. In Lua, replace the `D_GAMEPAD_AXIS_` prefix with `Input.GAMEPAD_AXIS_`.

C++ constant	Lua property	Value	Range
`D_GAMEPAD_AXIS_LEFT_X`	`Input.GAMEPAD_AXIS_LEFT_X`	0	Left stick X: `-1` left … `+1` right
`D_GAMEPAD_AXIS_LEFT_Y`	`Input.GAMEPAD_AXIS_LEFT_Y`	1	Left stick Y: `-1` up … `+1` down

C++ constant	Lua property	Value	Range
`D_GAMEPAD_AXIS_RIGHT_X`	`Input.GAMEPAD_AXIS_RIGHT_X`	2	Right stick X: `-1` left … `+1` right
`D_GAMEPAD_AXIS_RIGHT_Y`	`Input.GAMEPAD_AXIS_RIGHT_Y`	3	Right stick Y: `-1` up … `+1` down
`D_GAMEPAD_AXIS_LEFT_TRIGGER`	`Input.GAMEPAD_AXIS_LEFT_TRIGGER`	4	Left trigger: `-1` released … `+1` pressed
`D_GAMEPAD_AXIS_RIGHT_TRIGGER`	`Input.GAMEPAD_AXIS_RIGHT_TRIGGER`	5	Right trigger: `-1` released … `+1` pressed

Method details

isKeyPressed

- `static bool isKeyPressed(int key)`

Returns `true` while the given key is held down. Use `D_KEY_*` constants in C++ and `Input.KEY_*` in Lua for the key code.

C++

```
if (Input::isKeyPressed(D_KEY_W)) {
    player.moveForward(Engine::getDeltatime() * speed);
}
```

Lua

```
if Input.isKeyPressed(Input.KEY_W) then
    player:moveForward(Engine.deltatime * speed)
end
```

isMousePressed

- `static bool isMousePressed(int button)`

Returns `true` while the given mouse button is held down. Use `D_MOUSE_BUTTON_*` constants in C++ and `Input.MOUSE_BUTTON_*` in Lua.

C++

```
if (Input::isMousePressed(D_MOUSE_BUTTON_LEFT)) {
    // drag logic
}
```

Lua

```
if Input.isMousePressed(Input.MOUSE_BUTTON_LEFT) then
    -- drag logic
end
```

isTouch

- `static bool isTouch()`

Returns `true` if there is at least one active touch contact.

isMouseEntered

- `static bool isMouseEntered()`

Returns `true` while the mouse cursor is inside the canvas boundary.

getMousePosition

- `static Vector2 getMousePosition()`

Current cursor position in canvas coordinates. Updated every frame; use for hover detection in `onUpdate`.

This value can also be set manually with Engine.setMousePosition (p. 370). When Engine.mouseMode (p. 368) is `CAPTURED`, the position is a virtual canvas-space position accumulated from relative mouse movement and may move outside the normal canvas range.

C++

```
Vector2 pos = Input::getMousePosition();
```

Lua

```
local pos = Input.getMousePosition()
```

getMouseScroll

- `static Vector2 getMouseScroll()`

Accumulated scroll wheel delta since the last frame, in `(xoffset, yoffset)`. Reset each frame.

getTouchPosition

- `static Vector2 getTouchPosition(int pointer)`

Returns the current position of the touch contact identified by `pointer`. Use findTouchIndex to convert a pointer ID to an array index if needed.

getTouches

- `static std::vector<Touch> getTouches()`

Returns all active touch contacts. Each `Touch` has `int pointer` and `Vector2 position`.

C++

```
for (const auto& touch : Input::getTouches()) {
    drawCircle(touch.position.x, touch.position.y, 10.0f);
}
```

Lua

```
local touches = Input.getTouches()
for i = 1, #touches do
    drawCircle(touches[i].position.x, touches[i].position.y, 10)
end
```

numTouches

- `static size_t numTouches()`

Returns the count of currently active touch contacts.

isGamepadConnected

- `static bool isGamepadConnected(int id)`

Returns `true` while a controller with the given id is connected. Ids are assigned per platform and are **sparse** — enumerate connected controllers with getGamepadId, not by assuming ids `0..numGamepads()-1`.

C++

```
if (Input::isGamepadConnected(0)) {
    // controller 0 is present
}
```

Lua

```lua
if Input.isGamepadConnected(0) then
    -- controller 0 is present
end
```

getGamepadName

- `static std::string getGamepadName(int id)`

Returns the human-readable name of the controller with the given id, or an empty string if it is not connected.

isGamepadButtonPressed

- `static bool isGamepadButtonPressed(int id, int button)`

Returns `true` while the given button is held down on controller `id`. Use `D_GAMEPAD_BUTTON_*` constants in C++ and `Input.GAMEPAD_BUTTON_*` in Lua. Returns `false` for an unknown controller.

C++

```cpp
if (Input::isGamepadButtonPressed(0, D_GAMEPAD_BUTTON_A)) {
    player.jump();
}
```

Lua

```lua
if Input.isGamepadButtonPressed(0, Input.GAMEPAD_BUTTON_A) then
    player:jump()
end
```

getGamepadAxis

- `static float getGamepadAxis(int id, int axis)`

Returns the current value of an analog axis on controller `id`, in the range `-1.0`…`1.0`. Use `D_GAMEPAD_AXIS_*` constants in C++ and `Input.GAMEPAD_AXIS_*` in Lua. Returns `0.0` for an unknown controller.

Sticks are down-positive on the Y axis (up is `-1`). **Triggers rest at `-1`** and reach `+1` when fully pressed — a released trigger reads `-1` from the moment the controller connects, not `0`.

C++

```cpp
float x = Input::getGamepadAxis(0, D_GAMEPAD_AXIS_LEFT_X);
if (fabsf(x) > 0.15f) {  // dead zone
    player.strafe(x);
}
```

Lua

```lua
local x = Input.getGamepadAxis(0, Input.GAMEPAD_AXIS_LEFT_X)
if math.abs(x) > 0.15 then  -- dead zone
    player:strafe(x)
end
```

getGamepads

- `static std::vector<Gamepad> getGamepads()`
- C++ only

Returns all connected controllers. Each `Gamepad` holds an `int id`, a `std::string name`, and the current button and axis state. In Lua, iterate with numGamepads and getGamepadId instead.

numGamepads

- `static size_t numGamepads()`

Returns the count of currently connected controllers. This is a count, **not** the highest id — combine it with getGamepadId to iterate.

getGamepadId

- `static int getGamepadId(size_t index)`

Returns the real id of the `index`-th connected controller, or `-1` if `index` is out of range. Because gamepad ids are sparse (they can have gaps after a disconnect), this is the correct way to enumerate connected controllers:

C++

```cpp
for (size_t i = 0; i < Input::numGamepads(); i++) {
    int id = Input::getGamepadId(i);
    // poll controller `id`
}
```

Lua

```lua
for i = 0, Input.numGamepads() - 1 do
    local id = Input.getGamepadId(i)
    -- poll controller `id`
end
```

getModifiers

- `static int getModifiers()`

Returns the current modifier key bitmask. Check individual modifiers with bitwise AND.

C++

```cpp
int mods = Input::getModifiers();
if (mods & D_MODIFIER_SHIFT) {
    // Shift is held
}
```

Lua

```lua
local mods = Input.getModifiers()
if (mods & Input.MODIFIER_SHIFT) ~= 0 then
    -- Shift is held
end
```

findTouchIndex

- `static size_t findTouchIndex(int pointer)`

Returns the index into the getTouches array for the given pointer ID. Returns `SIZE_MAX` if not found.

findGamepadIndex

- `static size_t findGamepadIndex(int id)`
- C++ only

Returns the internal index of the controller with the given id, or `SIZE_MAX` if it is not connected. Most code should use isGamepadConnected and getGamepadId instead.

Log

Description

`Log` provides static methods for writing messages to the platform console or log sink. Output goes to the IDE console during editor play mode, and to `stdout`/`logcat`/Xcode console on each platform. All methods accept `printf`-style format strings.

In Lua, the global `print()` function is also available as an alias for `Log.print`.

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Log::print("Player health: %d", health);
Log::warn("Physics body not loaded yet!");
```

Lua

```lua
Log.print("Player health: " .. health)
Log.warn("Physics body not loaded yet!")
-- or use the Lua global:
print("Hello from script")
```

Methods

Returns	Name	Languages
static void	print	C++ | Lua
static void	verbose	C++ | Lua
static void	debug	C++ | Lua
static void	warn	C++ | Lua
static void	error	C++ | Lua

Log levels

The engine filters messages based on the build configuration (defined in `System.h`):

Macro	Level	Description
`D_LOG_VERBOSE`	1	Detailed trace information; typically disabled in release builds.
`D_LOG_DEBUG`	2	Debug-level messages useful during development.
`D_LOG_WARN`	3	Non-fatal warnings that may indicate incorrect usage.
`D_LOG_ERROR`	4	Errors that require attention; may terminate the operation.

Method details

print

- `static void print(const char* text, ...)`

Writes a plain message to the log with no severity prefix. Use for general informational output and debugging.

C++

```cpp
Log::print("Score: %d  Time: %.2f", score, elapsed);
```

Lua

```lua
Log.print("Score: " .. score .. "  Time: " .. string.format("%.2f", elapsed))
-- or:
print("Score:", score)
```

verbose

- `static void verbose(const char* text, ...)`

Logs a `VERBOSE`-level message. Typically compiled out in release; use for high-frequency trace output such as per-frame diagnostics.

C++

```cpp
Log::verbose("Frame %d: entity count = %zu", frame, scene.getEntityCount());
```

Lua

```lua
Log.verbose("Frame " .. frame .. ": entity count = " .. count)
```

debug

- `static void debug(const char* text, ...)`

Logs a `DEBUG`-level message. Use for state dumps, object creation, and lifecycle tracking during active development.

C++

```cpp
Log::debug("Enemy spawned at (%.1f, %.1f)", x, y);
```

Lua

```lua
Log.debug("Enemy spawned at " .. x .. ", " .. y)
```

warn

- `static void warn(const char* text, ...)`

Logs a `WARN`-level message. Use for recoverable issues: missing optional resources, fallback paths taken, or unexpected but non-fatal conditions.

C++

```cpp
if (!body.isEnabled()) {
    Log::warn("Trying to apply force to a disabled body (entity %u)", entity);
}
```

Lua

```lua
if not body.enabled then
    Log.warn("Trying to apply force to a disabled body")
end
```

error

- `static void error(const char* text, ...)`

Logs an `ERROR`-level message. Use for critical failures that prevent correct execution.

C++

```cpp
if (!mesh.load()) {
    Log::error("Failed to load mesh for entity %u", entity);
}
```

Lua

```lua
if not mesh:load() then
    Log.error("Failed to load mesh")
end
```

Scene

Description

A `Scene` is the root container for all objects, systems, and resources in a project. It manages the active camera, background color, global illumination, shadow settings, mesh detail levels, the optional depth prepass, and the update/draw lifecycle. You typically create one or more scenes at startup and set the active one with `Engine::setScene`.

Inherits: EntityRegistry (p. 379)

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Scene scene;

void MySystem::onUpdate() {
    scene.setBackgroundColor(0.1f, 0.1f, 0.2f);
}
```

Lua

```lua
local scene = Scene()
scene:setBackgroundColor(0.1, 0.1, 0.2)
Engine.setScene(scene)
```

Properties

Type	Name	Default	Languages
Vector4	backgroundColor	`(0,0,0,1)`	C++ | Lua
ShadowQuality	shadowQuality	`LOW`	C++ | Lua
bool	meshLodEnabled	`false`	C++ | Lua
float	meshLodThreshold	`1.0`	C++ | Lua
bool	depthPrepassEnabled	`false`	C++ | Lua
LightState	lightState	`AUTO`	C++ | Lua
float	globalIlluminationIntensity	`1.0`	C++ | Lua
Vector3	globalIlluminationColor	`(1,1,1)`	C++ | Lua
float	ambientLight2DIntensity	`1.0`	C++ | Lua
Vector3	ambientLight2DColor	`(1,1,1)`	C++ | Lua
ShadowQuality	shadow2DQuality	`LOW`	C++ | Lua
Vector2	gravity2D	`(0,-9.81)`	C++ | Lua
Vector3	gravity3D	`(0,-9.81,0)`	C++ | Lua
bool	ssaoEnabled	`false`	C++ | Lua
float	ssaoRadius	`0.5`	C++ | Lua
float	ssaoIntensity	`1.0`	C++ | Lua
float	ssaoBias	`0.025`	C++ | Lua
bool	ssaoHalfResolution	`false`	C++ | Lua
bool	ssaoDebug	`false`	C++ | Lua
bool	ssrEnabled	`false`	C++ | Lua
float	ssrMaxDistance	`8.0`	C++ | Lua
float	ssrThickness	`0.5`	C++ | Lua
int	ssrMaxSteps	`48`	C++ | Lua

Type	Name	Default	Languages
float	ssrIntensity	`1.0`	C++ | Lua
float	ssrBlur	`0.0`	C++ | Lua
int	ssrDebugMode	`0`	C++ | Lua
bool	fixedResolutionEnabled	`false`	C++ | Lua
unsigned int	fixedResolutionWidth	`640`	C++ | Lua
unsigned int	fixedResolutionHeight	`360`	C++ | Lua
TextureFilter	fixedResolutionFilter	`NEAREST`	C++ | Lua
string	defaultMeshShader	`""`	C++ | Lua
string	defaultUIShader	`""`	C++ | Lua
string	defaultSkyShader	`""`	C++ | Lua
string	defaultPointsShader	`""`	C++ | Lua
string	defaultLinesShader	`""`	C++ | Lua
string	defaultWaterShader	`""`	C++ | Lua
vector<PostProcessPass (p. 715)>	postProcessPasses	`{}`	C++ | Lua
UIEventState	enableUIEvents	`NOT_SET`	C++ | Lua

Methods

Returns	Name	Languages
void	load	C++ | Lua
void	destroy	C++ | Lua
void	draw	C++ | Lua
void	update	C++ | Lua
void	fixedUpdate	C++
void	setCamera	C++ | Lua
Entity	getCamera	C++
void	setViewCamera	C++
Entity	getRenderCamera	C++
void	setBackgroundColor	C++ | Lua
Vector4	getBackgroundColor	C++
void	setShadowQuality	C++
ShadowQuality	getShadowQuality	C++
void	setMeshLodEnabled	C++
bool	isMeshLodEnabled	C++
void	setMeshLodThreshold	C++
float	getMeshLodThreshold	C++
void	setDepthPrepassEnabled	C++
bool	isDepthPrepassEnabled	C++
void	setLightState	C++
LightState	getLightState	C++
void	setGlobalIllumination	C++ | Lua
float	getGlobalIlluminationIntensity	C++
Vector3	getGlobalIlluminationColor	C++
void	setAmbientLight2D	C++ | Lua
float	getAmbientLight2DIntensity	C++

Returns	Name	Languages
Vector3	getAmbientLight2DColor	C++
void	setShadow2DQuality	C++
ShadowQuality	getShadow2DQuality	C++
void	setGravity2D	C++ | Lua
Vector2	getGravity2D	C++
void	setGravity3D	C++ | Lua
Vector3	getGravity3D	C++
void	setShaderParameter	C++ | Lua
Vector4	getShaderParameter	C++ | Lua
bool	clearShaderParameter	C++ | Lua
void	setShaderParameterDefaults	C++
const ShaderUniformValues&	getShaderParameterDefaults	C++
void	setViewShaderParameter	C++ | Lua
Vector4	getViewShaderParameter	C++ | Lua
bool	clearViewShaderParameter	C++ | Lua
void	setMainViewShaderParameter	C++ | Lua
Vector4	getMainViewShaderParameter	C++ | Lua
bool	clearMainViewShaderParameter	C++ | Lua
void	setSSAOEnabled	C++
bool	isSSAOEnabled	C++
void	setSSAORadius	C++
float	getSSAORadius	C++
void	setSSAOIntensity	C++
float	getSSAOIntensity	C++
void	setSSAOBias	C++
float	getSSAOBias	C++
void	setSSAOHalfResolution	C++
bool	isSSAOHalfResolution	C++
void	setSSAODebug	C++
bool	isSSAODebug	C++
void	setSSREnabled	C++
bool	isSSREnabled	C++
void	setSSRMaxDistance	C++
float	getSSRMaxDistance	C++
void	setSSRThickness	C++
float	getSSRThickness	C++
void	setSSRMaxSteps	C++
int	getSSRMaxSteps	C++
void	setSSRIntensity	C++
float	getSSRIntensity	C++
void	setSSRBlur	C++
float	getSSRBlur	C++
void	setSSRDebugMode	C++
int	getSSRDebugMode	C++

Returns	Name	Languages
void	setFixedResolutionEnabled	C++
bool	isFixedResolutionEnabled	C++
void	setFixedResolutionWidth	C++
unsigned int	getFixedResolutionWidth	C++
void	setFixedResolutionHeight	C++
unsigned int	getFixedResolutionHeight	C++
void	setFixedResolutionSize	C++ | Lua
void	setFixedResolutionFilter	C++
TextureFilter	getFixedResolutionFilter	C++
void	setDefaultShaderUniform	C++ | Lua
Vector4	getDefaultShaderUniform	C++ | Lua
bool	removeDefaultShaderUniform	C++ | Lua
void	setDefaultShaderUniforms	C++
const ShaderUniformValues&	getDefaultShaderUniforms	C++
void	setPostProcessPasses	C++
vector<PostProcessPass>	getPostProcessPasses	C++
void	setPostProcessUniform	C++ | Lua
Vector4	getPostProcessUniform	C++ | Lua
bool	removePostProcessUniform	C++ | Lua
void	setPostProcessPassEnabled	C++ | Lua
bool	isPostProcessPassEnabled	C++ | Lua
void	enableUIEvents	C++
bool	isEnableUIEvents	C++
bool	canReceiveUIEvents	C++ | Lua
void	updateCameraSize	C++ | Lua
void	removeSubscriptionsByTag	C++

Entity and hierarchy methods — `createEntity`, `destroyEntity`, `setEntityName` / `getEntityName`, `findEntity` (p. 379), `addEntityChild`, `getEntityList` — are inherited from EntityRegistry (p. 379).

Enumerations

LightState

- **OFF** — Disables all lights in the scene; objects are rendered with no dynamic lighting.
- **ON** — Forces dynamic lighting on regardless of light objects present.
- **AUTO** — Activates lighting automatically when at least one light entity exists in the scene (default).

UIEventState

- **NOT_SET** — Inherits UI event behaviour from the engine default.
- **ENABLED** — This scene receives UI pointer events.
- **DISABLED** — This scene ignores UI pointer events.

ShadowQuality

PCF filter quality of shadow edges, shared by shadowQuality (3D shadow maps) and shadow2DQuality (2D lights).

- **NONE** — 1 tap, no filtering (hard edges)
- **LOW** — 3×3 taps in 3D / 5 taps in 2D (default)
- **MEDIUM** — 5×5 taps in 3D / 9 taps in 2D
- **HIGH** — 7×7 taps in 3D / 13 taps in 2D

ShaderType

The shader types that can have a scene default, as taken by setDefaultShaderUniform. The C++ enum has more values, for the engine's internal passes; Lua exposes only these.

- **MESH** — Mesh components (defaultMeshShader)
- **UI** — UI components (defaultUIShader)
- **SKYBOX** — the Sky component (defaultSkyShader)
- **POINTS** — Points components (defaultPointsShader)
- **LINES** — Lines components (defaultLinesShader)
- **WATER** — the water surface (defaultWaterShader)

Property details

backgroundColor

- *Setter:* `void setBackgroundColor(Vector4 color)`
- *Getter:* `Vector4 getBackgroundColor() const`

Background clear color for the scene, in RGBA [0, 1] range. Convenience overloads accept `(r, g, b)` or `(r, g, b, a)` floats directly.

shadowQuality

- *Setter:* `void setShadowQuality(ShadowQuality quality)`
- *Getter:* `ShadowQuality getShadowQuality() const`

Filter quality of 3D shadow map edges (PCF kernel size, applied instantly with no shader rebuild):

- **NONE** — 1 tap, hard edges (or performance savings on mobile)
- **LOW** — 3×3 taps (default)
- **MEDIUM** — 5×5 taps
- **HIGH** — 7×7 taps

The same ShadowQuality enum also drives shadow2DQuality for 2D lights.

Lua exposes this setting as the `shadowQuality` property:

```
scene.shadowQuality = ShadowQuality.MEDIUM
```

There is no separate `shadow3DQuality` Lua property; `shadowQuality` is the 3D shadow quality setting.

meshLodEnabled / meshLodThreshold

- *meshLodEnabled Setter:* `void setMeshLodEnabled(bool enabled)`
- *meshLodEnabled Getter:* `bool isMeshLodEnabled() const`

- *meshLodThreshold Setter:* `void setMeshLodThreshold(float pixels)`
- *meshLodThreshold Getter:* `float getMeshLodThreshold() const`

Scene-wide mesh detail levels (p. 275). `meshLodEnabled` (default `false`) turns them on: each mesh gets simplified index ranges and draws one once its geometric error projects below `meshLodThreshold` screen pixels (default `1`). While it is off no levels are built, and turning it off frees their GPU buffers. The threshold is clamped to `≥ 0`. Changing either takes effect immediately; a mesh can opt out with lodEnabled (p. 440).

In the editor these are **Scene → Mesh Detail (LOD)** when nothing is selected.

C++

```
scene.setMeshLodEnabled(true);
scene.setMeshLodThreshold(2.0f);
```

Lua

```
scene.meshLodEnabled = true
scene.meshLodThreshold = 2.0
```

depthPrepassEnabled

- *Setter:* `void setDepthPrepassEnabled(bool enabled)`
- *Getter:* `bool isDepthPrepassEnabled() const`

Writes a depth-only pass (p. 276) for the main camera before the opaque colour draws, so overlapping pixels fail the depth test instead of running the lit shader. Off by default. Toggling it reloads meshes (the prepass pipelines are built with each mesh).

C++

```
scene.setDepthPrepassEnabled(true);
```

Lua

```
scene.depthPrepassEnabled = true
```

lightState

- *Setter:* `void setLightState(LightState state)`
- *Getter:* `LightState getLightState() const`

Controls whether the render system activates the lighting pass. See LightState.

globalIlluminationIntensity

Part of setGlobalIllumination. Controls the brightness of the ambient light applied uniformly across the scene.

globalIlluminationColor

Part of setGlobalIllumination. Controls the tint of the ambient light (linear color, [0,1] per channel).

ambientLight2DIntensity

Part of setAmbientLight2D. Controls the brightness of the 2D ambient light applied to 2D-lit objects. Defaults to `1.0` (fully lit), so a scene looks unchanged until you dim it.

ambientLight2DColor

Part of setAmbientLight2D. Controls the tint of the 2D ambient light.

shadow2DQuality

- *Setter:* `void setShadow2DQuality(ShadowQuality quality)`
- *Getter:* `ShadowQuality getShadow2DQuality() const`

Filter quality of 2D light shadows (PCF taps along the 1D polar shadow map). More taps smooth the **same** penumbra width (set per light by Light2D — shadowSoftness (p. 432)), removing banding on wide penumbras. Changing it takes effect immediately (no shader rebuild).

- **NONE** — 1 tap, no filtering (hard edges regardless of softness)
- **LOW** — 5 taps (default)
- **MEDIUM** — 9 taps
- **HIGH** — 13 taps

C++

```
scene.setShadow2DQuality(ShadowQuality::MEDIUM);
```

Lua

```
scene.shadow2DQuality = ShadowQuality.MEDIUM
```

gravity2D

- *Setters:* `void setGravity2D(Vector2 gravity)`, `void setGravity2D(float x, float y)`
- *Getter:* `Vector2 getGravity2D() const`

Gravity of the **2D physics world** (Box2D), in meters per second squared. Affects every dynamic Body2D (p. 541) in the scene, scaled per body by gravityScale (p. 545). Independent from gravity3D. Changing it at runtime does not wake sleeping bodies. Forwards to the scene's PhysicsSystem (p. 567). Default `(0, -9.81)`.

C++

```
scene.setGravity2D(Vector2(0, -20));
```

Lua

```
scene.gravity2D = Vector2(0, -20)
```

gravity3D

- *Setters:* `void setGravity3D(Vector3 gravity)`, `void setGravity3D(float x, float y, float z)`
- *Getter:* `Vector3 getGravity3D() const`

Gravity of the **3D physics world** (Jolt), in meters per second squared. Affects every dynamic Body3D (p. 551) in the scene, scaled per body by gravityFactor (p. 554). Independent from gravity2D. Changing it at runtime does not wake sleeping bodies. Forwards to the scene's PhysicsSystem (p. 567). Default `(0, -9.81, 0)`.

C++

```
scene.setGravity3D(Vector3(0, -3.7f, 0)); // Mars
```

Lua

```
scene.gravity3D = Vector3(0, -3.7, 0) -- Mars
```

Shader parameters

- `void setShaderParameter(const std::string& collection, const std::string& name, value)`, where value is a `float`, `Vector2`, `Vector3` or `Vector4`
- `Vector4 getShaderParameter(const std::string& collection, const std::string& name) const`
- `bool clearShaderParameter(const std::string& collection, const std::string& name)`
- `void setShaderParameterDefaults(const ShaderUniformValues& values)` (C++ only)
- `const ShaderUniformValues& getShaderParameterDefaults() const` (C++ only)

Scene values that every shader declaring `#pragma scene <type> Collection.name` reads, such as the `Wind.velocity` of `offsets/wind.glsl`. `setShaderParameter` sets a runtime value: it overrides the one saved with the scene and is never saved, and clearing it brings the saved one back. The saved values are edited in the scene's **Shader Parameters**; the editor and exported scenes write the whole list, keyed by `Collection.name`, with `setShaderParameterDefaults`. `getShaderParameter` returns the runtime value, else the saved one, else zero, in which case a shader reads its declared default. Changing a value rebuilds no shader. Collection and name take letters, digits and underscores. See Shader Effects (p. 282).

C++

```
scene.setShaderParameter("Wind", "velocity", Vector3(4.0f, 0, -3.0f)); // 5 m/s
scene.setShaderParameter("Clip", "height", 2.5f);
```

Lua

```
scene:setShaderParameter("Wind", "velocity", Vector3(12, 0, 0))
scene:clearShaderParameter("Wind", "velocity")
```

View shader parameters

- *Setters:* `void setViewShaderParameter(Entity camera, collection, name, value)`, `void setMainViewShaderParameter(collection, name, value)`
- *Getters:* `Vector4 getViewShaderParameter(Entity camera, collection, name) const`, `Vector4 getMainViewShaderParameter(collection, name) const`
- *Removers:* `bool clearViewShaderParameter(Entity camera, collection, name)`, `bool clearMainViewShaderParameter(collection, name)`

Runtime values of `#pragma view <type> Collection.name` parameters, which belong to one camera, such as the target of `coverage/see_through.glsl`. The main view values go to whichever camera renders the scene. A camera reads its own runtime value, then the main view's while it renders the scene, then the value saved on its CameraComponent (p. 670). See Shader Effects (p. 283).

C++

```
scene.setMainViewShaderParameter("SeeThrough", "target", player.getWorldPosition() + Vector3(0, 0.9f,
0));
scene.setMainViewShaderParameter("SeeThrough", "radius", 2.5f);
```

Lua

```
scene:setMainViewShaderParameter("SeeThrough", "target", player:getWorldPosition() + Vector3(0, 0.9,
0))
scene:setViewShaderParameter(minimapCamera.entity, "SeeThrough", "radius", 0)
```

ssaoEnabled

- *Setter:* `void setSSAOEnabled(bool enabled)`
- *Getter:* `bool isSSAOEnabled() const`

Enables screen-space ambient occlusion. SSAO darkens the ambient/indirect lighting (IBL or global illumination) in creases and contact areas; it does not affect direct light. Toggling it recompiles lit mesh shaders. Applies to the main camera; terrain is currently excluded.

ssaoRadius

- *Setter:* `void setSSAORadius(float radius)`
- *Getter:* `float getSSAORadius() const`

View-space sampling radius (world units). Larger values gather occlusion from farther surfaces (broader, softer AO); smaller values keep it to tight contact creases.

ssaoIntensity

- *Setter:* `void setSSAOIntensity(float intensity)`
- *Getter:* `float getSSAOIntensity() const`

Strength of the effect, applied as an exponent on the occlusion factor — higher values darken occluded areas more.

ssaoBias

- *Setter:* `void setSSAOBias(float bias)`
- *Getter:* `float getSSAOBias() const`

View-space depth bias that prevents self-occlusion artifacts (acne) on flat surfaces. Increase slightly if flat areas appear dirty; keep small to preserve fine contact detail.

ssaoHalfResolution

- *Setter:* `void setSSAOHalfResolution(bool halfResolution)`
- *Getter:* `bool isSSAOHalfResolution() const`

Computes and blurs the occlusion at half the width and height, for about a quarter of the cost, and lit meshes upsample it. The depth pre-pass stays at full size. AO comes out a little softer. Default `false`.

ssaoDebug

- *Setter:* `void setSSAODebug(bool debug)`
- *Getter:* `bool isSSAODebug() const`

Debug aid: when enabled, lit meshes output the raw screen-space AO buffer as grayscale instead of their shaded color, so you can inspect and tune the occlusion directly. Not serialized.

ssrEnabled

- *Setter:* `void setSSREnabled(bool ssrEnabled)`
- *Getter:* `bool isSSREnabled() const`

Enables screen-space reflections. SSR reflects on-screen geometry by marching the camera depth/G-buffer, and where it finds a hit it *replaces* the surface's environment reflection (the IBL sky or a reflection probe) rather than adding to it, falling back to that reflection where the ray misses. Requires a framebuffer destination (editor viewport, render-to-texture camera, or engine framebuffer) and applies to the main camera. Toggling it reloads meshes to build the G-buffer shaders.

ssrMaxDistance

- *Setter:* `void setSSRMaxDistance(float maxDistance)`
- *Getter:* `float getSSRMaxDistance() const`

Maximum reflection ray length in view-space units. Longer rays catch more distant reflections at higher cost. Default `8.0`.

---

ssrThickness

- *Setter:* `void setSSRThickness(float thickness)`
- *Getter:* `float getSSRThickness() const`

Depth-compare tolerance (view-space units) for accepting a ray hit. Smaller is stricter; larger fills gaps but can smear at object contacts. Default `0.5`.

---

ssrMaxSteps

- *Setter:* `void setSSRMaxSteps(int maxSteps)`
- *Getter:* `int getSSRMaxSteps() const`

Linear march sample count. Higher gives sharper, longer reflections (and helps thin contacts register) at more cost. Default `48`.

---

ssrIntensity

- *Setter:* `void setSSRIntensity(float intensity)`
- *Getter:* `float getSSRIntensity() const`

Overall reflection strength multiplier applied in the composite. Default `1.0`.

---

ssrBlur

- *Setter:* `void setSSRBlur(float blur)`
- *Getter:* `float getSSRBlur() const`

Glossy blur amount in `[0..1]`. `0` keeps mirror-sharp reflections; higher values blur the reflection in proportion to each surface's roughness. Default `0.0`.

---

ssrDebugMode

- *Setter:* `void setSSRDebugMode(int mode)`
- *Getter:* `int getSSRDebugMode() const`

Debug visualization of the SSR G-buffer, rendered full-screen: `0` off, `1` reflection buffer, `2` normal, `3` roughness, `4` metallic, `5` albedo, `6` IBL specular. Not serialized.

---

fixedResolutionEnabled

- *Setter:* `void setFixedResolutionEnabled(bool fixedResolutionEnabled)`
- *Getter:* `bool isFixedResolutionEnabled() const`

Renders the main camera into an internal buffer of fixedResolutionWidth × fixedResolutionHeight and upscales the result to the view rect, instead of rendering at the window's native resolution. Takes effect only when this scene is the Engine main scene; scenes added as layers always render at native resolution. Letterboxing, input mapping, and object coordinates keep following the canvas and scaling mode. In the

editor, the fixed resolution is applied during play mode only. Toggling at runtime rebuilds the scene's render pipelines, which may cause a brief hitch — prefer changing the size instead. See Multiple Resolutions (p. 289).

C++

```
scene.setFixedResolutionSize(320, 180);
scene.setFixedResolutionFilter(TextureFilter::NEAREST);
scene.setFixedResolutionEnabled(true);
```

Lua

```
scene:setFixedResolutionSize(320, 180)
scene.fixedResolutionFilter = TextureFilter.NEAREST
scene.fixedResolutionEnabled = true
```

fixedResolutionWidth, fixedResolutionHeight

- *Setters:* `void setFixedResolutionWidth(unsigned int width)`, `void setFixedResolutionHeight(unsigned int height)`, `void setFixedResolutionSize(unsigned int width, unsigned int height)`
- *Getters:* `unsigned int getFixedResolutionWidth() const`, `unsigned int getFixedResolutionHeight() const`

The internal render resolution in pixels used when fixedResolutionEnabled is on. Can be changed at any time — the internal buffer is recreated on the next frame with no interruption, which makes render-scale options and dynamic resolution scaling cheap. Keep the aspect ratio equal to the canvas aspect ratio to avoid non-square pixels. Defaults `640 × 360`.

fixedResolutionFilter

- *Setter:* `void setFixedResolutionFilter(TextureFilter filter)`
- *Getter:* `TextureFilter getFixedResolutionFilter() const`

Sampling filter used when the fixed-resolution image is upscaled to the view rect: `TextureFilter::NEAREST` (default) keeps hard pixel edges for a pixel-art look; `TextureFilter::LINEAR` interpolates smoothly, which suits fixed resolution used purely as a performance measure.

defaultMeshShader

- *Setter:* `void setDefaultMeshShader(const std::string& path)`
- *Getter:* `const std::string& getDefaultMeshShader() const`

Scene-wide custom shader for Mesh components. The value is a project-relative base path to a forked shader (for example `"shaders/myMesh"`, resolving to `.vert` / `.frag`), or `"a.vert|b.frag"` for separately named files. Every Mesh whose own custom shader is empty uses it; an empty string (default) means the engine built-in. A shader assigned on the component always takes priority. Changing the value reloads the affected meshes. The fork's uniform values can be set for the whole scene with setDefaultShaderUniform. See Custom Shaders — Scene default shaders (p. 146).

C++

```
scene.setDefaultMeshShader("shaders/toon");
```

Lua

```
scene.defaultMeshShader = "shaders/toon"
```

defaultUIShader

- *Setter:* `void setDefaultUIShader(const std::string& path)`
- *Getter:* `const std::string& getDefaultUIShader() const`

Scene-wide custom shader for UI components. Same semantics as defaultMeshShader.

defaultSkyShader

- *Setter:* `void setDefaultSkyShader(const std::string& path)`
- *Getter:* `const std::string& getDefaultSkyShader() const`

Scene-wide custom shader for the Sky component. Same semantics as defaultMeshShader.

defaultPointsShader

- *Setter:* `void setDefaultPointsShader(const std::string& path)`
- *Getter:* `const std::string& getDefaultPointsShader() const`

Scene-wide custom shader for Points components. Same semantics as defaultMeshShader.

defaultLinesShader

- *Setter:* `void setDefaultLinesShader(const std::string& path)`
- *Getter:* `const std::string& getDefaultLinesShader() const`

Scene-wide custom shader for Lines components. Same semantics as defaultMeshShader.

defaultWaterShader

- *Setter:* `void setDefaultWaterShader(const std::string& path)`
- *Getter:* `const std::string& getDefaultWaterShader() const`

Scene-wide custom shader for Water (p. 496) components. Same semantics as defaultMeshShader. It covers the water surface only; the underwater fade has no scene default and is forked per water with customUnderwaterShader (p. 501).

postProcessPasses

- *Setter:* `void setPostProcessPasses(const std::vector<PostProcessPass>& passes)`
- *Getter:* `const std::vector<PostProcessPass>& getPostProcessPasses() const`

Ordered chain of fullscreen post-process passes run over the scene's finished image, on the main camera only. Each entry is a PostProcessPass (p. 715) with the fork to run, an enable flag, and the values of the parameters it declares. In Lua the property is a 1-based sequence of `PostProcessPass` values (a copy — assign it back to apply changes); build the chain as a sequence (a `{...}` literal or `table.insert`), whose array order is the pass order, and note that `pass_index` arguments such as setPostProcessUniform count from 0.

Setting the chain rebuilds the passes on the next frame, unless only uniform values changed (same shaders and enable flags), in which case the compiled chain is kept and just the values are rewritten. For a single value or pass on a live chain, setPostProcessUniform and setPostProcessPassEnabled are the direct route. See Custom Shaders — Post-process passes (p. 149).

C++

```cpp
PostProcessPass sharpen;
sharpen.shader = "shaders/sharpen";
sharpen.setUniform("amount", 0.5f);

scene.setPostProcessPasses({sharpen});
```

Lua

```lua
local sharpen = PostProcessPass()
sharpen.shader = "shaders/sharpen"
sharpen:setUniform("amount", 0.5)

scene.postProcessPasses = {sharpen}
```

enableUIEvents

- *Setter:* `void setEnableUIEvents(UIEventState enableUIEvents)`
- *Getter:* `UIEventState getEnableUIEvents() const`

Per-scene override for UI event routing. See UIEventState.

Method details

load

- `void load()`

Initializes the scene subsystems. Called automatically by the engine when the scene is added via `Engine::setScene` or `Engine::addSceneLayer`. Only call manually when managing scenes outside the engine lifecycle.

destroy

- `void destroy()`

Tears down all subsystems and destroys every entity in the scene. Called automatically by the engine when the scene is removed from the stack.

draw

- `void draw()`

Triggers a render pass for this scene. Normally called by the engine each frame; use only for custom render pipelines.

update

- `void update(double dt)`

Runs one variable-step update for all subscribed systems. `dt` is the frame delta in seconds. Normally called by the engine.

fixedUpdate

- `void fixedUpdate(double dt)`

Runs one fixed-step update for physics and other time-sensitive systems. `dt` equals Engine::updateTime (p. 365). Normally called by the engine.

setCamera

- `void setCamera(Camera* camera)`
- `void setCamera(Entity camera)`

Sets the active camera for this scene. Only one camera can be active at a time. Pass the `Camera` object or its underlying `Entity` handle.

C++

```
Camera cam(&scene);
cam.activate();           // equivalent shortcut
// or:
scene.setCamera(&cam);
```

Lua

```
local cam = Camera(scene)
scene:setCamera(cam)
```

getCamera

- `Entity getCamera() const`

Returns the entity handle of the currently active camera.

setViewCamera

- `void setViewCamera(Entity camera)`

Draws the scene from `camera` instead of the active camera, while getCamera keeps returning the active one, so scripts, audio, and UI events are unaffected. Effects tied to the main camera (shadows, SSAO, SSR, post-processing, terrain and mesh detail) follow the view camera. Pass `NULL_ENTITY` to draw from the active camera again. The editor uses it for its detached camera (p. 89).

getRenderCamera

- `Entity getRenderCamera() const`

Returns the camera the scene is drawn from: the view camera when one is set, otherwise the active camera.

setBackgroundColor

- `void setBackgroundColor(Vector4 color)`
- `void setBackgroundColor(float red, float green, float blue)`
- `void setBackgroundColor(float red, float green, float blue, float alpha)`

Sets the clear color used before rendering the scene each frame.

C++

```
scene.setBackgroundColor(0.05f, 0.05f, 0.1f, 1.0f);
```

Lua

```
scene:setBackgroundColor(0.05, 0.05, 0.1, 1.0)
```

setLightState

- `void setLightState(LightState state)`
- `LightState getLightState() const`

Overrides automatic lighting detection. See LightState.

setGlobalIllumination

- `void setGlobalIllumination(float intensity, Vector3 color)`
- `void setGlobalIllumination(float intensity)`
- `void setGlobalIllumination(Vector3 color)`

Sets the ambient (global illumination) light for the scene. Intensity scales brightness; color tints the light. Flat-shaded or unlit materials are not affected.

C++

```
scene.setGlobalIllumination(0.3f, Vector3(1.0f, 0.95f, 0.9f));
```

Lua

```
scene:setGlobalIllumination(0.3, Vector3(1.0, 0.95, 0.9))
```

setAmbientLight2D

- `void setAmbientLight2D(float intensity, Vector3 color)`
- `void setAmbientLight2D(float intensity)`
- `void setAmbientLight2D(Vector3 color)`

Sets the ambient light for the **2D lighting path** (see Light2D (p. 430)). Every Light2D (p. 430) adds on top of this base level, so dim the ambient to make 2D lights visible. Separate from setGlobalIllumination, which drives the 3D PBR ambient.

C++

```
scene.setAmbientLight2D(0.15f, Vector3(0.9f, 0.9f, 1.0f)); // dark, blueish night
```

Lua

```
scene:setAmbientLight2D(0.15, Vector3(0.9, 0.9, 1.0)) -- dark, blueish night
```

enableUIEvents

- `void enableUIEvents()`

Shorthand for `setEnableUIEvents(UIEventState::ENABLED)`.

isEnableUIEvents

- `bool isEnableUIEvents() const`

Returns `true` if UI events are enabled for this scene (either via explicit `ENABLED` state or engine default).

canReceiveUIEvents

- `bool canReceiveUIEvents()`

Returns `true` if this scene is currently the topmost scene that is able to receive UI pointer events. The engine calls this internally to route events to the correct scene layer.

updateCameraSize

- `void updateCameraSize()`

Recalculates the active camera's projection to match the current canvas size. Called automatically when the canvas changes; call manually after resizing the viewport from script.

setDefaultShaderUniform

- `void setDefaultShaderUniform(ShaderType type, const std::string& name, const Vector4& value)`
- `void setDefaultShaderUniform(ShaderType type, const std::string& name, const Vector3& value)`
- `void setDefaultShaderUniform(ShaderType type, const std::string& name, const Vector2& value)`
- `void setDefaultShaderUniform(ShaderType type, const std::string& name, float value)`
- `Vector4 getDefaultShaderUniform(ShaderType type, const std::string& name) const`
- `bool removeDefaultShaderUniform(ShaderType type, const std::string& name)`
- `void setDefaultShaderUniforms(ShaderType type, const ShaderUniformValues& values)` (C++ only)
- `const ShaderUniformValues& getDefaultShaderUniforms(ShaderType type) const` (C++ only)

Scene values for the parameters the scene default shader of `type` declares (see ShaderType), by name. Every component that inherits that default reads them for the parameters it does not set itself; a value set on the component with setShaderUniform (p. 445) wins. Setting a value only rewrites the uploaded blocks, so it can run every frame. A narrower value leaves the remaining components at zero, and a parameter with no value on either side reads its declared default. A type without a scene default is ignored. `getDefaultShaderUniform` returns the scene's stored value, zero when unset; `removeDefaultShaderUniform` drops it and returns whether it existed. The values stay when the default shader changes, so a fork with the same parameter names picks them up. See Custom Shaders — Scene default shaders (p. 146).

C++

```
scene.setDefaultShaderUniform(ShaderType::MESH, "exposure", 0.5f);
scene.setDefaultShaderUniform(ShaderType::MESH, "rimColor", Vector4(0.2f, 0.6f, 1.0f, 1.0f));
```

Lua

```
scene:setDefaultShaderUniform(ShaderType.MESH, "exposure", 0.5)
scene:setDefaultShaderUniform(ShaderType.MESH, "rimColor", Vector4(0.2, 0.6, 1, 1))
```

setPostProcessUniform

- `void setPostProcessUniform(unsigned int index, const std::string& name, const Vector4& value)`
- `void setPostProcessUniform(unsigned int index, const std::string& name, const Vector3& value)`
- `void setPostProcessUniform(unsigned int index, const std::string& name, const Vector2& value)`
- `void setPostProcessUniform(unsigned int index, const std::string& name, float value)`
- `Vector4 getPostProcessUniform(unsigned int index, const std::string& name) const`
- `bool removePostProcessUniform(unsigned int index, const std::string& name)`

Sets one parameter of the pass at `index` (0 = first pass of postProcessPasses), by the name its shader declares with `#pragma param`. Only the uploaded block is rewritten, and the chain is not rebuilt, so it can run every frame. A narrower value leaves the remaining components at zero; a parameter with no value reads its declared default. An index past the chain logs an error and is ignored. `getPostProcessUniform` returns the stored value, zero when unset; `removePostProcessUniform` drops it (the parameter reads its declared default again) and returns whether it existed.

C++

```
scene.setPostProcessUniform(0, "amount", 0.8f);
scene.setPostProcessUniform(0, "tintColor", Vector4(1.0f, 0.9f, 0.8f, 1.0f));
```

Lua

```
scene:setPostProcessUniform(0, "amount", 0.8)
scene:setPostProcessUniform(0, "tintColor", Vector4(1, 0.9, 0.8, 1))
```

setPostProcessPassEnabled

- `void setPostProcessPassEnabled(unsigned int index, bool enabled)`
- `bool isPostProcessPassEnabled(unsigned int index) const`

Turns the pass at `index` on or off. A change rebuilds the chain on the next frame (the compiled shader is kept, so it is cheap, but not something to toggle every frame). An index past the chain logs an error; `isPostProcessPassEnabled` returns `false` for it.

C++

```
scene.setPostProcessPassEnabled(1, false);
```

Lua

```
scene:setPostProcessPassEnabled(1, false)
```

removeSubscriptionsByTag

- `void removeSubscriptionsByTag(const std::string& substring)`

Removes all event subscriptions whose tag string contains `substring`. Used to clean up callbacks belonging to a destroyed script or component.

System

Description

`System` is a virtual platform-abstraction layer. It provides screen dimensions, file-system paths, virtual keyboard control, fullscreen management, and persistent key-value storage. You access it via `System::instance()` in C++ or the global `System` table in Lua.

Ads, in-app purchases and web portal SDKs have their own classes: AdMob (p. 610), InAppPurchase (p. 618) and WebPortal (p. 626). See Monetization (p. 349) for the `System` methods they replaced.

Each target platform provides its own concrete `System` subclass; the engine injects the correct implementation at startup.

C++

```cpp
#include "Doriax.h"
using namespace doriax;

int screenW = System::instance().getScreenWidth();
int screenH = System::instance().getScreenHeight();
Log::print("Screen: %dx%d", screenW, screenH);
```

Lua

```lua
local w = System.getScreenWidth()
local h = System.getScreenHeight()
Log.print("Screen: " .. w .. "x" .. h)
```

Methods

Returns	Name	Languages
int	getScreenWidth	C++ | Lua
int	getScreenHeight	C++ | Lua
int	getSampleCount	C++
bool	isTouchDevice	C++ | Lua
void	showVirtualKeyboard	C++ | Lua
void	hideVirtualKeyboard	C++ | Lua
bool	isFullscreen	C++ | Lua
void	requestFullscreen	C++ | Lua
void	exitFullscreen	C++ | Lua
bool	isWindowMaximized	C++ | Lua
void	maximizeWindow	C++ | Lua
void	restoreWindow	C++ | Lua
void	setWindowSize	C++ | Lua
bool	isWindowResizable	C++ | Lua
void	setWindowResizable	C++ | Lua
void	setWindowTitle	C++ | Lua
void	quit	C++ | Lua
char	getDirSeparator	C++ | Lua
std::string	getAssetPath	C++ | Lua
std::string	getUserDataPath	C++ | Lua
std::string	getLuaPath	C++ | Lua
std::string	getShaderPath	C++ | Lua

Returns	Name	Languages
std::string	getImportPath	C++
bool	getBoolForKey	C++ | Lua (UserSettings (p. 707))
int	getIntegerForKey	C++ | Lua (UserSettings (p. 707))
long	getLongForKey	C++ | Lua (UserSettings (p. 707))
float	getFloatForKey	C++ | Lua (UserSettings (p. 707))
double	getDoubleForKey	C++ | Lua (UserSettings (p. 707))
std::string	getStringForKey	C++ | Lua (UserSettings (p. 707))
void	setBoolForKey	C++ | Lua (UserSettings (p. 707))
void	setIntegerForKey	C++ | Lua (UserSettings (p. 707))
void	setLongForKey	C++ | Lua (UserSettings (p. 707))
void	setFloatForKey	C++ | Lua (UserSettings (p. 707))
void	setDoubleForKey	C++ | Lua (UserSettings (p. 707))
void	setStringForKey	C++ | Lua (UserSettings (p. 707))
void	removeKey	C++ | Lua (UserSettings (p. 707))
AdMobBackend*	getAdMobBackend	C++
InAppPurchaseBackend*	getInAppPurchaseBackend	C++
WebPortalBackend*	getWebPortalBackend	C++

Method details

getScreenWidth / getScreenHeight

- `virtual int getScreenWidth() = 0`
- `virtual int getScreenHeight() = 0`

Physical screen (window) dimensions in pixels. These may differ from Engine::canvasWidth / canvasHeight (p. 366) when a scaling mode other than `NATIVE` is active.

C++

```
int w = System::instance().getScreenWidth();
int h = System::instance().getScreenHeight();
```

Lua

```
local w = System.getScreenWidth()
local h = System.getScreenHeight()
```

getSampleCount

- `virtual int getSampleCount()`

Returns the MSAA sample count configured for the current graphics backend (e.g. 1, 2, or 4). Useful when creating render targets that must match the main framebuffer.

isTouchDevice

- `virtual bool isTouchDevice()`

Returns `true` when touch is the main input, as on phones and tablets. Android and iOS always report it. On the web the browser decides from its primary pointer, so a phone reports `true` and a laptop with a touch screen and a mouse reports `false`. Desktop platforms report `false`, and the editor reports `true` while touch simulation (p. 88) is on.

Use it to decide whether to start with on-screen controls. The player can still connect a keyboard or touch a laptop screen, so following the last input used as well is more reliable; see On-screen controls (p. 232).

C++

```
bool showControls = System::instance().isTouchDevice();
```

Lua

```
local showControls = System.isTouchDevice()
```

showVirtualKeyboard

- `virtual void showVirtualKeyboard(std::wstring text = L"")`

Displays the platform's on-screen keyboard with `text` as the initial input buffer. Has no effect on desktop platforms. On iOS/Android, raises the software keyboard and fires `Engine::onCharInput` for each key typed.

C++

```
System::instance().showVirtualKeyboard(L"Enter name");
```

Lua

```
System.showVirtualKeyboard("Enter name")
```

hideVirtualKeyboard

- `virtual void hideVirtualKeyboard()`

Dismisses the on-screen keyboard.

isFullscreen / requestFullscreen / exitFullscreen

- `virtual bool isFullscreen()`
- `virtual void requestFullscreen()`
- `virtual void exitFullscreen()`

Query and control fullscreen mode. On desktop platforms, `requestFullscreen()` typically toggles borderless fullscreen; on web, it requests the Fullscreen API. Has no effect on mobile (always fullscreen).

Whether a desktop build *starts* fullscreen is a project setting (see Project Workflow — Window (p. 54)); `exitFullscreen()` restores the window size configured there.

C++

```
if (!System::instance().isFullscreen()) {
    System::instance().requestFullscreen();
}
```

Lua

```
if not System.isFullscreen() then
    System.requestFullscreen()
end
```

Window control

- `virtual bool isWindowMaximized()`
- `virtual void maximizeWindow()`
- `virtual void restoreWindow()`
- `virtual void setWindowSize(int width, int height)`

- `virtual bool isWindowResizable()`
- `virtual void setWindowResizable(bool resizable)`
- `virtual void setWindowTitle(const std::string& title)`

Control the OS window of desktop builds at runtime. `maximizeWindow()` / `restoreWindow()` toggle the maximized state, `setWindowSize()` resizes the window in screen coordinates (while fullscreen it instead updates the size restored by `exitFullscreen()`), and `setWindowTitle()` changes the title-bar text. The initial values come from the project's Window settings (p. 54).

Note that `setWindowSize()` takes *window* coordinates while getScreenWidth/getScreenHeight report *framebuffer* pixels — on HiDPI/scaled displays these differ by the content scale, so the two do not round-trip 1:1.

Platform support: fully implemented on GLFW-based desktop builds (generated desktop projects and Linux exports). Windows/macOS Sokol exports support only `setWindowTitle()` — the other calls are no-ops there. On web, `setWindowTitle()` sets the browser tab title and the rest are no-ops. All are no-ops on mobile.

C++

```cpp
System::instance().setWindowTitle("My Game — Level 2");
System::instance().setWindowSize(1600, 900);
if (!System::instance().isWindowMaximized()) {
    System::instance().maximizeWindow();
}
```

Lua

```lua
System.setWindowTitle("My Game — Level 2")
System.setWindowSize(1600, 900)
if not System.isWindowMaximized() then
    System.maximizeWindow()
end
```

quit

- `virtual void quit()`

Closes the application. The shutdown happens after the current frame ends, so it is safe to call from a button callback or any other script code, and `Engine::onShutdown` (p. 373) still runs before the app goes away.

Platform support: desktop builds close the window, web stops the render loop and shuts the engine down (writing pending user data to IndexedDB first — the browser tab itself stays open), and Android finishes the activity. On iOS it is a no-op and logs a warning, since apps are not supposed to terminate themselves.

Running a scene with Play in the editor stops play instead of closing the editor, so a quit button can be tested there.

C++

```cpp
void UI_MainMenu::onButtonClickQuit() {
    System::instance().quit();
}
```

Lua

```lua
function UI_MainMenu:onButtonClickQuit()
    System.quit()
end
```

getDirSeparator

- `virtual char getDirSeparator()`

Returns the platform path directory separator: `'/'` on Unix/macOS/iOS/Android/Web, `'\\'` on Windows.

getAssetPath

- `virtual std::string getAssetPath()`

Returns the root path for read-only assets bundled with the project: the project's assets directory in the editor, and the `assets` folder beside the executable in an exported build. Prepend this to asset file names when opening files directly from C++. In Lua and the high-level C++ loaders (`setTexture`, etc.), paths are automatically resolved relative to the asset root.

C++

```
std::string path = System::instance().getAssetPath() + "textures/sprite.png";
```

getUserDataPath

- `virtual std::string getUserDataPath()`

Returns a writable per-user data directory (Documents on iOS, internal storage on Android, `%AppData%` on Windows, `~/.local/share` on Linux). Use for save files and configuration.

C++

```
std::string savePath = System::instance().getUserDataPath() + "save.dat";
```

getLuaPath

- `virtual std::string getLuaPath()`

Returns the base path prepended when Lua `require()` looks for script modules and when a `ScriptComponent` entry is loaded: the project's Lua directory in the editor, and the `lua` folder beside the executable in an exported build.

getShaderPath

- `virtual std::string getShaderPath()`

Returns the directory holding compiled `.sdat` shader binaries, and the root for `shader://` paths: the project's `shaders` folder in the editor, and `assets/shaders` beside the executable in an exported build. The folder name is fixed, not configurable.

getImportPath

- `virtual std::string getImportPath()`

Returns the directory holding the glTF conversions of FBX models: the project's `.doriax/imported` folder in the editor. It is empty in an exported build, where each conversion sits beside its FBX's path as `<name>.fbx.glb`. See 3D Graphics — FBX models (p. 251).

Persistent key–value storage

The system exposes a simple cross-platform persistent store backed by `NSUserDefaults` (iOS/macOS), `SharedPreferences` (Android), `localStorage` (Web), and a JSON file (desktop/Linux/Windows).

On the web, keys are stored as `<id>/<key>`, where `<id>` is the application identifier (p. 56), so games on the same site keep separate values. While the identifier is the default `com.yourcompany.project`, the project name is used instead, so set one before publishing. Characters other than ASCII letters, digits, `.`, and `-` are written as `_` and their hex code, so `My Game` becomes `My_20Game`. Keys saved by builds before v0.8.1 are still read.

Getters return the stored value or `defaultValue` if the key does not exist:

Method	Type
`getBoolForKey(key, defaultValue)`	bool
`getIntegerForKey(key, defaultValue)`	int
`getLongForKey(key, defaultValue)`	long
`getFloatForKey(key, defaultValue)`	float
`getDoubleForKey(key, defaultValue)`	double
`getStringForKey(key, defaultValue)`	string

Setters write or update a value:

Method	Type
`setBoolForKey(key, value)`	bool
`setIntegerForKey(key, value)`	int
`setLongForKey(key, value)`	long
`setFloatForKey(key, value)`	float
`setDoubleForKey(key, value)`	double
`setStringForKey(key, value)`	string

removeKey deletes a stored entry:

- `virtual void removeKey(const char* key)`

C++

```
System& sys = System::instance();

// Save high score
sys.setIntegerForKey("highScore", score);

// Load with default 0
int best = sys.getIntegerForKey("highScore", 0);
```

Lua

```
-- Preferences are exposed to Lua through UserSettings (not System).
UserSettings.setIntegerForKey("highScore", score)
local best = UserSettings.getIntegerForKey("highScore", 0)
```

Service backends

- `virtual AdMobBackend* getAdMobBackend()`
- `virtual InAppPurchaseBackend* getInAppPurchaseBackend()`
- `virtual WebPortalBackend* getWebPortalBackend()`

How the AdMob (p. 610), InAppPurchase (p. 618) and WebPortal (p. 626) classes reach the platform's SDKs. Each returns the platform's implementation, or `nullptr` where the platform or the build has none, which the classes report as "not available". Games call those classes rather than these methods; a port to a new platform overrides them to plug in its own SDKs.

Objects

Bone

Inherits: Object (p. 455)
C++ type: `Bone`

Skeletal bone in a hierarchy. On GLTF files that import the full node tree (p. 249), the bone is the same entity as that glTF joint node (it also has a `Transform`). `Model::getBone` finds it by joint name or by glTF node index.

Methods

Returns	Name	Languages
BoneComponent (p. 668)	`getBoneComponent`	Lua

Camera

Description

`Camera` defines the viewpoint from which the scene is rendered. It supports both orthographic (2D) and perspective (3D) projection, a look-at target, manual navigation helpers (walk, slide, orbit), and render-to-texture via an internal `Framebuffer`.

Inherits: Object (p. 455) → EntityHandle (p. 711)

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Camera cam(&scene);
cam.setPerspective(60.0f, 1.333f, 0.1f, 1000.0f);
cam.setPosition(0.0f, 2.0f, 10.0f);
cam.setTarget(Vector3(0.0f, 0.0f, 0.0f));
cam.activate();
```

Lua

```lua
local cam = Camera(scene)
cam:setPerspective(60, 1.333, 0.1, 1000)
cam.position = Vector3(0, 2, 10)
cam.target = Vector3(0, 0, 0)
cam:activate()
```

Properties

Type	Name	Default	Languages
CameraType	type	`ORTHOGRAPHIC`	C++ | Lua
bool	autoResize	`true`	C++ | Lua
float	nearClip	`0.1`	C++ | Lua
float	farClip	`1000.0`	C++ | Lua
float	leftClip	—	C++ | Lua
float	rightClip	—	C++ | Lua
float	bottomClip	—	C++ | Lua
float	topClip	—	C++ | Lua
float	aspect	—	C++ | Lua
float	yfov	`60.0°`	C++ | Lua
Vector3	target	`(0,0,0)`	C++ | Lua
Vector3	up	`(0,1,0)`	C++ | Lua
bool	renderToTexture	`false`	C++ | Lua
bool	transparentSort	`true`	C++ | Lua
bool	depthTest	`true`	C++ | Lua

Methods

Returns	Name	Languages
void	activate	C++ | Lua
void	setOrtho	C++ | Lua
void	setPerspective	C++ | Lua
void	setAutoResize	C++

Returns	Name	Languages
bool	isAutoResize	C++
void	setNearClip	C++
float	getNearClip	C++
void	setFarClip	C++
float	getFarClip	C++
void	setLeftClip	C++
float	getLeftClip	C++
void	setRightClip	C++
float	getRightClip	C++
void	setBottomClip	C++
float	getBottomClip	C++
void	setTopClip	C++
float	getTopClip	C++
void	setAspect	C++
float	getAspect	C++
void	setYFov	C++
float	getYFov	C++
void	setType	C++ | Lua
CameraType	getType	C++
void	setTarget	C++ | Lua
Vector3	getTarget	C++
void	disableTarget	C++ | Lua
bool	isUsingTarget	C++ | Lua
void	setUp	C++ | Lua
Vector3	getUp	C++ | Lua
Vector3	getDirection	C++ | Lua
Vector3	getRight	C++ | Lua
Vector3	getWorldTarget	C++ | Lua
Vector3	getWorldDirection	C++ | Lua
Vector3	getWorldUp	C++ | Lua
Vector3	getWorldRight	C++ | Lua
Matrix4	getViewMatrix	C++ | Lua
Matrix4	getProjectionMatrix	C++ | Lua
Matrix4	getViewProjectionMatrix	C++ | Lua
void	rotateView	C++ | Lua
void	rotatePosition	C++ | Lua
void	elevateView	C++ | Lua
void	elevatePosition	C++ | Lua
void	walkForward	C++ | Lua
void	zoom	C++ | Lua
void	slide	C++ | Lua
void	slideForward	C++ | Lua
void	slideUp	C++ | Lua
void	setRenderToTexture	C++

Returns	Name	Languages
bool	isRenderToTexture	C++
Framebuffer*	getFramebuffer	C++ | Lua
void	setFramebufferSize	C++ | Lua
void	setFramebufferFilter	C++ | Lua
void	setTransparentSort	C++
bool	isTransparentSort	C++
void	setDepthTest	C++
bool	isDepthTest	C++
Ray	screenToRay	C++ | Lua
float	getDistanceFromTarget	C++ | Lua
void	updateCamera	C++ | Lua
void	setShaderParameter	C++ | Lua
Vector4	getShaderParameter	C++ | Lua
bool	clearShaderParameter	C++ | Lua
CameraComponent (p. 670)	`getCameraComponent`	Lua

Enumerations

CameraType

- **ORTHOGRAPHIC** — Parallel projection; objects do not shrink with distance. Used for 2D games and UI.
- **PERSPECTIVE** — Frustum projection; objects appear smaller with distance. Used for 3D scenes.

Property details

type

- *Setter:* `void setType(CameraType type)`
- *Getter:* `CameraType getType() const`

Switches between orthographic and perspective projection. Setting the type does not automatically recalculate clip planes; use setOrtho or setPerspective for that.

autoResize

- *Setter:* `void setAutoResize(bool autoResize)`
- *Getter:* `bool isAutoResize() const`

When `true` (default), the projection is automatically recalculated when the canvas size changes. Disable if you manage the projection matrix manually.

nearClip / farClip

- *Setters:* `void setNearClip(float nearValue)` / `void setFarClip(float farValue)`
- *Getters:* `float getNearClip() const` / `float getFarClip() const`

Near and far clip plane distances. Objects closer than `nearClip` or further than `farClip` are not rendered. Tighten these bounds to maximise depth-buffer precision.

A perspective camera needs `nearClip` above zero and `farClip` above `nearClip`. Other values are clamped when rendering, but the getters return what you set. An orthographic camera can use a negative `nearClip` to see behind its position, like the default 2D camera (`-10` to `10`).

leftClip / rightClip / bottomClip / topClip

Orthographic frustum extents. Set via setOrtho or individually. In auto-resize mode the engine keeps these aligned to the canvas.

aspect

- *Setter:* `void setAspect(float aspect)`
- *Getter:* `float getAspect() const`

Width-to-height ratio for perspective projection. In auto-resize mode the engine updates this automatically.

yfov

- *Setter:* `void setYFov(float yfov)`
- *Getter:* `float getYFov() const`

Vertical field-of-view in degrees (when `Engine::useDegrees` is `true`). Affects how wide the perspective frustum opens.

target

- *Setter:* `void setTarget(Vector3 target)` / `void setTarget(float x, float y, float z)`
- *Getter:* `Vector3 getTarget() const`

Look-at point in local space. When set, the camera orientation is computed each frame so the camera always faces this point. Disable with disableTarget.

> **In Lua, `setTarget` only takes three numbers**
>
> Lua binds the `Vector3` form as the **`target` property** and `setTarget` as the three-float overload only. Passing a `Vector3` to `setTarget` fails with *"Error decoding argument #1: The lua object can't be cast to desired type"*.
>
> ```
> cam.target = Vector3(0, 3, 0)   -- Vector3 form
> cam:setTarget(0, 3, 0)          -- three numbers
> ```

up

- *Setter:* `void setUp(Vector3 up)` / `void setUp(float x, float y, float z)`
- *Getter:* `Vector3 getUp() const`

Local up vector used when computing the view matrix from a target. Defaults to `(0, 1, 0)`.

renderToTexture

- *Setter:* `void setRenderToTexture(bool renderToTexture)`
- *Getter:* `bool isRenderToTexture() const`

When `true`, the camera renders to an internal Framebuffer instead of the main window surface. Retrieve the framebuffer with `getFramebuffer()` and pass it to a `Mesh::setTexture` call to create render-to-texture effects.

transparentSort

- *Setter:* `void setTransparentSort(bool transparentSort)`
- *Getter:* `bool isTransparentSort() const`

Enables back-to-front depth sorting for transparent objects from this camera's viewpoint. Disable for orthographic UI cameras where order is determined by hierarchy. It only applies while depthTest is on — without a depth buffer the camera already draws in submission order.

depthTest

- *Setter:* `void setDepthTest(bool depthTest)`
- *Getter:* `bool isDepthTest() const`

When `false`, the camera ignores the depth buffer entirely — no depth test, no depth write — and composites every renderable in submission order, like a 2D canvas: what is drawn later appears on top, regardless of its distance. Distance sorting is skipped as well, so transparentSort has no effect.

Keep it `true` for 3D scenes, where geometry has to occlude by depth. A `Camera` you create defaults to `true`; the canvas camera a scene creates for itself when none is set, and the cameras the editor adds to a 2D scene, have it off, so entity order decides what covers what.

> **Instances and points follow the main camera**
> Instanced meshes and point clouds share one GPU buffer across every camera, so their batch order is fixed once per frame by the scene's active camera. A render-to-texture camera whose `depthTest` differs from the main camera's reuses that order instead of computing its own.

Method details

activate

- `void activate()`

Makes this camera the active camera in its scene. Equivalent to calling `scene.setCamera(this)`.

setOrtho

- `void setOrtho(float left, float right, float bottom, float top, float nearValue, float farValue)`

Configures the camera for orthographic projection with explicit clip planes. All objects within the box `(left, right, bottom, top, near, far)` are rendered.

C++

```
cam.setOrtho(0.0f, 800.0f, 0.0f, 600.0f, -100.0f, 100.0f);
```

Lua

```
cam:setOrtho(0, 800, 0, 600, -100, 100)
```

setPerspective

- `void setPerspective(float yfov, float aspect, float nearValue, float farValue)`

Configures the camera for perspective (3D) projection.

- **yfov** — Vertical field-of-view in degrees.
- **aspect** — Width ÷ height ratio.

- **nearValue** — Near clip plane distance, above zero (keep as large as possible).
- **farValue** — Far clip plane distance.

C++
```
cam.setPerspective(60.0f, (float)Engine::getCanvasWidth() / Engine::getCanvasHeight(), 0.1f, 500.0f);
```

Lua
```
cam:setPerspective(60, Engine.canvasWidth / Engine.canvasHeight, 0.1, 500)
```

setTarget / disableTarget / isUsingTarget

- `void setTarget(Vector3 target)` — Sets the look-at point. Activates target mode. In Lua this form is the `target` property; `setTarget` there takes three numbers.
- `void disableTarget()` — Removes the look-at constraint and allows free rotation.
- `bool isUsingTarget() const` — Returns `true` while target mode is active.

getDirection / getRight

- `Vector3 getDirection() const` — Local forward vector in camera space.
- `Vector3 getRight() const` — Local right vector in camera space.

Use `getWorldDirection()`, `getWorldUp()`, `getWorldRight()` for world-space equivalents.

getViewMatrix / getProjectionMatrix / getViewProjectionMatrix

Computed matrices ready to pass to shaders or custom render pipelines.

rotateView

- `void rotateView(float angle)`

Rotates the camera's *look direction* around the up axis by `angle` degrees (yaw). The camera position stays fixed.

rotatePosition

- `void rotatePosition(float angle)`

Orbits the camera's *position* around the current target by `angle` degrees. Useful for orbit camera controllers.

C++
```
// Orbit controller driven by mouse delta
cam.rotatePosition(mouseDelta.x * 0.3f);
cam.elevatePosition(-mouseDelta.y * 0.3f);
```

elevateView

- `void elevateView(float angle)`

Tilts the camera's look direction up or down (pitch) by `angle` degrees. Position stays fixed.

elevatePosition

- `void elevatePosition(float angle)`

Moves the camera position up or down along a sphere centered at target. Useful for orbit elevation.

walkForward

- `void walkForward(float distance)`

Moves the camera forward along its look direction by `distance` units.

zoom

- `void zoom(float distance)`

Moves the camera position along the direction toward target by `distance` units. A positive value zooms in; negative zooms out.

slide / slideForward / slideUp

- `void slide(float distance)` — Strafes along the local right vector.
- `void slideForward(float distance)` — Moves along the local forward vector.
- `void slideUp(float distance)` — Moves along the local up vector.

screenToRay

- `Ray screenToRay(float x, float y)`

Converts a screen-space position (in canvas pixels) to a world-space Ray (p. 649) for mouse picking.

C++

```cpp
// Pick on mouse click
Engine::onMouseDown.add("pick", [&](int btn, float x, float y, int mods) {
    Ray ray = cam.screenToRay(x, y);
    // test ray against physics or scene objects
});
```

Lua

```lua
Engine.onMouseDown = function(btn, x, y, mods)
    local ray = cam:screenToRay(x, y)
end
```

getDistanceFromTarget

- `float getDistanceFromTarget() const`

Returns the distance between the camera position and the current target point. Useful for zoom limits in orbit controllers.

setRenderToTexture / getFramebuffer / setFramebufferSize / setFramebufferFilter

Configure render-to-texture mode.

- `void setRenderToTexture(bool renderToTexture)` — Enable/disable off-screen rendering.
- `Framebuffer* getFramebuffer()` — Returns the internal framebuffer; valid only after `setRenderToTexture(true)`.
- `void setFramebufferSize(int width, int height)` — Overrides the framebuffer resolution (defaults to 512×512).
- `void setFramebufferFilter(TextureFilter filter)` — Sets the sampling filter for the output texture.

The size and filter are also editable in the editor's **Render Target** section on a render-to-texture camera and are saved with the scene.

C++

```
Camera rtCam(&scene);
rtCam.setPerspective(60.0f, 1.0f, 0.1f, 100.0f);
rtCam.setRenderToTexture(true);
rtCam.setFramebufferSize(512, 512);

Sprite screen(&scene);
screen.setTexture(rtCam.getFramebuffer());
screen.setSize(512, 512);
screen.createSprite();
```

updateCamera

- `void updateCamera()`

Forces an immediate recalculation of view and projection matrices. Called automatically each frame; call manually only when you need a fresh matrix outside the normal update cycle.

Shader parameters

- `void setShaderParameter(const std::string& collection, const std::string& name, const Vector4& value)`
- `Vector4 getShaderParameter(const std::string& collection, const std::string& name) const`
- `bool clearShaderParameter(const std::string& collection, const std::string& name)`

The setter also takes a `Vector3`, a `Vector2` or a `float`.

Values for the view parameters (`#pragma view`) of shader effects (p. 282) while this camera renders. `setShaderParameter` is a runtime value: it is not saved and is dropped when the scene reloads. The value saved with the camera is the one the editor shows in the camera's **Shader Parameters** section; C++ code sets saved values in CameraComponent (p. 670) `::shaderParameters`. `getShaderParameter` returns the value this camera renders with, or zero when nothing sets it; the shader then uses the default from the declaration.

These calls are shortcuts for the same functions on Scene (p. 397). Use `Scene::setMainViewShaderParameter` for values that should follow whichever camera is the main one.

C++

```
// each player camera keeps its own see-through opening
leftCamera.setShaderParameter("SeeThrough", "target", playerOne.getWorldPosition());
rightCamera.setShaderParameter("SeeThrough", "target", playerTwo.getWorldPosition());
leftCamera.setShaderParameter("SeeThrough", "radius", 1.5f);
rightCamera.setShaderParameter("SeeThrough", "radius", 1.5f);
```

Lua

```
leftCamera:setShaderParameter("SeeThrough", "target", playerOne:getWorldPosition())
leftCamera:setShaderParameter("SeeThrough", "radius", 1.5)
```

Fog

Inherits: EntityHandle (p. 711)
C++ type: `Fog`

Description

Applies atmospheric fog to a scene. Fog blends the scene's rendered colour towards a fog colour based on the depth of each fragment from the camera, simulating haze, mist, or distance-based visibility falloff.

Three fog models are supported: linear, exponential, and squared-exponential.

Properties

Type	Name	Default	Langs
FogType	type	`LINEAR`	C++ | Lua
Vector3	color	`(0.8, 0.8, 0.8)`	C++ | Lua
float	density	`0.01`	C++ | Lua
float	linearStart	`10.0`	C++ | Lua
float	linearEnd	`100.0`	C++ | Lua

Methods

Type	Name	Langs
void	setType	C++
void	setColor	C++ | Lua
void	setDensity	C++
void	setLinearStart	C++
void	setLinearEnd	C++
void	setLinearStartEnd	C++ | Lua
FogComponent (p. 671)	`getFogComponent`	Lua

Enumerations

FogType

- **LINEAR** — Fog increases linearly between `linearStart` and `linearEnd` distances. Full opacity at `linearEnd`.
- **EXPONENTIAL** — Fog uses the formula `exp(-density * d)`. Grows gradually regardless of distance bounds.
- **EXPONENTIALSQUARED** — Fog uses `exp(-(density * d)²)`. More concentrated near objects, faster roll-off.

Property details

type

- *Setter*: void **setType**(FogType type)
- *Getter*: FogType **getType**() const

The fog calculation model.

C++

```cpp
Fog fog(&scene);
fog.setType(FogType::LINEAR);
fog.setColor(Vector3(0.7f, 0.8f, 0.9f));
fog.setLinearStartEnd(50.0f, 200.0f);
```

Lua

```lua
local fog = Fog(scene)
fog.type = FogType.LINEAR
fog.color = Vector3(0.7, 0.8, 0.9)
fog:setLinearStartEnd(50, 200)
```

color

- *Setter*: void **setColor**(Vector3 color)
- *Setter*: void **setColor**(float red, float green, float blue)
- *Getter*: Vector3 **getColor**() const

The fog colour in linear space. Match this to your sky colour for a natural-looking effect.

density

- *Setter*: void **setDensity**(float density)
- *Getter*: float **getDensity**() const

The density coefficient for `EXPONENTIAL` and `EXPONENTIALSQUARED` modes. Larger values produce thicker fog.

linearStart / linearEnd

- *Setter*: void **setLinearStart**(float start) / **setLinearEnd**(float end)
- *Setter*: void **setLinearStartEnd**(float start, float end)
- *Getter*: float **getLinearStart**() / **getLinearEnd**() const

The near and far clip distances for `LINEAR` fog. Objects closer than `linearStart` are unaffected; objects beyond `linearEnd` are fully in the fog colour.

Light

Inherits: Object (p. 455)
C++ type: `Light`

Description

Represents a light source in a 3D scene. Doriax supports three light types — directional, point, and spot — all sharing the same `Light` class. The light type determines which properties have effect: a directional light uses `direction` and an unlimited `range`, a point light radiates in all directions with a finite `range`, and a spot light uses `direction` with either cone angles or an optional projected mask.

Shadows are opt-in per-light and render via a shadow map. Cascaded shadow maps (CSM) are available for directional lights to spread shadow quality across a large view distance.

Up to `MAX_LIGHTS` (6) lights are used at once. A scene with more uses its directional lights first, then the lights nearest the camera; see Rendering Pipeline — More lights than `MAX_LIGHTS` (p. 260).

Properties

Type	Name	Default	Langs
LightType	type	`DIRECTIONAL`	C++ | Lua
Vector3	direction	`(0,-1,0)`	C++ | Lua
Vector3	color	`(1,1,1)`	C++ | Lua
float	range	`10.0`	C++ | Lua
float	intensity	`1.0`	C++ | Lua
float	innerConeAngle	`10°`	C++ | Lua
float	outerConeAngle	`45°`	C++ | Lua
Texture (p. 738)	spotMask	empty	C++ | Lua (read-only)
bool	shadows	`false`	C++ | Lua
float	bias	`0.002`	C++ | Lua
unsigned int	shadowMapSize	`2048`	C++ | Lua
float	cameraNear	`0.1`	C++ | Lua
float	cameraFar	`100.0`	C++ | Lua
bool	automaticShadowCamera	`true`	C++ | Lua
unsigned int	numCascades	`3`	C++ | Lua
bool	cascadeBlend	`false`	C++ | Lua

Methods

Type	Name	Langs
void	setType	C++ | Lua
LightType	getType	C++
void	setDirection	C++ | Lua
Vector3	getDirection	C++
void	setColor	C++ | Lua
Vector3	getColor	C++
void	setRange	C++ | Lua
float	getRange	C++
void	setIntensity	C++ | Lua

Type	Name	Langs
float	getIntensity	C++
void	setConeAngle	C++ | Lua
void	setInnerConeAngle	C++ | Lua
float	getInnerConeAngle	C++
void	setOuterConeAngle	C++ | Lua
float	getOuterConeAngle	C++
void	setSpotMask	C++ | Lua
void	clearSpotMask	C++ | Lua
Texture (p. 738)	getSpotMask	C++
void	setShadows	C++ | Lua
bool	isShadows	C++
void	setBias	C++ | Lua
float	getBias	C++
void	setShadowMapSize	C++ | Lua
unsigned int	getShadowMapSize	C++
void	setShadowCameraNearFar	C++ | Lua
void	setCameraNear	C++ | Lua
float	getCameraNear	C++
void	setCameraFar	C++ | Lua
float	getCameraFar	C++
void	setAutomaticShadowCamera	C++
bool	isAutomaticShadowCamera	C++
void	setNumCascades	C++ | Lua
float	getNumCascades	C++
void	setCascadeBlend	C++
bool	isCascadeBlend	C++
LightComponent (p. 677)	`getLightComponent`	Lua

Enumerations

LightType

- **DIRECTIONAL** - A light that shines uniformly in one direction from infinitely far away, like the sun. Position is irrelevant; only `direction` matters.
- **POINT** - A light that radiates equally in all directions from its world-space position, like a light bulb. Use `range` and `intensity` to control reach.
- **SPOT** - A projected light emanating from its world-space position in the given `direction`. Use `innerConeAngle` and `outerConeAngle` for the default circular cone, or assign a spotMask for an arbitrary shape.

Property details

type

- *Setter*: void **setType**(LightType type)
- *Getter*: LightType **getType**() const

The light model to use. See LightType for the available values. Changing the type at run-time is supported.

C++

```
Light sun(&scene);
sun.setType(LightType::DIRECTIONAL);
sun.setDirection(Vector3(-0.5f, -1.0f, -0.5f));
```

Lua

```
local sun = Light(scene)
sun:setType(LightType.DIRECTIONAL)
sun:setDirection(Vector3(-0.5, -1.0, -0.5))
```

direction

- *Setter*: void **setDirection**(Vector3 direction)
- *Setter*: void **setDirection**(float x, float y, float z)
- *Getter*: Vector3 **getDirection**() const

The normalised direction vector the light shines *towards*. Used by `DIRECTIONAL` and `SPOT` lights.

color

- *Setter*: void **setColor**(Vector3 color)
- *Setter*: void **setColor**(float r, float g, float b)
- *Getter*: Vector3 **getColor**() const

The RGB color of the emitted light, in sRGB. Each channel ranges from `0.0` to `1.0` for standard colours, though values above `1.0` are valid for HDR scenes. It is stored linear internally, so the value read back is the sRGB form of the stored one.

range

- *Setter*: void **setRange**(float range)
- *Getter*: float **getRange**() const

The maximum distance (in world units) at which the light has any effect. Relevant for `POINT` and `SPOT` lights. `DIRECTIONAL` lights have an infinite range and ignore this value.

intensity

- *Setter*: void **setIntensity**(float intensity)
- *Getter*: float **getIntensity**() const

A multiplier applied to the light's color. `1.0` is the default. Higher values create a brighter light without altering the hue. At `0` the light is off and leaves its slot to another light.

innerConeAngle / outerConeAngle

- *Setter*: void **setConeAngle**(float inner, float outer)
- *Setter*: void **setInnerConeAngle**(float inner)
- *Getter*: float **getInnerConeAngle**() const
- *Setter*: void **setOuterConeAngle**(float outer)
- *Getter*: float **getOuterConeAngle**() const

Define the shape of a `SPOT` light. The angle is specified in degrees (or radians if `Engine::useDegrees` is `false`).

- **innerConeAngle** — the full-brightness cone. Everything inside this angle receives 100 % of the light.
- **outerConeAngle** — the penumbra edge. Light fades to zero between the inner and outer angle.

`setConeAngle()` sets both values at once.

When a spotMask is assigned, `outerConeAngle` controls the mask's vertical projection angle and the image aspect ratio controls its width. `innerConeAngle` is ignored until the mask is cleared.

C++

```
Light torch(&scene);
torch.setType(LightType::SPOT);
torch.setConeAngle(15.0f, 40.0f);
```

Lua

```
local torch = Light(scene)
torch:setType(LightType.SPOT)
torch:setConeAngle(15.0, 40.0)
```

spotMask

- *Setter*: void **setSpotMask**(const std::string& path)
- *Setter*: void **setSpotMask**(const std::string& id, TextureData (p. 744) data)
- *Clearer*: void **clearSpotMask**()
- *Getter*: Texture (p. 738) **getSpotMask**() const

An optional image projected by a `SPOT` light. It replaces the default circular cone attenuation, allowing shapes such as windows, logos, arrows, and soft-edged patterns without a separate spot-shape setting. The `spotMask` property only reads the current mask in either language; `setSpotMask` and `clearSpotMask` are the write path, Lua included.

For an image with transparency, alpha controls the light intensity. Otherwise, the image luminance is used. Black produces no light, white produces full light, and intermediate values produce partial intensity. The source aspect ratio is preserved, and spot-light shadows use the same projection frame and aspect ratio.

Camera/framebuffer textures are not supported as spot masks. Call `clearSpotMask()` to return to the normal inner/outer circular cone.

C++

```
Light projector(&scene);
projector.setType(LightType::SPOT);
projector.setOuterConeAngle(50.0f);
projector.setSpotMask("textures/window-mask.png");

// Restore the default circular cone.
projector.clearSpotMask();
```

Lua

```
local projector = Light(scene)
projector:setType(LightType.SPOT)
projector:setOuterConeAngle(50.0)
projector:setSpotMask("textures/window-mask.png")

-- Restore the default circular cone.
projector:clearSpotMask()
```

shadows

- *Setter*: void **setShadows**(bool shadows)
- *Getter*: bool **isShadows**() const

Enables shadow map generation for this light. Shadow casting also requires the caster mesh to have `castShadows` enabled and the receiver mesh to have `receiveShadows` enabled. See the Mesh (p. 436) documentation.

bias

- *Setter*: void **setBias**(float bias)
- *Getter*: float **getBias**() const

The shadow-map depth bias added to prevent self-shadowing artefacts (shadow acne). Increase this value if you see acne on surfaces; decrease it if you see shadows detaching from geometry (Peter-panning). A good starting value is `0.005`.

shadowMapSize

- *Setter*: void **setShadowMapSize**(unsigned int size)
- *Getter*: unsigned int **getShadowMapSize**() const

The resolution of the shadow map texture in pixels (e.g. `512`, `1024`, `2048`). Higher values produce sharper shadows at the cost of GPU memory and fillrate. Must be a power of two.

cameraNear / cameraFar

- *Setter*: void **setShadowCameraNearFar**(float nearValue, float farValue)
- *Setter*: void **setCameraNear**(float nearValue)
- *Getter*: float **getCameraNear**() const
- *Setter*: void **setCameraFar**(float farValue)
- *Getter*: float **getCameraFar**() const

The near and far clip planes of the shadow-map camera. Tighten these around the scene geometry to maximise shadow-map precision. `setShadowCameraNearFar()` sets both at once.

Has no effect when automaticShadowCamera is `true`.

automaticShadowCamera

- *Setter*: void **setAutomaticShadowCamera**(bool automatic)
- *Getter*: bool **isAutomaticShadowCamera**() const

When `true`, the engine automatically fits the shadow-map camera frustum around the visible scene each frame, optimising shadow quality. Disable this if you need manual control over cameraNear / cameraFar.

numCascades

- *Setter*: void **setNumCascades**(unsigned int numCascades)
- *Getter*: float **getNumCascades**() const

Number of cascades for Cascaded Shadow Maps (CSM). Only relevant for `DIRECTIONAL` lights. More cascades distribute shadow-map resolution more evenly across the view frustum, reducing the "blurry distant shadows" artefact at the cost of additional render passes. Typical values are `1` (off) to `4`.

cascadeBlend

- *Setter*: void **setCascadeBlend**(bool cascadeBlend)
- *Getter*: bool **isCascadeBlend**() const

Fades each cascade into the next over a short band past its split, which hides the line where the shadow resolution changes. Only relevant for `DIRECTIONAL` lights with more than one cascade. It is off by default because the band samples two cascades and the finer one draws more casters.

Light2D

Inherits: Object (p. 455)
C++ type: `Light2D`

Description

A point light for 2D scenes. Unlike Light (p. 424), which drives the 3D PBR pipeline, a `Light2D` lives in the scene's XY plane and uses a simple radius falloff designed for the classic 2D look. Light from every `Light2D` is added on top of the scene's 2D ambient light (see Scene — setAmbientLight2D (p. 404)); dim the ambient to make lights visible.

Sprites, tilemaps, and mesh polygons receive 2D light when their Mesh component has **Receive Lights** enabled (the default). UI objects are not affected. Up to `MAX_LIGHTS_2D` (16) lights can be active at once.

Shadows are opt-in per light and are cast by Occluder2D (p. 462) components using per-light 1D polar shadow maps — see Rendering Pipeline — 2D lighting (p. 261).

Properties

Type	Name	Default	Langs
Vector3	color	`(1,1,1)`	C++ | Lua
float	intensity	`1.0`	C++ | Lua
float	range	`200.0`	C++ | Lua
float	falloff	`1.0`	C++ | Lua
float	height	`0.0`	C++ | Lua
bool	shadows	`false`	C++ | Lua
float	shadowBias	`0.01`	C++ | Lua
float	shadowSoftness	`2.0`	C++ | Lua
unsigned int	mapResolution	`512`	C++ | Lua

Methods

Type	Name	Langs
void	setColor	C++ | Lua
Vector3	getColor	C++
void	setIntensity	C++ | Lua
float	getIntensity	C++
void	setRange	C++ | Lua
float	getRange	C++
void	setFalloff	C++ | Lua
float	getFalloff	C++
void	setHeight	C++ | Lua
float	getHeight	C++
void	setShadows	C++ | Lua
bool	isShadows	C++
void	setShadowBias	C++
float	getShadowBias	C++
void	setShadowSoftness	C++
float	getShadowSoftness	C++

Type	Name	Langs
void	setMapResolution	C++
unsigned int	getMapResolution	C++
Light2DComponent (p. 676)	`getLight2DComponent`	Lua

Property details

color

- *Setter*: void **setColor**(Vector3 color)
- *Setter*: void **setColor**(float r, float g, float b)
- *Getter*: Vector3 **getColor**() const

The RGB color of the emitted light, in sRGB. Each channel ranges from `0.0` to `1.0`. It is stored linear internally, so the value read back is the sRGB form of the stored one.

C++

```
scene.setAmbientLight2D(0.15f); // dark scene so the light is visible

Light2D torch(&scene);
torch.setPosition(Vector3(300, 200, 0));
torch.setColor(1.0f, 0.8f, 0.5f);
torch.setRange(250);
```

Lua

```
scene:setAmbientLight2D(0.15) -- dark scene so the light is visible

local torch = Light2D(scene)
torch.position = Vector3(300, 200, 0)
torch:setColor(1.0, 0.8, 0.5)
torch:setRange(250)
```

intensity

- *Setter*: void **setIntensity**(float intensity)
- *Getter*: float **getIntensity**() const

A multiplier applied to the light's color. `1.0` is the default. Higher values create a brighter light without altering the hue.

range

- *Setter*: void **setRange**(float range)
- *Getter*: float **getRange**() const

The radius of the light in world units. Nothing beyond this distance receives any light. In the editor, the range is drawn as a circle around the selected light.

falloff

- *Setter*: void **setFalloff**(float falloff)
- *Getter*: float **getFalloff**() const

The attenuation exponent. Brightness fades as `pow(1 - distance/range, falloff)`: `1.0` is a linear fade, higher values concentrate the light near its center, values below `1.0` push light towards the edge of the range.

height

- *Setter*: void **setHeight**(float height)
- *Getter*: float **getHeight**() const

A virtual Z height of the light above the 2D plane. With a height greater than zero, the light has a direction relative to each surface point, so **normal maps** on sprites produce relief shading that shifts as the light moves. With `0.0` (default) the light is purely radial and normals are ignored.

C++

```
// relief shading on a sprite with a normal texture in its material
torch.setHeight(80.0f);
```

Lua

```
-- relief shading on a sprite with a normal texture in its material
torch.height = 80.0
```

shadows

- *Setter*: void **setShadows**(bool shadows)
- *Getter*: bool **isShadows**() const

Enables shadow casting for this light. Shadows are produced by Occluder2D (p. 462) components in the scene; the receiver mesh must have `receiveShadows` enabled. Each light's shadow darkens only that light's contribution — the scene's ambient light is never shadowed.

shadowBias

- *Setter*: void **setShadowBias**(float shadowBias)
- *Getter*: float **getShadowBias**() const

Depth bias for the shadow comparison, in normalized units of the light's range. Increase it if surfaces right at an occluder edge show shadow acne; decrease it if shadows detach from their occluders.

shadowSoftness

- *Setter*: void **setShadowSoftness**(float shadowSoftness)
- *Getter*: float **getShadowSoftness**() const

The penumbra half-width, in shadow-map texels. `0.0` gives hard edges; values around `2.0`–`4.0` give a soft natural edge. If a wide penumbra shows banding, raise the scene-wide filter quality — see Scene — shadow2DQuality (p. 396).

mapResolution

- *Setter*: void **setMapResolution**(unsigned int mapResolution)
- *Getter*: unsigned int **getMapResolution**() const

The width in pixels of this light's 1D polar shadow map. Higher values give more angular precision for distant or thin occluders. The shadow atlas is sized by the largest `mapResolution` among the shadow-casting lights.

Lines

Inherits: Object (p. 455)
C++ type: `Lines`

Description

Renders a collection of 3D line segments. Useful for debug overlays, grids, trajectory paths, and other wireframe visuals. Each line is defined by two endpoints and an optional per-endpoint colour for gradient lines.

The `Lines` object maintains an internal dynamic buffer. Call `updateLines()` after modifying line data to upload changes to the GPU.

Properties

Type	Name	Default	Langs
unsigned int	maxLines	`100`	C++ | Lua
string	customShader	`""`	C++ | Lua

Methods

Type	Name	Langs
bool	load	C++
void	addLine	C++ | Lua
LineData (p. 713)	getLine	C++ | Lua
void	updateLine	C++ | Lua
void	removeLine	C++ | Lua
void	updateLines	C++ | Lua
size_t	getNumLines	C++ | Lua
void	clearLines	C++ | Lua
void	setShaderUniform	C++ | Lua
Vector4	getShaderUniform	C++ | Lua
bool	removeShaderUniform	C++ | Lua
LinesComponent (p. 678)	`getLinesComponent`	Lua

Property details

maxLines

- *Setter*: void **setMaxLines**(unsigned int maxLines)
- *Getter*: unsigned int **getMaxLines**() const

Pre-allocates buffer space for this many lines. Resize before calling `load()` for best performance.

Method details

load

- bool **load**()

Explicitly initialises the GPU buffer. This method is **C++ only** and is normally optional because the render system initialises lines automatically.

addLine

Several overloads are available:

- void **addLine**(LineData (p. 713) line)
- void **addLine**(Vector3 pointA, Vector3 pointB)
- void **addLine**(Vector3 pointA, Vector3 pointB, Vector3 color)
- void **addLine**(Vector3 pointA, Vector3 pointB, Vector4 color)
- void **addLine**(Vector3 pointA, Vector3 pointB, Vector4 colorA, Vector4 colorB)

Adds a new line segment. The last overload lets each endpoint have a different colour.

C++

```
Lines debug(&scene);
debug.load();
debug.addLine(Vector3(0,0,0), Vector3(1,0,0), Vector4(1,0,0,1));  // red X axis
debug.addLine(Vector3(0,0,0), Vector3(0,1,0), Vector4(0,1,0,1));  // green Y axis
debug.addLine(Vector3(0,0,0), Vector3(0,0,1), Vector4(0,0,1,1));  // blue Z axis
debug.updateLines();
```

Lua

```
local debug = Lines(scene)
debug:addLine(Vector3(0,0,0), Vector3(1,0,0), Vector4(1,0,0,1))
debug:addLine(Vector3(0,0,0), Vector3(0,1,0), Vector4(0,1,0,1))
debug:addLine(Vector3(0,0,0), Vector3(0,0,1), Vector4(0,0,1,1))
debug:updateLines()
```

getLine

- LineData (p. 713)& **getLine**(size_t index)

Returns a reference to the LineData (p. 713) at the given index. Modify in-place then call `updateLines()`.

updateLine

- void **updateLine**(size_t index, ...)

Overwrites an existing line's endpoints and/or colour. Multiple overloads mirror `addLine()`. Call `updateLines()` afterwards.

removeLine

- void **removeLine**(size_t index)

Removes the line at `index`. All subsequent indices shift down.

updateLines

- void **updateLines**()

Uploads all pending line changes to the GPU. Must be called after any `addLine`, `updateLine`, or `removeLine` call.

getNumLines

- size_t **getNumLines**()

Returns the current number of line segments.

clearLines

- void **clearLines**()

Removes all line segments.

customShader

- *Setter:* `void setCustomShader(const std::string& path)`
- *Getter:* `std::string getCustomShader() const`

Project-relative base path of a forked shader (`"shaders/laser"` resolves to `shaders/laser.vert` and `shaders/laser.frag`, or `"a.vert|b.frag"` names the files separately). Empty (default) uses the scene default shader for this type, or the engine built-in. Changing it reloads the component. Same semantics as Mesh.customShader (p. 441); see Custom Shaders (p. 143).

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`
- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the custom shader declares with `#pragma param`, by name. Setting a value only rewrites the uploaded block, so it can run every frame; a parameter with no value reads the scene's value under a scene default shader (p. 405), else its declared default. Same semantics as Mesh.setShaderUniform (p. 445); see Custom Shaders — Shader parameters (p. 147).

C++

```cpp
lines.setCustomShader("shaders/laser");
lines.setShaderUniform("strength", 0.8f);
lines.setShaderUniform("glowColor", Vector4(1.0f, 0.8f, 0.2f, 1.0f));
```

Lua

```lua
lines.customShader = "shaders/laser"
lines:setShaderUniform("strength", 0.8)
lines:setShaderUniform("glowColor", Vector4(1, 0.8, 0.2, 1))
```

Mesh

Description

`Mesh` is the base class for all renderable 3D objects. It exposes the material system (color, texture, PBR material), shadow casting and receiving, transparency control, face culling, automatic detail levels, and GPU instancing (including distance fade and per-view instance culling) for drawing thousands of objects in a single draw call.

Inherits: Object (p. 455) → EntityHandle (p. 711)

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Mesh mesh(&scene);
mesh.setTexture("textures/crate.png");
mesh.setColor(1.0f, 1.0f, 1.0f, 1.0f);
mesh.setCastShadows(true);
```

Lua

```lua
local mesh = Mesh(scene)
mesh:setTexture("textures/crate.png")
mesh:setColor(1, 1, 1, 1)
mesh.castShadows = true
```

Constructors

Available in C++ and Lua:

- `Mesh(Scene* scene)` — creates a new entity.
- `Mesh(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `Mesh(scene, entity)` to access an existing mesh. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Properties

Type	Name	Default	Languages
Vector4	color	`(1,1,1,1)`	C++ | Lua
float	alpha	`1.0`	C++ | Lua
Material	material	default	C++ | Lua
PrimitiveType	primitiveType	`TRIANGLES`	C++ | Lua
bool	faceCulling	`true`	C++ | Lua
CullingMode	cullingMode	`BACK`	C++ | Lua
WindingOrder	windingOrder	`CCW`	C++ | Lua
bool	receiveLights	`true`	C++ | Lua
bool	receiveIBL	`false`	C++ | Lua
bool	castShadows	`true`	C++ | Lua
bool	receiveShadows	`true`	C++ | Lua
bool	lodEnabled	`true`	C++ | Lua
float	lodBias	`1.0`	C++ | Lua
bool	shadowsBillboard	`true`	C++ | Lua
bool	renderInReflectionProbes	`true`	C++ | Lua
bool	castShadowsWithTexture	`false`	C++ | Lua

Type	Name	Default	Languages
bool	transparent	`false`	C++ | Lua
bool	autoTransparency	`true`	C++ | Lua
string	customShader	`""`	C++ | Lua
string	customDepthShader	`""`	C++ | Lua
string	vertexOffsetShader	`""`	C++ | Lua
float	vertexOffsetMax	`0`	C++ | Lua
string	coverageShader	`""`	C++ | Lua
bool	coverageShadows	`false`	C++ | Lua
unsigned int	maxInstances	`100`	C++ | Lua
bool	instancedBillboard	`false`	C++ | Lua
bool	instancedCylindricalBillboard	`false`	C++ | Lua
bool	distanceFade	`false`	C++ | Lua
float	fadeStart	`0`	C++ | Lua
float	fadeEnd	`0`	C++ | Lua
bool	cullInstances	`true`	C++ | Lua
float	cullDistance	`0`	C++ | Lua

Methods

Returns	Name	Languages
bool	load	C++
void	setTexture	C++ | Lua
void	setColor	C++ | Lua
Vector4	getColor	C++
void	setAlpha	C++
float	getAlpha	C++
void	setMaterial	C++ | Lua
Material	getMaterial	C++ | Lua
void	setPrimitiveType	C++ | Lua
PrimitiveType	getPrimitiveType	C++ | Lua
void	setFaceCulling	C++ | Lua
bool	isFaceCulling	C++ | Lua
void	setCullingMode	C++ | Lua
CullingMode	getCullingMode	C++ | Lua
void	setWindingOrder	C++ | Lua
WindingOrder	getWindingOrder	C++ | Lua
AABB	getAABB	C++ | Lua
AABB	getVerticesAABB	C++ | Lua
AABB	getWorldAABB	C++ | Lua
unsigned int	getNumSubmeshes	C++ | Lua
void	setReceiveLights	C++ | Lua
bool	isReceiveLights	C++ | Lua
void	setCastShadows	C++
bool	isCastShadows	C++
void	setReceiveShadows	C++

Returns	Name	Languages
bool	isReceiveShadows	C++
void	setLodEnabled	C++
bool	isLodEnabled	C++
void	setLodBias	C++
float	getLodBias	C++
void	setShadowsBillboard	C++ | Lua
bool	isShadowsBillboard	C++ | Lua
void	setCastShadowsWithTexture	C++ | Lua
bool	isCastShadowsWithTexture	C++ | Lua
void	setTransparent	C++
bool	isTransparent	C++
void	setAutoTransparency	C++
bool	isAutoTransparency	C++
void	setCustomShader	C++
string	getCustomShader	C++
void	setCustomDepthShader	C++
string	getCustomDepthShader	C++
void	setShaderUniform	C++ | Lua
Vector4	getShaderUniform	C++ | Lua
bool	removeShaderUniform	C++ | Lua
void	setAsMirror	C++ | Lua
void	removeMirror	C++ | Lua
bool	isMirror	C++ | Lua
void	createInstancedMesh	C++ | Lua
void	removeInstancedMesh	C++ | Lua
bool	hasInstancedMesh	C++ | Lua
void	addInstance	C++ | Lua
InstanceData&	getInstance	C++ | Lua
void	updateInstance	C++ | Lua
void	removeInstance	C++ | Lua
bool	isInstanceVisible	C++ | Lua
void	setInstanceVisible	C++ | Lua
void	updateInstances	C++ | Lua
size_t	getNumInstances	C++ | Lua
void	clearInstances	C++ | Lua
void	setInstancedBillboard	C++ | Lua
bool	isInstancedBillboard	C++
void	setInstancedCylindricalBillboard	C++
bool	isInstancedCylindricalBillboard	C++
void	setMaxInstances	C++
unsigned int	getMaxInstances	C++
void	setDistanceFade	C++
bool	isDistanceFade	C++
void	setFadeRange	C++ | Lua

Returns	Name	Languages
void	setFadeStart	C++
float	getFadeStart	C++
void	setFadeEnd	C++
float	getFadeEnd	C++
void	setCullInstances	C++
bool	isCullInstances	C++
void	setCullDistance	C++
float	getCullDistance	C++
MeshComponent (p. 679)	`getMeshComponent`	Lua
InstancedMeshComponent (p. 672)	`getInstancedMeshComponent`	Lua

Enumerations

PrimitiveType

- **TRIANGLES** — Solid triangle meshes (default).
- **TRIANGLE_STRIP** — Triangle strip.
- **POINTS** — Renders each vertex as a point.
- **LINES** — Renders edges as lines.

CullingMode

- **BACK** — Culls back faces (default, outward-facing normals).
- **FRONT** — Culls front faces (useful for inside-out rendering).

WindingOrder

- **CCW** — Counter-clockwise winding defines the front face (OpenGL convention, default).
- **CW** — Clockwise winding defines the front face.

Property details

color

- *Setter:* `void setColor(Vector4 color)`
- *Setter:* `void setColor(float red, float green, float blue, float alpha)`
- *Setter:* `void setColor(float red, float green, float blue)`
- *Getter:* `Vector4 getColor() const`

Tint color multiplied with the material texture. `(1,1,1,1)` means no tint.

alpha

- *Setter:* `void setAlpha(float alpha)`
- *Getter:* `float getAlpha() const`

Shorthand for the alpha channel of color. Values below 1.0 require transparent to be enabled for correct rendering order.

material

- *Setter:* `void setMaterial(const Material& material)` *(all submeshes)*
- *Setter:* `void setMaterial(unsigned int submesh, const Material& material)` *(per submesh)*
- *Getter:* `Material getMaterial() const`
- *Getter:* `Material getMaterial(unsigned int submesh) const`

PBR material descriptor. Contains base color, metallic/roughness factors, normal map, emissive settings, and shader overrides.

primitiveType

- *Setter:* `void setPrimitiveType(PrimitiveType primitiveType)`
- *Getter:* `PrimitiveType getPrimitiveType() const`

GPU draw mode. Overloads are available per submesh.

faceCulling

- *Setter:* `void setFaceCulling(bool faceCulling)`
- *Getter:* `bool isFaceCulling() const`

Enables or disables back-face culling globally for this mesh. Disable for double-sided geometry (e.g. leaves, cards): back faces are then lit with their normal flipped toward the viewer, as glTF double-sided materials are.

castShadows / receiveShadows

- *Setter:* `void setCastShadows(bool castShadows)` / `void setReceiveShadows(bool receiveShadows)`
- *Getter:* `bool isCastShadows() const` / `bool isReceiveShadows() const`

Shadow participation flags. Enabling both adds depth-map cost; disable on distant or small objects for performance.

lodEnabled / lodBias

- *lodEnabled Setter:* `void setLodEnabled(bool lodEnabled)`
- *lodEnabled Getter:* `bool isLodEnabled() const`
- *lodBias Setter:* `void setLodBias(float lodBias)`
- *lodBias Getter:* `float getLodBias() const`

Mesh detail levels (p. 275). `lodEnabled` (default `true`) lets this mesh follow the scene's detail levels; turn it off to keep it at full detail, for example a hero asset whose silhouette must never change. Toggling it reloads the mesh. `lodBias` (default `1`) scales how far detail is kept: above 1 keeps the source longer, below 1 drops to a coarser level sooner.

Nothing happens until the scene has meshLodEnabled (p. 394) on, which is off by default. Terrain and tilemaps ignore these flags and keep their own LOD.

C++

```
mesh.setLodBias(1.5f);       // keep detail longer than the scene threshold
statue.setLodEnabled(false); // always full detail
```

Lua

```
mesh.lodBias = 1.5         -- keep detail longer than the scene threshold
statue.lodEnabled = false  -- always full detail
```

renderInReflectionProbes

- *Setter:* `void setRenderInReflectionProbes(bool renderInReflectionProbes)`
- *Getter:* `bool isRenderInReflectionProbes() const`

Whether the mesh is drawn into ReflectionProbe (p. 468) captures. Turn it off when an object surrounds or sits on a probe and would otherwise show up inside its own reflection. Normal rendering is unaffected — this only controls the capture pass.

Changing it re-captures every probe in the scene: a probe that already baked its cubemap keeps the old contents until it captures again.

receiveIBL

- *Setter:* `void setReceiveIBL(bool receiveIBL)`
- *Getter:* `bool isReceiveIBL() const`

When `true`, the mesh samples image-based lighting: diffuse irradiance plus specular reflections on top of punctual lights. The source is the scene's Sky environment, a Reflection Probe (p. 468) whose volume covers the mesh, or both. Sky lighting needs a Sky entity with a cubemap texture; probes need none. Also requires **Receive Lights** enabled and surface normals.

Also settable in the editor on the Mesh component (**Receive IBL**); the flag is serialized with the scene. Metallic, low-roughness materials show the strongest reflections.

Beyond reflective materials, this is what keeps *unlit* sides of an object from going flat. A surface no light reaches falls back to the scene's global illumination, which is a single flat colour — with a bright white ambient it washes out to a pale grey. With `receiveIBL` the same surface takes its colour from the sky instead. This shows up most in mirrors (p. 269), which show the side of an object facing the mirror — usually the side the key light misses.

transparent / autoTransparency

- *transparent Setter:* `void setTransparent(bool transparent)`
- *autoTransparency Setter:* `void setAutoTransparency(bool autoTransparency)`

`transparent` marks the mesh for the transparent render pass (sorted back-to-front). `autoTransparency` detects blending from the material (`BLEND`, or `AUTO` when the texture or factor is translucent) and enables the transparent pass automatically. A blended submesh keeps depth testing but disables depth writes in the colour pass so overlapping translucent surfaces composite correctly.

customShader

- *Setter:* `void setCustomShader(const std::string& path)`
- *Getter:* `std::string getCustomShader() const`

Project-relative base path of a forked shader used for this mesh's colour pass — `"shaders/wave"` resolves to `shaders/wave.vert` and `shaders/wave.frag`, or `"a.vert|b.frag"` names the two files separately. Empty (default) uses the scene's defaultMeshShader (p. 400), or the engine built-in when the scene has none. Changing it reloads the mesh. The engine loads the compiled shader; the sources are only needed by the editor. See Custom Shaders (p. 143).

Like colour and material, this is per Mesh component: a multi-node glTF Model (p. 450) renders through its part entities, so assign the shader (and its uniforms) on the parts — `Mesh(scene, partEntity)` — not on the Model root, whose own Mesh is empty unless the model is merged or single-node.

C++

```
mesh.setCustomShader("shaders/wave");
```

Lua

```
mesh.customShader = "shaders/wave"
```

customDepthShader

- *Setter:* `void setCustomDepthShader(const std::string& path)`
- *Getter:* `std::string getCustomDepthShader() const`

Project-relative base path of a forked **depth** shader — the one the mesh is drawn with into shadow maps, into the optional scene depth prepass (p. 276), and into the depth pre-pass that feeds SSAO and post-process depth while SSR is off. Same path forms as customShader; empty (default) uses the engine built-in, and there is no scene default. Fork it when the colour fork displaces vertices or discards fragments, so the shadow follows; it reads the same shader uniforms. Changing it reloads the mesh. See Custom Shaders — The Depth Shader row (p. 144).

C++

```
mesh.setCustomShader("shaders/wave");
mesh.setCustomDepthShader("shaders/wave_depth");
```

Lua

```
mesh.customShader = "shaders/wave"
mesh.customDepthShader = "shaders/wave_depth"
```

vertexOffsetShader / vertexOffsetMax

- *vertexOffsetShader Setter:* `void setVertexOffsetShader(const std::string& path)`
- *vertexOffsetShader Getter:* `std::string getVertexOffsetShader() const`
- *vertexOffsetMax Setter:* `void setVertexOffsetMax(float vertexOffsetMax)`
- *vertexOffsetMax Getter:* `float getVertexOffsetMax() const`

`vertexOffsetShader` is a `.glsl` file, from the project or the engine, defining `vec3 getVertexOffset(vec3 localPosition, vec3 worldPosition)`, which returns how far each vertex moves in world units. The color, depth and G-buffer passes all apply it, so shadows and reflections follow, and its `#pragma param` values are the mesh's shader uniforms. The engine's `offsets/wind.glsl` sways the mesh in the scene's `Wind.velocity`. Empty (default) moves nothing. Changing it reloads the mesh. `vertexOffsetMax` is the longest offset: longer ones are clamped, and culling widens the bounds by it; `0` means no limit. Terrain ignores both. See Shader Effects (p. 283).

C++

```
flag.setVertexOffsetShader("shaders/flag_offset.glsl");
flag.setVertexOffsetMax(0.5f);
flag.setShaderUniform("waveHeight", 0.2f);
```

Lua

```
flag.vertexOffsetShader = "shaders/flag_offset.glsl"
flag.vertexOffsetMax = 0.5
flag:setShaderUniform("waveHeight", 0.2)
```

coverageShader / coverageShadows

- *coverageShader Setter:* `void setCoverageShader(const std::string& path)`
- *coverageShader Getter:* `std::string getCoverageShader() const`

- *coverageShadows Setter:* `void setCoverageShadows(bool coverageShadows)`
- *coverageShadows Getter:* `bool isCoverageShadows() const`

`coverageShader` is a `.glsl` file, from the project or the engine, defining `float getCoverage(CoverageContext context)`, which returns how much of each fragment stays: 1 keeps it, 0 removes it, and opaque surfaces keep a dithered share in between while blended ones fade. The color, depth and G-buffer passes all apply it, so SSAO and reflections match, and its `#pragma param` values are the mesh's shader uniforms. The engine ships `coverage/see_through.glsl`, `coverage/dissolve.glsl` and `coverage/height_clip.glsl`. Empty (default) keeps everything. Changing it reloads the mesh. With `coverageShadows` on, the shadow maps follow the coverage too; off (default) they keep the whole silhouette, though the material's alpha mask still applies. Terrain ignores both. See Shader Effects (p. 284).

C++

```
crate.setCoverageShader("coverage/dissolve.glsl");
crate.setCoverageShadows(true);
crate.setShaderUniform("dissolve", 0.5f);
```

Lua

```
crate.coverageShader = "coverage/dissolve.glsl"
crate.coverageShadows = true
crate:setShaderUniform("dissolve", 0.5)
```

maxInstances

- *Setter:* `void setMaxInstances(unsigned int maxInstances)`
- *Getter:* `unsigned int getMaxInstances() const`

Slots in the GPU instance buffer, `100` by default. Drawing clamps to it, but addInstance grows the cap as the list outgrows it — reloading the mesh each time — so the clamp only bites when the instance list is filled without going through `addInstance`. Set the capacity up front when the count is already known, to skip those reloads. Unlike the billboard setters, this one logs an error if the mesh is not instanced yet — call createInstancedMesh first.

---

instancedBillboard / instancedCylindricalBillboard

- *Setter:* `void setInstancedBillboard(bool billboard)`
- *Setter:* `void setInstancedBillboard(bool billboard, bool cylindrical)`
- *Getters:* `bool isInstancedBillboard() const`, `bool isInstancedCylindricalBillboard() const`

Turns every instance to face the camera, the per-instance counterpart of Object.billboard (p. 458). Both default to `false`, and `instancedCylindricalBillboard` only matters while `instancedBillboard` is on. The setters create the instanced mesh if it does not exist yet.

> **Cylindrical is not Y-only yet**
> The instanced path builds its rotation from the inverse view matrix whichever flag is set, so instanced cylindrical facing currently matches the spherical one. Object.cylindricalBillboard (p. 458) does constrain to Y.

C++

```
grass.setInstancedBillboard(true, true);
```

Lua

```
grass.instancedBillboard = true
grass.instancedCylindricalBillboard = true
```

---

distanceFade / fadeStart / fadeEnd

- *distanceFade Setter:* `void setDistanceFade(bool distanceFade)`
- *distanceFade Getter:* `bool isDistanceFade() const`
- *Setter:* `void setFadeRange(float fadeStart, float fadeEnd)`
- *fadeStart Setter / Getter:* `void setFadeStart(float fadeStart)` / `float getFadeStart() const`
- *fadeEnd Setter / Getter:* `void setFadeEnd(float fadeEnd)` / `float getFadeEnd() const`

Instanced-mesh distance fade (p. 274). When `distanceFade` is on, instances shrink toward their base between `fadeStart` and `fadeEnd`, both model-space distances from the camera. Toggling the flag reloads the mesh (shader variant `Ifd`); the two distances are uniforms and can change without a reload. `Fade End` must be greater than `Fade Start`. Logs an error if the mesh is not instanced yet.

On the built-in triangle shaders, instances past `fadeEnd` are also omitted from the draw. Custom shaders and the SSR G-buffer still submit them and rely on the vertex scale.

C++

```
trees.createInstancedMesh();
trees.setDistanceFade(true);
trees.setFadeRange(40.0f, 60.0f);
```

Lua

```
trees:createInstancedMesh()
trees.distanceFade = true
trees:setFadeRange(40, 60)
```

cullInstances

- *Setter:* `void setCullInstances(bool cullInstances)`
- *Getter:* `bool isCullInstances() const`

When `true` (default), instances outside the main camera or a shadow atlas slot are left out of that view. Turn it off when a shader moves instances away from the bounds the CPU computed; the whole entity then skips frustum culling too. Logs an error if the mesh is not instanced yet. See Culling granularity (p. 274).

cullDistance

- *Setter:* `void setCullDistance(float distance)`
- *Getter:* `float getCullDistance() const`

World-space distance from the main camera past which instances are dropped from the culled views (main camera and shadow slots). `0` (default) means unlimited; negative or non-finite values are ignored. A hard cut, unlike distanceFade: instances keep their size until they vanish, and it applies with custom shaders and SSR too, so hide it with fog. Requires cullInstances. Logs an error if the mesh is not instanced yet.

C++

```
trees.createInstancedMesh();
trees.setCullDistance(150.0f);
```

Lua

```
trees:createInstancedMesh()
trees.cullDistance = 150
```

Method details

load

- `bool load()`

Uploads geometry and material resources to the GPU. Must be called after building or modifying geometry programmatically. Subclasses like `Sprite` call this internally when needed.

---

setTexture

- `void setTexture(const std::string& path)`
- `void setTexture(const std::string& id, TextureData data)`
- `void setTexture(Framebuffer* framebuffer)`

Assigns the diffuse/base-color texture. Accepts a file path, raw `TextureData`, or a framebuffer render target.

C++

```
mesh.setTexture("textures/stone.png");
// or use a render-to-texture framebuffer:
Camera cam(&scene);
cam.setRenderToTexture(true);
mesh.setTexture(cam.getFramebuffer());
```

Lua

```
mesh:setTexture("textures/stone.png")
```

---

setAsMirror / removeMirror / isMirror

- `void setAsMirror()` — turn this flat mesh into a planar reflection surface, using the default `+Z` normal (matching a Wall (p. 471)).
- `void setAsMirror(Vector3 normal)` — same, with an explicit surface normal in local space (e.g. `Vector3(0, 1, 0)` for a `createPlane` floor).
- `void removeMirror()` — stop being a mirror.
- `bool isMirror() const` — whether this mesh is currently a mirror.

The engine creates and drives the reflection camera internally and feeds its render target to the mesh's base texture — no camera setup is required. Set `receiveLights` to `false` for a reflection shown without surface shading. See Rendering Pipeline — Mirrors (p. 269).

C++

```
Shape mirror(&scene);
mirror.createWall(10.0f, 10.0f);
mirror.setAsMirror();
mirror.setReceiveLights(false);
```

Lua

```
mirror:createWall(10.0, 10.0)
mirror:setAsMirror()
mirror.receiveLights = false
```

Use these to make an existing mesh reflective. To create a mirror outright, the Mirror (p. 448) class does both steps at once and exposes the surface normal as a property.

---

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`

- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the mesh's shaders declare with `#pragma param`, by name (see Custom Shaders — Shader parameters (p. 147)). A value is stored on the mesh and applied to every submesh, in the colour fork, the depth fork, and the effect files alike; a narrower value leaves the remaining components at zero, and an `int` parameter takes the truncated value. Setting a value only rewrites the uploaded block, so it can be called every frame. Names no shader declares are kept but unused, and a parameter with no value reads the scene's value while the mesh inherits a scene default shader (p. 405), else its declared default.

`getShaderUniform` returns the stored value, zero when unset. It reads the mesh's own value list, not the uploaded block or the scene's values, so an `int` parameter set to `3.7` reads back `3.7` while the shader sees `3`. `removeShaderUniform` drops a value and returns whether it existed; the parameter reads its inherited or declared default again. For a glTF model, call these on the part entities (see customShader).

C++

```
mesh.setCustomShader("shaders/wave");
mesh.setShaderUniform("amplitude", 0.3f);
mesh.setShaderUniform("direction", Vector2(1.0f, 0.0f));
mesh.setShaderUniform("tint", Vector4(0.2f, 0.5f, 1.0f, 0.5f));

Vector4 tint = mesh.getShaderUniform("tint");
```

Lua

```
mesh.customShader = "shaders/wave"
mesh:setShaderUniform("amplitude", 0.3)
mesh:setShaderUniform("direction", Vector2(1, 0))
mesh:setShaderUniform("tint", Vector4(0.2, 0.5, 1.0, 0.5))

local tint = mesh:getShaderUniform("tint")
```

getAABB / getVerticesAABB / getWorldAABB

- `AABB getAABB() const` — Axis-aligned bounding box of the mesh in local space.
- `AABB getVerticesAABB() const` — AABB computed from raw vertex positions, ignoring the transform.
- `AABB getWorldAABB() const` — AABB in world space after applying the full transform chain.

Useful for frustum culling, click picking, and physics proxy fitting.

getNumSubmeshes

- `unsigned int getNumSubmeshes() const`

Returns the number of submeshes. Each submesh can have its own material, primitive type, and culling settings.

createInstancedMesh

- `void createInstancedMesh()`

Enables GPU instancing for this mesh. After calling this, use addInstance / updateInstance to populate the instance list, then updateInstances to flush changes to the GPU.

Instancing draws geometry on this entity only. Multi-node GLTF models that keep child mesh entities need static merge (p. 250) first (or use a single-mesh / procedural mesh).

C++

```cpp
Mesh tree(&scene);
tree.setTexture("textures/tree.png");
tree.createInstancedMesh();
tree.setMaxInstances(500);

for (int i = 0; i < 500; i++) {
    tree.addInstance(Vector3(i * 2.0f, 0.0f, 0.0f));
}
tree.updateInstances();
```

Lua

```lua
local tree = Mesh(scene)
tree:setTexture("textures/tree.png")
tree:createInstancedMesh()
tree.maxInstances = 500

for i = 0, 499 do
    tree:addInstance(Vector3(i * 2, 0, 0))
end
tree:updateInstances()
```

addInstance

- `void addInstance(InstanceData instance)`
- `void addInstance(Vector3 position)`
- `void addInstance(float x, float y, float z)`
- `void addInstance(Vector3 position, Quaternion rotation, Vector3 scale)`
- `void addInstance(Vector3 position, Quaternion rotation, Vector3 scale, Vector4 color)`
- `void addInstance(Vector3 position, Quaternion rotation, Vector3 scale, Vector4 color, Rect textureRect)`

Appends a new instance with the specified transform, creating the instanced mesh if the entity does not have one yet. Growing past maxInstances doubles the capacity and reloads the mesh.

updateInstance

- `void updateInstance(size_t index, InstanceData instance)` *(and overloads)*

Replaces the data for an existing instance at `index`. Call updateInstances afterwards to sync changes to the GPU.

updateInstances

- `void updateInstances()`

Uploads all pending instance data changes to the GPU vertex buffer. Must be called after any addInstance or updateInstance calls.

clearInstances

- `void clearInstances()`

Removes all instances from the instance list. The mesh geometry is preserved; call updateInstances afterwards.

Mirror

Inherits: Shape (p. 471)
C++ type: `Mirror`

Description

A flat mesh that reflects the scene — a true planar reflection, used for mirrors, still water, and polished floors. Constructing a `Mirror` attaches a `MirrorComponent` to the entity; the engine creates and drives the reflection camera internally, so no camera or texture wiring is required.

`Mirror` inherits Shape (p. 471), so it builds its own surface. Use createWall (p. 471) — its `+Z` normal matches the default normal, so an upright mirror needs no rotation.

See Rendering Pipeline — Mirrors and planar reflections (p. 269) for how the reflection is rendered and what it costs.

C++

```
Mirror mirror(&scene);
mirror.createWall(10.0f, 10.0f);
mirror.setReceiveLights(false);   // reflection shown unshaded
```

Lua

```
local mirror = Mirror(scene)
mirror:createWall(10.0, 10.0)
mirror.receiveLights = false
```

To make an existing mesh reflective instead — a floor, a model surface — use Mesh::setAsMirror (p. 445), which attaches the same component without replacing the geometry.

Properties

Type	Name	Default	Langs
Vector3 (p. 660)	normal	`Vector3(0, 0, 1)`	C++ | Lua

Methods

Type	Name	Langs
void	setNormal	C++ | Lua
Vector3	getNormal	C++
MirrorComponent (p. 680)	`getMirrorComponent`	Lua

Property details

normal

- *Setter*: void **setNormal**(Vector3 (p. 660) normal)
- *Setter*: void **setNormal**(const float x, const float y, const float z)
- *Getter*: Vector3 (p. 660) **getNormal**() const

The reflecting surface direction in the mesh's local space. It is transformed by the entity's world rotation to build the mirror plane, so rotating the entity orients the reflection automatically.

The default `+Z` matches a Wall (p. 471). Pass `Vector3(0, 1, 0)` for a horizontal surface such as a `createPlane` floor.

If the reflection looks wrong

The reflection image is the same whichever way the normal points, but the behind-the-mirror clipping depends on its sign. If the reflection is clipped on the wrong side, flip the sign of the normal.

Model

Inherits: Mesh (p. 436)
C++ type: `Model`

Description

Loads and displays 3D model files in a scene. `Model` extends Mesh (p. 436) and adds support for loading OBJ and GLTF files (including binary GLB), and FBX files through the glTF conversion the editor makes of them, including skeletal animation (via `Animation` / `Bone`), blend-shape morph targets, and PBR materials.

A single GLTF file can contain multiple meshes, materials, textures, skins, and animations; Doriax imports them into the scene hierarchy and makes each accessible through the API. Files with animation clips, or more than one skin, import the **full glTF node tree** so joints, mesh nodes, and transform-only helpers keep the parentage authored in the file. Use getAnimation() and getBone() to drive skeletal playback, and setMorphWeight() to control blend shapes. See 3D Graphics — GLTF node hierarchy (p. 249).

The Model entity has a `MeshComponent` only when the geometry loads into it (a single mesh, an OBJ, or a merged model). When the meshes go to child entities it keeps just `Transform` and `ModelComponent`, and the inherited Mesh (p. 436) methods act on those parts: setters change every part, getters read the first one, and a submesh index counts across the parts the way the merged model numbers its submeshes. `getAABB()` covers all the parts in the model's space and `getWorldAABB()` in the world. Settings made before the model has any geometry are held on a `MeshComponent` of the model's own until the load passes them to the meshes it builds.

Reloading the same file with `loadModel()` keeps the mesh settings (lighting, shadows, detail levels, transparency and the custom shaders) of each part, found by node name and then by index. Another file is a fresh import, and the meshes it creates start with the default settings.

For static multi-node GLTFs that need GPU instancing (p. 274) on the root entity, set `ModelComponent::mergeStaticMeshes` (or use **Merge static model** in the editor) so child transforms bake into the root mesh instead of creating child mesh entities. See 3D Graphics — Merging static model meshes (p. 250).

Methods

Type	Name	Langs
bool	loadModel	C++ | Lua
bool	loadOBJ	C++ | Lua
bool	loadGLTF	C++ | Lua
Animation (p. 573)	getAnimation	C++ | Lua
Animation (p. 573)	findAnimation	C++ | Lua
void	playAnimation	C++ | Lua
void	stopAnimations	C++ | Lua
Bone (p. 413)	getBone	C++ | Lua
float	getMorphWeight	C++ | Lua
void	setMorphWeight	C++ | Lua
void	resetToBindPose	C++ | Lua
ModelComponent (p. 681)	`getModelComponent`	Lua

GLTF import support and limits

Feature	Import behaviour
Skinning	Bone matrices are a storage buffer (Vulkan, Metal, Direct3D 11) or an unfilterable bone texture (OpenGL / OpenGL ES), so shader variants are not capped by a uniform-block size. Each skinned mesh entity gets a `SkinComponent` holding its matrices, limited by `MAX_BONES` (default **96**). The editor grows past that for large skins; export raises `MAX_BONES` to the largest skin in the project's scenes and bundles. A runtime load above the compiled capacity is rejected and logged.
Multiple skins	Supported. Each skinned mesh keeps its own joint order and inverse-bind matrices.
Node hierarchy	Animation clips, or more than one skin, import every glTF node as a child entity. Animation channels target those nodes, including transform-only helpers.
Position-only morph targets	Up to **8** targets per mesh primitive.
Position + normal or position + tangent morph targets	Up to **4** targets per mesh primitive.
Position + normal + tangent morph targets	Up to **2** targets per mesh primitive.
Sparse morph accessors	Supported. The loader expands sparse target data into a dense GPU buffer, including accessors that omit the optional base `bufferView` and therefore start from zeroes.

The morph limit is derived from the leading target prefix that the loader actually binds. Extra trailing targets are ignored; semantics that appear only after that loaded prefix do not reduce its shader capacity. Keep the same morph semantics on every target when exporting from a DCC tool for predictable results.

Imported PBR materials also preserve GLTF `OPAQUE`, `MASK`, and `BLEND` alpha modes and the `alphaCutoff` value used by masked materials. Blended submeshes disable depth writes in the colour pass. See Material (p. 735).

Loading rewrites every submesh, so submesh edits made outside the loader are kept in `ModelComponent::submeshOverrides` and re-applied at the end of each load. The editor fills that list when you change a submesh in Properties; in code you can add entries yourself, keyed by the GLTF node and primitive (`nodeIndex` / `primitiveIndex`, with `nodeIndex = -1` and the material index for OBJ) and a `fields` mask of the values to pin. See 3D Graphics — Editing an imported model's submeshes (p. 254).

Method details

loadModel

- bool **loadModel**(const std::string& filename)

Loads a 3D model file from disk. Doriax automatically detects the format based on file extension:

- `.obj` — Wavefront OBJ
- `.gltf` / `.glb` — GL Transmission Format (GLTF 2.0 / binary GLB)
- `.fbx` — loaded from its glTF conversion, which the editor makes and an export ships as `<name>.fbx.glb` (see 3D Graphics — FBX models (p. 251))

Returns `true` on success. On failure, check the log for details.

C++

```
Model character(&scene);
if (!character.loadModel("characters/hero.glb")) {
    Log::error("Failed to load hero model");
}
```

Lua

```lua
local character = Model(scene)
if not character:loadModel("characters/hero.glb") then
    Log.error("Failed to load hero model")
end
```

loadOBJ / loadGLTF

- bool **loadOBJ**(const std::string& filename)
- bool **loadGLTF**(const std::string& filename)

Format-specific loaders. Use these when you want to be explicit about the format, or when the file extension is non-standard. `loadGLTF()` accepts both text `.gltf` and binary `.glb` files.

getAnimation / findAnimation

- Animation (p. 573) **getAnimation**(int index)
- Animation (p. 573) **findAnimation**(const std::string& name)

Retrieve a skeletal or morph-target animation embedded in the loaded model. `getAnimation()` accesses by zero-based index; `findAnimation()` searches the animation entity name. GLTF import initializes that entity name from the clip name exported by the DCC tool, and an unnamed imported clip receives a generated name such as `"Animation 0"`. Both methods return an Animation (p. 573) object that can be started, stopped, and configured for looping.

Renaming an animation entity changes the string accepted by `findAnimation()` and the named playAnimation() overloads. The `AnimationComponent` itself has no separate name property.

C++

```cpp
Model soldier(&scene);
soldier.loadModel("soldiers/rifleman.glb");

Animation walkAnim = soldier.findAnimation("Walk");
walkAnim.setLoop(true);
walkAnim.start();
```

Lua

```lua
local soldier = Model(scene)
soldier:loadModel("soldiers/rifleman.glb")

local walkAnim = soldier:findAnimation("Walk")
walkAnim.loop = true
walkAnim:start()
```

playAnimation / stopAnimations

- void **playAnimation**(int index)
- void **playAnimation**(int index, float fadeTime)
- void **playAnimation**(const std::string& name)
- void **playAnimation**(const std::string& name, float fadeTime)
- void **stopAnimations**(float fadeTime)

Switch the model's active clip with a **smooth crossfade** instead of an instant pose snap. `playAnimation()` fades out every other running clip on this model and fades the requested one in over `fadeTime` seconds; `stopAnimations()` fades all running clips out.

This is the recommended way to change animation state at runtime (idle → run → jump → die). During the fade both clips play and their poses are blended by weight, so the character eases between motions rather

than popping. When `fadeTime` is omitted, the target clip's authored defaultFadeTime (p. 574) is used (set per-clip as **AnimationComponent → Fade time**). A `fadeTime` of `0` switches instantly.

Unlike getAnimation(), which returns a handle you start yourself, `playAnimation()` manages the transition for you. Set looping on the clips you intend to hold (idle, run) via Animation::loop (p. 573).

C++

```
Model hero(&scene);
hero.loadModel("characters/hero.glb");

// Make locomotion clips loop, then start on idle
hero.findAnimation("Idle").setLoop(true);
hero.findAnimation("Run").setLoop(true);
hero.playAnimation("Idle");          // uses each clip's Fade time

// Later, on input:
hero.playAnimation("Run", 0.2f);     // crossfade to run over 0.2s
hero.playAnimation("Die", 0.1f);     // snappy transition into a one-shot
```

Lua

```
local hero = Model(scene)
hero:loadModel("characters/hero.glb")

hero:findAnimation("Idle").loop = true
hero:findAnimation("Run").loop = true
hero:playAnimation("Idle")           -- uses each clip's Fade time

-- Later, on input:
hero:playAnimation("Run", 0.2)       -- crossfade to run over 0.2s
hero:playAnimation("Die", 0.1)       -- snappy transition into a one-shot
```

> **Crossfades assume full-skeleton clips**
>
> The blend is a weight-normalized average per bone, so it looks best when both clips animate the whole skeleton (the usual case for GLTF character clips). A bone animated by only the outgoing clip holds its pose until that clip finishes fading rather than easing.

getBone

- Bone (p. 413) **getBone**(const std::string& name)
- Bone (p. 413) **getBone**(int id)

Returns a Bone (p. 413) handle by joint name or by glTF node index (`id`). That integer is the node's index in the glTF file, which can differ from its ordinal in a skin's joint list. A `Bone` inherits Object (p. 455) so you can read and write its local transform to override the skeleton. For example, you can aim a character's head bone toward a target while the rest of the body plays a walk cycle.

C++

```
Bone head = soldier.getBone("Head");
head.setRotation(targetRotation);
```

Lua

```
local head = soldier:getBone("Head")
head.rotation = targetRotation
```

getMorphWeight / setMorphWeight

- float **getMorphWeight**(const std::string& name)
- float **getMorphWeight**(int id)

- void **setMorphWeight**(const std::string& name, float value)
- void **setMorphWeight**(int id, float value)

Read or write the weight of a blend-shape morph target. Weights range from `0.0` (no influence) to `1.0` (full influence). Multiple morphs can be active simultaneously. Access by exported name or by zero-based index.

When the model's meshes are child entities, each part numbers its own targets. A name finds the parts that have that target. An index sets every part, and reads the part that has morph targets.

C++

```cpp
Model face(&scene);
face.loadModel("characters/face.glb");
face.setMorphWeight("Smile", 0.8f);
face.setMorphWeight("Blink_L", 1.0f);
```

Lua

```lua
local face = Model(scene)
face:loadModel("characters/face.glb")
face:setMorphWeight("Smile", 0.8)
face:setMorphWeight("Blink_L", 1.0)
```

resetToBindPose

- void **resetToBindPose**()

Resets every imported node (or, on the legacy skeleton path, every bone) back to the bind pose defined in the model file, clearing any programmatic or animation-driven overrides. Useful when switching between animations or when stopping all playback. A mesh part that was moved away from its file node (p. 77) keeps its transform, since the file's pose is relative to another parent.

Object

Description

`Object` is the base class for all scene objects. It wraps an ECS entity and exposes the transform (position, rotation, scale), parent–child hierarchy, visibility, billboard orientation, and shortcuts to the physics body attachments.

Inherits: EntityHandle (p. 711)

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Sprite sprite(&scene);
sprite.setPosition(100.0f, 200.0f, 0.0f);
sprite.setVisible(true);
```

Lua

```lua
local sprite = Sprite(scene)
sprite.position = Vector3(100, 200, 0)
sprite.visible = true
```

Properties

Type	Name	Default	Languages
Vector3	position	`(0,0,0)`	C++ | Lua
Quaternion	rotation	identity	C++ | Lua
Vector3	scale	`(1,1,1)`	C++ | Lua
bool	visible	`true`	C++ | Lua
bool	visibleInHierarchy	`true`	C++ | Lua (read-only)
bool	billboard	`false`	C++ | Lua
bool	fakeBillboard	`false`	C++ | Lua
bool	cylindricalBillboard	`false`	C++ | Lua
Quaternion	billboardRotation	identity	C++ | Lua
Matrix4	localMatrix	identity	C++ | Lua
Matrix4	modelMatrix	—	C++ | Lua (read-only)
Matrix4	normalMatrix	—	C++ | Lua (read-only)

Methods

Returns	Name	Languages
void	addChild	C++ | Lua
void	removeChild	C++ | Lua
void	removeParent	C++ | Lua
void	moveToTop	C++ | Lua
void	moveUp	C++ | Lua
void	moveDown	C++ | Lua
void	moveToBottom	C++ | Lua
void	setPosition	C++ | Lua
Vector3	getPosition	C++
Vector3	getWorldPosition	C++ | Lua

Returns	Name	Languages
void	setRotation	C++ | Lua
Quaternion	getRotation	C++
Quaternion	getWorldRotation	C++ | Lua
void	setScale	C++ | Lua
Vector3	getScale	C++
Vector3	getWorldScale	C++ | Lua
void	setVisible	C++
bool	isVisible	C++
bool	isVisibleInHierarchy	C++
void	setBillboard	C++ | Lua
bool	isBillboard	C++
void	setFakeBillboard	C++
bool	isFakeBillboard	C++
void	setCylindricalBillboard	C++
bool	isCylindricalBillboard	C++
void	setBillboardRotation	C++ | Lua
Quaternion	getBillboardRotation	C++
void	setLocalMatrix	C++
Matrix4	getLocalMatrix	C++
Matrix4	getModelMatrix	C++
Matrix4	getNormalMatrix	C++
void	updateTransform	C++ | Lua
Body2D (p. 541)	getBody2D	C++ | Lua
void	removeBody2D	C++ | Lua
Body3D (p. 551)	getBody3D	C++ | Lua
void	removeBody3D	C++ | Lua
Ray (p. 649)	getRay	C++ | Lua
Transform (p. 696)	`getTransform`	Lua

Property details

position

- *Setter:* `void setPosition(Vector3 position)`
- *Setter:* `void setPosition(float x, float y, float z)`
- *Setter:* `void setPosition(float x, float y)` *(2D shorthand, z = 0)*
- *Getter:* `Vector3 getPosition() const`

Local position relative to the parent object (or world origin if no parent). Changing position does not automatically update children — that is handled by the scene's transform system each frame.

On an entity with a physics body this is not the way to move it from `onFixedUpdate` — see Body2D (p. 543) / Body3D (p. 555) `position`.

> **Returns a copy**
>
> `obj.position.y = 2` changes only the copy, not the object; assign the whole value back. The same goes for `rotation` and `scale` — see Changing one component of a property (p. 329).

C++
```
obj.setPosition(100.0f, 0.0f, -50.0f);
Vector3 pos = obj.getPosition();
```

Lua
```
obj.position = Vector3(100, 0, -50)
local pos = obj.position   -- a copy
pos.y = 10
obj.position = pos         -- assigning it back moves the object
```

rotation

- *Setter:* `void setRotation(Quaternion rotation)`
- *Setter:* `void setRotation(float xAngle, float yAngle, float zAngle)` *(Euler angles in degrees when* `Engine::useDegrees` *is true)*
- *Getter:* `Quaternion getRotation() const`

Local rotation relative to the parent. The Euler overload uses the same default as `Quaternion(x, y, z)` — **ZYX** order (see Quaternion (p. 645)).

C++
```
obj.setRotation(0.0f, 45.0f, 0.0f);  // 45° around Y (ZYX order)
```

Lua
```
obj.rotation = Quaternion(0, 45, 0)
```

scale

- *Setter:* `void setScale(Vector3 scale)`
- *Setter:* `void setScale(float factor)` *(uniform scale)*
- *Getter:* `Vector3 getScale() const`

Local scale relative to the parent.

visible

- *Setter:* `void setVisible(bool visible)`
- *Getter:* `bool isVisible() const`

Hides or shows the object. Its children are hidden with it but keep their own flag, so showing the object again shows only the children that are visible themselves. `setVisibleOnly` is gone for the same reason: `setVisible` no longer touches the children's flags, so call it instead.

Scenes saved by earlier editors, where showing a parent made all its children visible, are converted when opened: the children of a hidden entity are made visible, so showing it brings them back as before.

visibleInHierarchy

- *Getter:* `bool isVisibleInHierarchy() const`

`true` when the object and all of its ancestors are visible. The render system refreshes it on each update, so right after a `setVisible` call it still holds the previous value.

billboard

- *Setter:* `void setBillboard(bool billboard)`
- *Getter:* `bool isBillboard() const`

When `true`, the object's orientation is overridden each frame so it always faces the active camera. Position and scale are unaffected.

---

fakeBillboard

- *Setter:* `void setFakeBillboard(bool fakeBillboard)`
- *Getter:* `bool isFakeBillboard() const`

Fake billboard rotates the object to face the camera while keeping the local `position` offset in world space. Unlike a true billboard, the object moves as its parent moves but stays facing the camera.

---

cylindricalBillboard

- *Setter:* `void setCylindricalBillboard(bool cylindricalBillboard)`
- *Getter:* `bool isCylindricalBillboard() const`

Constrains billboard rotation to the Y axis only (vertical axis). The object faces the camera horizontally but does not tilt up or down.

---

billboardRotation

- *Setter:* `void setBillboardRotation(Quaternion rotation)`
- *Setter:* `void setBillboardRotation(float xAngle, float yAngle, float zAngle)`
- *Getter:* `Quaternion getBillboardRotation() const`

An additional rotation applied on top of the billboard face-camera rotation. Use to offset or spin billboard sprites.

---

localMatrix

- *Setter:* `void setLocalMatrix(Matrix4 localMatrix)`
- *Getter:* `Matrix4 getLocalMatrix() const`

Provides direct access to the local-space transform matrix. Setting this overrides position/rotation/scale decomposed values. Useful for loading pre-computed transforms from GLTF animations or procedural geometry.

---

modelMatrix

- *Getter:* `Matrix4 getModelMatrix() const`

Read-only world-space transform matrix, computed from the full parent chain. Updated by the transform system every frame.

---

normalMatrix

- *Getter:* `Matrix4 getNormalMatrix() const`

Inverse transpose of the model matrix. Used in shaders to correctly transform surface normals when non-uniform scale is applied.

Method details

addChild

- `void addChild(Object* child)`
- `void addChild(Entity child)`

Attaches `child` to this object's transform hierarchy. The child's `position`, `rotation`, and `scale` become relative to this object.

C++

```
Sprite parent(&scene);
Sprite child(&scene);
parent.addChild(&child);
```

Lua

```
local parent = Sprite(scene)
local child  = Sprite(scene)
parent:addChild(child)
```

removeChild

- `void removeChild(Object* child)`
- `void removeChild(Entity child)`

Detaches `child` from this hierarchy. The child's world transform is preserved by converting it back to world-space values.

removeParent

- `void removeParent()`

Detaches this object from its parent and re-parents it to the scene root. The world transform is preserved.

moveToTop / moveUp / moveDown / moveToBottom

Control the draw/update order of this object among its siblings.

- **moveToTop** — Drawn last (on top of siblings).
- **moveUp** — Moves one position toward the top.
- **moveDown** — Moves one position toward the bottom.
- **moveToBottom** — Drawn first (behind siblings).

C++

```
sprite.moveToTop();
```

Lua

```
sprite:moveToTop()
```

setPosition

- `void setPosition(Vector3 position)`
- `void setPosition(float x, float y, float z)`
- `void setPosition(float x, float y)`

Sets local position. See position property.

getWorldPosition

- `Vector3 getWorldPosition() const`

Returns the position of this object in world space, taking the full parent chain into account.

setRotation

- `void setRotation(Quaternion rotation)`
- `void setRotation(float xAngle, float yAngle, float zAngle)`

Sets local rotation. See rotation property.

getWorldRotation

- `Quaternion getWorldRotation() const`

Returns the combined world-space rotation after applying all parent rotations.

setScale

- `void setScale(Vector3 scale)`
- `void setScale(float factor)`

Sets local scale. The single-float overload applies uniform scale on all axes.

getWorldScale

- `Vector3 getWorldScale() const`

Returns the effective world-space scale, multiplied through the parent chain.

setVisible

- `void setVisible(bool visible)`
- `bool isVisible() const`

Shows or hides this object. Descendants are hidden with it but keep their own `visible` flag.

setBillboard

- `void setBillboard(bool billboard, bool fake, bool cylindrical)`
- `void setBillboard(bool billboard)`

Enables billboard mode. The three-argument overload allows setting `fake` and `cylindrical` flags in a single call. See billboard, fakeBillboard, and cylindricalBillboard.

updateTransform

- `void updateTransform()`

Forces an immediate recalculation of the world transform. The engine handles this automatically each frame; call only when you need an up-to-date transform outside the normal update order. Writing a transform from `onFixedUpdate` is one such case: the physics step reads the world transform, which is otherwise only refreshed once per frame.

getBody2D

- `Body2D getBody2D()`

Returns the Body2D (p. 541) physics handle for this object. Creates the body component if it does not already exist.

C++

```
Body2D body = sprite.getBody2D();
body.createBoxShape(64.0f, 64.0f);
body.setType(BodyType::DYNAMIC);
body.load();
```

Lua

```
local body = sprite:getBody2D()
body:createBoxShape(64, 64)
body.type = BodyType.DYNAMIC
body:load()
```

removeBody2D

- `void removeBody2D()`

Destroys the 2D physics body attached to this object.

getBody3D

- `Body3D getBody3D()`

Returns the Body3D (p. 551) physics handle for this object. Creates the body component if it does not already exist.

removeBody3D

- `void removeBody3D()`

Destroys the 3D physics body attached to this object.

getRay

- `Ray getRay(Vector3 direction)`

Constructs a Ray (p. 649) originating from this object's world position in the given world-space direction. Useful for raycasting from an object's location.

C++

```
Ray ray = obj.getRay(Vector3(0, -1, 0));  // ray pointing down
```

Lua

```
local ray = obj:getRay(Vector3(0, -1, 0))
```

Occluder2D

Inherits: Object (p. 455)
C++ type: `Occluder2D`

Description

A shadow caster for 2D lights. An `Occluder2D` defines an outline (a quad derived from the entity's mesh, or a custom polygon) that blocks light from every shadow-enabled Light2D (p. 430) in the scene.

The typical uses are:

- **Add the component to a sprite** with the default `AUTO_QUAD` shape — the sprite's bounds cast the shadow.
- **Create a standalone occluder entity** with a `POLYGON` shape — an invisible shadow wall you can shape freely. The editor's **2D → 2D Occluder** entry creates one with a starter square; polygon points can be dragged directly in the scene view or edited in Properties.

See 2D Graphics — 2D shadows (p. 245) for the usage guide.

Properties

Type	Name	Default	Langs
Occluder2DShape	shape	`AUTO_QUAD`	C++ | Lua
bool	closed	`true`	C++ | Lua
bool	enabled	`true`	C++ | Lua

Methods

Type	Name	Langs
void	setShape	C++
Occluder2DShape	getShape	C++
void	addVertex	C++ | Lua
void	clearVertices	C++ | Lua
unsigned int	getVertexCount	C++ | Lua
void	setClosed	C++
bool	isClosed	C++
void	setEnabled	C++
bool	isEnabled	C++
Occluder2DComponent (p. 682)	`getOccluder2DComponent`	Lua

Enumerations

Occluder2DShape

- **AUTO_QUAD** - The outline is derived automatically from the bounds of the mesh on the same entity (for example, a sprite quad). Casts nothing when the entity has no mesh.
- **POLYGON** - The outline is the component's custom point list in local space.

Property details

shape

- *Setter*: void **setShape**(Occluder2DShape shape)
- *Getter*: Occluder2DShape **getShape**() const

Selects how the occluder outline is defined. See Occluder2DShape. Calling addVertex switches the shape to `POLYGON` automatically.

addVertex

- void **addVertex**(Vector2 vertex)
- void **addVertex**(float x, float y)
- void **clearVertices**()
- unsigned int **getVertexCount**() const

Build a custom polygon outline in local space. Points are connected in order; with closed enabled the last point connects back to the first.

C++

```
Occluder2D wall(&scene);
wall.setPosition(Vector3(500, 200, 0));
wall.addVertex(-60, -20);
wall.addVertex(60, -20);
wall.addVertex(60, 20);
wall.addVertex(-60, 20);
```

Lua

```
local wall = Occluder2D(scene)
wall.position = Vector3(500, 200, 0)
wall:addVertex(-60, -20)
wall:addVertex(60, -20)
wall:addVertex(60, 20)
wall:addVertex(-60, 20)
```

closed

- *Setter*: void **setClosed**(bool closed)
- *Getter*: bool **isClosed**() const

Whether the polygon outline is a closed loop (last point connects to the first) or an open chain. Only relevant for the `POLYGON` shape — `AUTO_QUAD` is always closed.

enabled

- *Setter*: void **setEnabled**(bool enabled)
- *Getter*: bool **isEnabled**() const

Toggles the occluder without removing the component. Disabled occluders cast no shadows.

Points

Inherits: Object (p. 455)
C++ type: `Points`

Description

Renders a collection of 3D billboard points (sprites). Each point is a camera-facing quad with an individual position, colour, size, rotation, and optional texture atlas frame. Useful for particle-like effects, star fields, or any situation where a large number of individually controlled sprites is needed.

Like Lines (p. 433), `Points` uses a dynamic GPU buffer. Call `updatePoints()` after modifying point data.

Properties

Type	Name	Default	Langs
unsigned int	maxPoints	`100`	C++ | Lua
bool	transparent	`false`	C++ | Lua
bool	autoTransparency	`true`	C++ | Lua
string	customShader	`""`	C++ | Lua

Methods

Type	Name	Langs
bool	load	C++
void	addPoint	C++ | Lua
PointData (p. 714)	getPoint	C++ | Lua
void	updatePoint	C++ | Lua
void	removePoint	C++ | Lua
bool	isPointVisible	C++ | Lua
void	setPointVisible	C++ | Lua
void	updatePoints	C++ | Lua
size_t	getNumPoints	C++ | Lua
void	clearPoints	C++ | Lua
void	addSpriteFrame	C++ | Lua
void	removeSpriteFrame	C++ | Lua
void	setTexture	C++ | Lua
void	setShaderUniform	C++ | Lua
Vector4	getShaderUniform	C++ | Lua
bool	removeShaderUniform	C++ | Lua
PointsComponent (p. 684)	`getPointsComponent`	Lua

Property details

maxPoints

- *Setter*: void **setMaxPoints**(unsigned int maxPoints)
- *Getter*: unsigned int **getMaxPoints**() const

Pre-allocates GPU buffer space for this many points.

transparent / autoTransparency

- *Setter/Getter*: void **setTransparent**(bool) / bool **isTransparent**()
- *Setter/Getter*: void **setAutoTransparency**(bool) / bool **isAutoTransparency**()

Controls transparency rendering for this points object. See Mesh (p. 441) for details.

Method details

load

- bool **load**()

Explicitly initialises the GPU buffer. This method is **C++ only** and is normally optional because the render system initialises points automatically.

addPoint

Several overloads:

- void **addPoint**(PointData (p. 714) point)
- void **addPoint**(Vector3 position)
- void **addPoint**(Vector3 position, Vector4 color)
- void **addPoint**(Vector3 position, Vector4 color, float size)
- void **addPoint**(Vector3 position, Vector4 color, float size, float rotation)
- void **addPoint**(Vector3 position, Vector4 color, float size, float rotation, Rect textureRect)

Adds a new billboard point. The most complete overload specifies all visual attributes.

C++

```
Points stars(&scene);
stars.setTexture("particles/star.png");
stars.setMaxPoints(500);
stars.load();

for (int i = 0; i < 500; i++) {
    stars.addPoint(
        Vector3(rand() % 200 - 100, rand() % 200 - 100, 0),
        Vector4(1, 1, 1, 1), 2.0f
    );
}
stars.updatePoints();
```

Lua

```
local stars = Points(scene)
stars:setTexture("particles/star.png")
stars.maxPoints = 500

for i = 1, 500 do
    stars:addPoint(
        Vector3(math.random(-100, 100), math.random(-100, 100), 0),
        Vector4(1, 1, 1, 1), 2.0
    )
end
stars:updatePoints()
```

getPoint

- PointData (p. 714)& **getPoint**(size_t index)

Returns a reference to the `PointData` at `index`. Modify in-place then call `updatePoints()`.

updatePoint

Overloads mirror `addPoint()`. Replaces data at `index`.

removePoint

- void **removePoint**(size_t index)

Removes the point at `index`.

isPointVisible / setPointVisible

- bool **isPointVisible**(size_t index)
- void **setPointVisible**(size_t index, bool visible) const

Show or hide individual points without removing them from the buffer.

updatePoints

- void **updatePoints**()

Uploads changes to the GPU. Must be called after any modification.

getNumPoints

- size_t **getNumPoints**()

Returns the current number of points.

clearPoints

- void **clearPoints**()

Removes all points.

addSpriteFrame

- void **addSpriteFrame**(int id, const std::string& name, Rect rect)
- void **addSpriteFrame**(const std::string& name, float x, float y, float width, float height)
- void **addSpriteFrame**(float x, float y, float width, float height)
- void **addSpriteFrame**(Rect rect)
- void **removeSpriteFrame**(int id)
- void **removeSpriteFrame**(const std::string& name)

Defines a named atlas frame for use with the `textureRect` field of PointData (p. 714). This enables different points to show different sub-regions of the same texture atlas.

setTexture

- void **setTexture**(const std::string& path)
- void **setTexture**(const std::string& id, TextureData data)
- void **setTexture**(Framebuffer* framebuffer)

Sets the texture used for all points. When using a texture atlas, call addSpriteFrame to define the atlas regions.

customShader

- *Setter:* `void setCustomShader(const std::string& path)`
- *Getter:* `std::string getCustomShader() const`

Project-relative base path of a forked shader (`"shaders/sparks"` resolves to `shaders/sparks.vert` and `shaders/sparks.frag`, or `"a.vert|b.frag"` names the files separately). Empty (default) uses the scene default shader for this type, or the engine built-in. Changing it reloads the component. Same semantics as Mesh.customShader (p. 441); see Custom Shaders (p. 143).

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`
- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the custom shader declares with `#pragma param`, by name. Setting a value only rewrites the uploaded block, so it can run every frame; a parameter with no value reads the scene's value under a scene default shader (p. 405), else its declared default. Same semantics as Mesh.setShaderUniform (p. 445); see Custom Shaders — Shader parameters (p. 147).

C++

```cpp
points.setCustomShader("shaders/sparks");
points.setShaderUniform("strength", 0.8f);
points.setShaderUniform("glowColor", Vector4(1.0f, 0.8f, 0.2f, 1.0f));
```

Lua

```lua
points.customShader = "shaders/sparks"
points:setShaderUniform("strength", 0.8)
points:setShaderUniform("glowColor", Vector4(1, 0.8, 0.2, 1))
```

ReflectionProbe

Inherits: Object (p. 455)
C++ type: `ReflectionProbe`

Description

A local reflection environment. Meshes inside the probe's box-shaped influence volume sample its cubemap (box-projected, so reflections stay anchored to the volume) instead of the global sky environment. Meshes need **Receive IBL** and **Receive Lights** enabled to pick probes up.

Meshes with `renderInReflectionProbes` disabled are skipped by captures, which keeps an object that wraps around a probe out of its own reflection — see Mesh (p. 441).

Static probes use an authored cubemap — or capture the scene once at load when no cubemap is set. Dynamic probes re-capture at runtime according to their update mode, budgeted at one cubemap face per frame. See Rendering Pipeline — Reflection probes (p. 263) for how probes are selected, blended, and captured.

Properties

Type	Name	Default	Langs
ReflectionProbeMode	mode	`STATIC`	C++ | Lua
ReflectionProbeUpdateMode	updateMode	`ON_LOAD`	C++ | Lua
Vector3	boxOffset	`(0, 0, 0)`	C++ | Lua
Vector3	boxSize	`(10, 10, 10)`	C++ | Lua
float	blendDistance	`1.0`	C++ | Lua
float	intensity	`1.0`	C++ | Lua
int	priority	`0`	C++ | Lua
unsigned int	resolution	`128`	C++ | Lua
float	nearClip	`0.1`	C++ | Lua
float	farClip	`100.0`	C++ | Lua
float	updateInterval	`1.0`	C++ | Lua
bool	includeSky	`true`	C++ | Lua

Methods

Type	Name	Langs
void	setBoxOffset	C++ | Lua
void	setBoxSize	C++ | Lua
void	setClipPlanes	C++ | Lua
void	setTextures	C++ | Lua
void	setTexture	C++ | Lua
void	refresh	C++ | Lua
ReflectionProbeComponent (p. 686)	`getReflectionProbeComponent`	Lua

Enumerations

ReflectionProbeMode

- **STATIC** — Uses the authored cubemap (setTextures); with no cubemap set, captures the scene once at load and keeps that result.
- **DYNAMIC** — Re-captures the scene at runtime according to updateMode.

ReflectionProbeUpdateMode

- **ON_LOAD** — Captures once when the scene starts.
- **ON_MOVE** — Re-captures whenever the probe entity moves.
- **INTERVAL** — Re-captures every updateInterval seconds.
- **MANUAL** — Re-captures only when refresh is called.

Property details

mode

- *Setter*: void **setMode**(ReflectionProbeMode mode)
- *Getter*: ReflectionProbeMode **getMode**() const

Whether the probe is static or re-captures at runtime. Changing the mode restarts any in-flight capture.

C++

```
ReflectionProbe probe(&scene);
probe.setPosition(0.0f, 2.0f, 0.0f);
probe.setBoxSize(12.0f, 6.0f, 12.0f);
probe.setMode(ReflectionProbeMode::DYNAMIC);
probe.setUpdateMode(ReflectionProbeUpdateMode::MANUAL);
probe.refresh();   // capture now
```

Lua

```
local probe = ReflectionProbe(scene)
probe:setPosition(0, 2, 0)
probe:setBoxSize(12, 6, 12)
probe.mode = ReflectionProbeMode.DYNAMIC
probe.updateMode = ReflectionProbeUpdateMode.MANUAL
probe:refresh()   -- capture now
```

updateMode

- *Setter*: void **setUpdateMode**(ReflectionProbeUpdateMode updateMode)
- *Getter*: ReflectionProbeUpdateMode **getUpdateMode**() const

When a `DYNAMIC` probe re-captures. Ignored by static probes.

boxOffset / boxSize

- *Setter*: void **setBoxOffset**(Vector3 boxOffset) / **setBoxOffset**(float x, float y, float z)
- *Setter*: void **setBoxSize**(Vector3 boxSize) / **setBoxSize**(float x, float y, float z)
- *Getter*: Vector3 **getBoxOffset**() / **getBoxSize**() const

The influence volume, in the entity's local space. The box is world-axis-aligned, centred on the entity plus `boxOffset`, and scaled by the entity's world scale. The cubemap is always captured at the entity origin, not the offset centre.

blendDistance

- *Setter*: void **setBlendDistance**(float blendDistance)
- *Getter*: float **getBlendDistance**() const

Fade band inside the box edges where the probe blends into the sky IBL, hiding the seam at the volume boundary. Runtime blending is limited to the box's smallest half-extent.

intensity

- *Setter*: void **setIntensity**(float intensity)
- *Getter*: float **getIntensity**() const

Multiplier on the probe's reflection contribution.

priority

- *Setter*: void **setPriority**(int priority)
- *Getter*: int **getPriority**() const

When influence boxes overlap, the higher-priority probe wins; on a tie, the probe whose centre is nearest the mesh wins.

resolution

- *Setter*: void **setResolution**(unsigned int resolution)
- *Getter*: unsigned int **getResolution**() const

Capture cubemap face size, clamped to 16–1024 at capture time. Changing it restarts any in-flight capture. The capture is prefiltered into 128-pixel faces like an authored cubemap, so sizes above 128 make reflections smoother, not sharper.

nearClip / farClip

- *Setter*: void **setNearClip**(float nearClip) / **setFarClip**(float farClip)
- *Setter*: void **setClipPlanes**(float nearClip, float farClip)
- *Getter*: float **getNearClip**() / **getFarClip**() const

Clip planes of the capture cameras.

updateInterval

- *Setter*: void **setUpdateInterval**(float updateInterval)
- *Getter*: float **getUpdateInterval**() const

Seconds between captures for `INTERVAL` update mode.

includeSky

- *Setter*: void **setIncludeSky**(bool includeSky)
- *Getter*: bool **isIncludeSky**() const

Whether the sky (and scene background colour) appears in runtime captures.

Method details

setTextures / setTexture

- void **setTextures**(const std::string& texturePositiveX, const std::string& textureNegativeX, const std::string& texturePositiveY, const std::string& textureNegativeY, const std::string& texturePositiveZ, const std::string& textureNegativeZ)
- void **setTexture**(const std::string& texture)

Assigns the authored cubemap used by `STATIC` probes — six face image paths, or a single cross-layout image. Authored cubemaps are GGX-prefiltered like the sky environment, so rough surfaces get correct blurry reflections.

refresh

- void **refresh**()

Requests a new capture (or re-bake of the authored cubemap), regardless of mode and update mode. This is what the editor's **Refresh Probe** button calls, and the way to update a `MANUAL` probe from gameplay code.

Shape

Inherits: Mesh (p. 436)
C++ type: `Shape`

Procedural mesh factory (box, sphere, etc.). Each `create*` call replaces the mesh geometry; call it once after constructing the `Shape`.

Methods

Name	Languages	Geometry
`createPlane`	C++ | Lua	Flat quad in the XZ plane, normal `+Y` (lies down, like a floor)
`createWall`	C++ | Lua	Upright quad in the XY plane, normal `+Z` (stands up, faces the camera)
`createBox`	C++ | Lua	Box centred on the origin
`createSphere`	C++ | Lua	UV sphere
`createCylinder`	C++ | Lua	Cylinder or cone (separate base/top radii)
`createCapsule`	C++ | Lua	Capsule (cylinder with hemispherical caps)
`createTorus`	C++ | Lua	Torus / ring

```
Shape floor(&scene);
floor.createPlane(20.0f, 20.0f);     // width, depth — horizontal

Shape wall(&scene);
wall.createWall(10.0f, 10.0f);       // width, height — vertical
```

A **Wall** is the natural surface for a mirror (p. 269): its `+Z` normal matches the default `MirrorComponent` normal, so it reflects without any rotation. The Mirror (p. 448) subclass pairs the two — a `Shape` that is already reflective.

SkyBox

Inherits: EntityHandle (p. 711)
C++ type: `SkyBox`

Cubemap sky rendering and the source environment for **image-based lighting (IBL)**.

When a Sky entity has a valid cubemap texture, the engine generates irradiance and prefiltered specular environment maps from it. Meshes with **Receive IBL** enabled on their Mesh component use those maps for reflections and indirect diffuse lighting.

Editor-only component fields

These are edited on the **Sky** component in the Properties window and serialized with the scene (not yet exposed on the `SkyBox` script class):

Field	Default	Purpose
visible	`true`	Draw the sky background. When `false`, the sky is hidden but IBL maps are still built from the texture.
texture	—	Cubemap source (six faces or single file).
color	white	Linear RGBA tint.
rotation	`0`	Y-axis rotation in degrees.

Use **Visible = false** for invisible HDR environments that only drive reflections.

Properties

Name	Languages
`color`	C++ | Lua
`alpha`	C++ | Lua
`rotation`	C++ | Lua
`customShader`	C++ | Lua

Methods

Name	Languages
`setTextures`	C++ | Lua
`setTexture`	C++ | Lua
`setTexturePositiveX`	C++ | Lua
`setTextureNegativeX`	C++ | Lua
`setTexturePositiveY`	C++ | Lua
`setTextureNegativeY`	C++ | Lua
`setTexturePositiveZ`	C++ | Lua
`setTextureNegativeZ`	C++ | Lua
`setColor`	C++ | Lua
`setShaderUniform`	C++ | Lua
`getShaderUniform`	C++ | Lua
`removeShaderUniform`	C++ | Lua
`getSkyComponent` (SkyComponent (p. 688))	Lua

Example

C++

```
SkyBox sky(&scene);
// faces in +X, -X, +Y, -Y, +Z, -Z order
sky.setTextures(
    "sky/px.png", "sky/nx.png",
    "sky/py.png", "sky/ny.png",
    "sky/pz.png", "sky/nz.png");
sky.setRotation(15.0f);
```

Lua

```
local sky = SkyBox(scene)
-- faces in +X, -X, +Y, -Y, +Z, -Z order
sky:setTextures(
    "sky/px.png", "sky/nx.png",
    "sky/py.png", "sky/ny.png",
    "sky/pz.png", "sky/nz.png")
sky.rotation = 15
```

Enable **Receive IBL** on meshes that should reflect this environment. See Rendering Pipeline — IBL (p. 263).

Method details

customShader

- *Setter:* `void setCustomShader(const std::string& path)`
- *Getter:* `std::string getCustomShader() const`

Project-relative base path of a forked shader (`"shaders/dayNight"` resolves to `shaders/dayNight.vert` and `shaders/dayNight.frag`, or `"a.vert|b.frag"` names the files separately). Empty (default) uses the scene default shader for this type, or the engine built-in. Changing it reloads the component. Same semantics as Mesh.customShader (p. 441); see Custom Shaders (p. 143).

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`
- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the custom shader declares with `#pragma param`, by name. Setting a value only rewrites the uploaded block, so it can run every frame; a parameter with no value reads the scene's value under a scene default shader (p. 405), else its declared default. Same semantics as Mesh.setShaderUniform (p. 445); see Custom Shaders — Shader parameters (p. 147).

C++

```
sky.setCustomShader("shaders/dayNight");
sky.setShaderUniform("strength", 0.8f);
sky.setShaderUniform("glowColor", Vector4(1.0f, 0.8f, 0.2f, 1.0f));
```

Lua

```
sky.customShader = "shaders/dayNight"
sky:setShaderUniform("strength", 0.8)
sky:setShaderUniform("glowColor", Vector4(1, 0.8, 0.2, 1))
```

See also

- Custom Shaders — Shader parameters (p. 147)
- Mesh — receiveIBL (p. 441)
- Texture — cube maps (p. 738)
- Properties — Sky component (p. 98)

Sound

Inherits: EntityHandle (p. 711)
C++ type: `Sound`

Description

Plays audio clips in a scene, supporting both 2D and 3D positional audio. `Sound` wraps the SoLoud[23] audio engine, giving you precise control over playback state, volume, pitch, panning, looping, 3D attenuation and Doppler effect.

A 2D sound is heard at the same volume and position regardless of where in the scene the listener or source is placed; a 3D sound is automatically attenuated based on the distance between the sound entity and the active camera.

Constructors

All three overloads are available in C++ and Lua:

- `Sound(Scene* scene)` — creates a new 2D sound entity.
- `Sound(Scene* scene, bool is3D)` — creates a new sound entity, adding a transform when `is3D` is `true`.
- `Sound(Scene* scene, Entity entity)` — wraps an existing sound entity without taking ownership or adding components.

In Lua, a numeric second argument is an entity ID; use `true` or `false` to choose the 3D/2D creation overload. For example:

```
local sound2D = Sound(scene)
local sound3D = Sound(scene, true)
local existingSound = Sound(scene, soundEntity)
```

Keep newly created owning handles alive while you need their entities. See EntityHandle ownership (p. 712).

Properties

Type	Name	Default	Langs
bool	sound3D	`false`	C++ | Lua
bool	clockedSound	`false`	C++ | Lua
double	volume	`1.0`	C++ | Lua
float	speed	`1.0`	C++ | Lua
float	pan	`0.0`	C++ | Lua
bool	looping	`false`	C++ | Lua
double	loopingPoint	`0.0`	C++ | Lua
bool	protectVoice	`false`	C++ | Lua
float	minDistance	`1.0`	C++ | Lua
float	maxDistance	`1000.0`	C++ | Lua
SoundAttenuation	attenuationModel	`NO_ATTENUATION`	C++ | Lua
float	attenuationRolloffFactor	`1.0`	C++ | Lua
float	dopplerFactor	`1.0`	C++ | Lua

23. https://solhsa.com/soloud/

Methods

Type	Name	Langs
int	loadSound	C++ | Lua
void	destroySound	C++ | Lua
Object (p. 455)	getObject	C++ | Lua
void	play	C++ | Lua
void	pause	C++ | Lua
void	stop	C++ | Lua
void	seek	C++ | Lua
double	getLength	C++ | Lua
double	getPlayingTime	C++ | Lua
bool	isPlaying	C++ | Lua
bool	isPaused	C++ | Lua
bool	isStopped	C++ | Lua
void	setInaudibleBehavior	C++ | Lua
void	setMinMaxDistance	C++ | Lua

Enumerations

SoundAttenuation

Controls how volume falls off with distance in 3D audio.

- **NO_ATTENUATION** - No distance attenuation; the sound is heard at full volume everywhere
- **INVERSE_DISTANCE** - Classic `1 / distance` model, similar to the OpenAL default
- **LINEAR_DISTANCE** - Linear ramp from `minDistance` to `maxDistance`
- **EXPONENTIAL_DISTANCE** - Exponential roll-off; volume drops quickly after `minDistance`

Property details

sound3D

- *Setter*: void **setSound3D**(bool sound3D)
- *Getter*: bool **isSound3D**() const

When `true`, the sound is rendered in 3D space relative to the active camera. The entity's world position is used as the source position. When `false`, the sound is a standard stereo/mono clip with no positional information.

clockedSound

- *Setter*: void **setClockedSound**(bool enableClocked)
- *Getter*: bool **isClockedSound**() const

When enabled, SoLoud plays the sound in *clocked* mode. This keeps multiple streamed sounds synchronised to the engine's update tick, which is important when mixing music with synchronised sound effects. Has no effect on loaded WAV/OGG samples (`Wav`).

volume

- *Setter*: void **setVolume**(double volume)
- *Getter*: double **getVolume**() const

Master volume for this source. A value of `1.0` is full-volume, `0.0` is silent. Values above `1.0` are allowed but may clip the audio output.

---

speed

- *Setter*: void **setSpeed**(float speed)
- *Getter*: float **getSpeed**() const

Playback speed (pitch) multiplier. `1.0` is normal, `2.0` plays at double speed (one octave up), `0.5` plays at half speed (one octave down).

---

pan

- *Setter*: void **setPan**(float pan)
- *Getter*: float **getPan**() const

Stereo pan. `-1.0` is fully left, `0.0` is centre, `1.0` is fully right. Panning has no effect when `sound3D` is `true`.

---

looping

- *Setter*: void **setLooping**(bool looping)
- *Getter*: bool **isLooping**() const

When `true`, the clip restarts from loopingPoint after it finishes.

---

loopingPoint

- *Setter*: void **setLoopingPoint**(double loopingPoint)
- *Getter*: double **getLoopingPoint**() const

Position in **seconds** at which the clip restarts when looping is `true`. Defaults to `0.0` (beginning of the clip).

---

protectVoice

- *Setter*: void **setProtectVoice**(bool protectVoice)
- *Getter*: bool **isProtectVoice**() const

If `true`, SoLoud will never steal this voice when all voice slots are occupied. Use for critical audio (e.g. player death sounds) that must always play.

---

minDistance / maxDistance

- *Setter*: void **setMinDistance**(float minDistance)
- *Getter*: float **getMinDistance**() const
- *Setter*: void **setMaxDistance**(float maxDistance)
- *Getter*: float **getMaxDistance**() const
- *Combined setter*: void **setMinMaxDistance**(float minDistance, float maxDistance)

Distance thresholds used by the attenuationModel when `sound3D` is `true`.

- Inside `minDistance` the sound plays at full volume.
- Beyond `maxDistance` the sound is inaudible (for `LINEAR_DISTANCE`) or effectively silent.

attenuationModel

- *Setter*: void **setAttenuationModel**(SoundAttenuation attenuationModel)
- *Getter*: SoundAttenuation **getAttenuationModel**() const

Select the distance-attenuation algorithm for this 3D source. See SoundAttenuation for the available values.

attenuationRolloffFactor

- *Setter*: void **setAttenuationRolloffFactor**(float attenuationRolloffFactor)
- *Getter*: float **getAttenuationRolloffFactor**() const

Scales how aggressively the attenuation model reduces volume with distance. Higher values mean faster roll-off. The exact meaning depends on the attenuationModel in use.

dopplerFactor

- *Setter*: void **setDopplerFactor**(float dopplerFactor)
- *Getter*: float **getDopplerFactor**() const

Controls the intensity of the Doppler pitch shift for this 3D source. `0.0` disables Doppler; `1.0` is physically accurate; values above `1.0` exaggerate the effect. Requires `sound3D` to be `true`.

Method details

loadSound

- int **loadSound**(const std::string& filename)

Loads an audio file from disk. Supported formats depend on the SoLoud build bundled with Doriax (WAV, OGG, FLAC, MP3, and others). Returns `1` once the file is loaded, and `0` while it is still loading with async loading (p. 367), when it failed, or before the view is ready. Loading a different file replaces the current one.

C++

```cpp
Sound bgm(&scene);
bgm.loadSound("sounds/music.ogg");
bgm.setLooping(true);
bgm.play();
```

Lua

```lua
local bgm = Sound(scene)
bgm:loadSound("sounds/music.ogg")
bgm.looping = true
bgm:play()
```

destroySound

- void **destroySound**()

Unloads the audio data and frees the SoLoud voice slot. Call this when the sound is no longer needed to free memory immediately without destroying the entity.

getObject

- Object (p. 455) **getObject**() const

Returns the `Object` component of this sound entity. Use this to set the world-space position for 3D audio or to attach the sound to a parent entity.

C++

```
Sound footstep(&scene, true);  // 3D sound
footstep.loadSound("sounds/step.wav");
footstep.getObject().setPosition(playerPos);
```

Lua

```
local footstep = Sound(scene, true)
footstep:loadSound("sounds/step.wav")
local source = footstep:getObject()
source.position = playerPos
```

play / pause / stop

- void **play**()
- void **pause**()
- void **stop**()

Control the playback state of the sound. `play()` starts or resumes from the current position, `pause()` freezes playback at the current position, and `stop()` resets the position to the beginning.

seek

- void **seek**(double time)

Jump to a specific position in **seconds** within the audio clip. Has no effect if no clip is loaded.

getLength / getPlayingTime

- double **getLength**()
- double **getPlayingTime**()

`getLength()` returns the total duration of the loaded clip in **seconds**. `getPlayingTime()` returns the current playback cursor position in **seconds**.

isPlaying / isPaused / isStopped

- bool **isPlaying**()
- bool **isPaused**()
- bool **isStopped**()

Query the current playback state.

setInaudibleBehavior

- void **setInaudibleBehavior**(bool mustTick, bool kill)

Controls what SoLoud does when this voice becomes inaudible (volume reaches zero or the source is out of range).

- **mustTick** — when `true`, the voice continues to advance its playback position even while inaudible, so it stays in sync with other sources.
- **kill** — when `true`, the voice is automatically removed once it becomes inaudible, freeing a voice slot.

If both are `false`, the voice is simply paused while inaudible and resumes when it becomes audible again.

Sprite

Description

`Sprite` is the primary class for 2D images and sprite sheets. It renders a textured quad, supports texture atlas frames for manual sprite selection, and provides a simple frame animation player for looping sequences.

Inherits: Mesh (p. 436) → Object (p. 455) → EntityHandle (p. 711)

C++

```cpp
#include "Doriax.h"
using namespace doriax;

Sprite sprite(&scene);
sprite.setTexture("sprites/character.png");
sprite.setSize(64, 64);
sprite.createSprite();
```

Lua

```lua
local sprite = Sprite(scene)
sprite:setTexture("sprites/character.png")
sprite:setSize(64, 64)
```

Properties

Type	Name	Default	Languages
unsigned int	width	`0`	C++ | Lua
unsigned int	height	`0`	C++ | Lua
bool	flipY	`false`	C++ | Lua
float	textureScaleFactor	`0.0`	C++ | Lua
Rect	textureRect	full texture	C++ | Lua
PivotPreset	pivotPreset	`CENTER`	C++ | Lua

Methods

Returns	Name	Languages
bool	createSprite	C++
Occluder2D (p. 462)	getOccluder2D	C++ | Lua
void	removeOccluder2D	C++ | Lua
void	setSize	C++ | Lua
void	setWidth	C++ | Lua
void	setHeight	C++ | Lua
unsigned int	getWidth	C++ | Lua
unsigned int	getHeight	C++ | Lua
void	setFlipY	C++ | Lua
bool	isFlipY	C++ | Lua
void	setTextureScaleFactor	C++ | Lua
float	getTextureScaleFactor	C++ | Lua
void	setTextureRect	C++ | Lua
Rect	getTextureRect	C++ | Lua
void	setPivotPreset	C++ | Lua

Returns	Name	Languages
PivotPreset	getPivotPreset	C++ | Lua
void	addFrame	C++ | Lua
void	removeFrame	C++ | Lua
void	setFrame	C++ | Lua
void	startAnimation	C++ | Lua
void	pauseAnimation	C++ | Lua
void	resumeAnimation	C++ | Lua
void	stopAnimation	C++ | Lua
SpriteComponent (p. 691)	`getSpriteComponent`	Lua

Enumerations

PivotPreset

Controls the origin point used for position and rotation.

- **CENTER** — Origin at the geometric center of the sprite (default).
- **TOP_LEFT** — Origin at the top-left corner.
- **TOP_CENTER** — Origin at the top-center.
- **TOP_RIGHT** — Origin at the top-right corner.
- **CENTER_LEFT** — Origin at the center-left edge.
- **CENTER_RIGHT** — Origin at the center-right edge.
- **BOTTOM_LEFT** — Origin at the bottom-left corner.
- **BOTTOM_CENTER** — Origin at the bottom-center.
- **BOTTOM_RIGHT** — Origin at the bottom-right corner.

Property details

width / height

- *Setter:* `void setWidth(unsigned int width)` / `void setHeight(unsigned int height)`
- *Setter combined:* `void setSize(unsigned int width, unsigned int height)`
- *Getter:* `unsigned int getWidth() const` / `unsigned int getHeight() const`

Physical size of the rendered quad in world units. If both are zero, the sprite uses the texture dimensions automatically after `createSprite()`.

flipY

- *Setter:* `void setFlipY(bool flipY)`
- *Getter:* `bool isFlipY() const`

Flips the texture vertically. Useful when loading textures from APIs that use a top-left origin (e.g. some framebuffer captures).

textureScaleFactor

- *Setter:* `void setTextureScaleFactor(float textureScaleFactor)`
- *Getter:* `float getTextureScaleFactor() const`

Insets the UV rectangle by this many texels on every side, `0.0` by default. A small value such as `0.5` pulls the sampled area away from the edges of a packed atlas cell, hiding the neighbouring pixels that bilinear filtering would otherwise bleed in.

textureRect

- *Setter:* `void setTextureRect(float x, float y, float width, float height)`
- *Setter:* `void setTextureRect(Rect textureRect)`
- *Getter:* `Rect getTextureRect() const`

Defines the sub-rectangle of the texture to display in **UV coordinates**, where `Rect(0, 0, 1, 1)` covers the full texture. Use addFrame and setFrame for pixel-based atlas regions.

pivotPreset

- *Setter:* `void setPivotPreset(PivotPreset pivotPreset)`
- *Getter:* `PivotPreset getPivotPreset() const`

Sets the pivot (origin) point used for position placement and rotation. See PivotPreset.

Method details

createSprite

- `bool createSprite()`

Builds the quad geometry and uploads it to the GPU. Optional in most cases — the mesh system rebuilds sprites automatically — but useful in C++ when you need the geometry immediately. **Not bound to Lua.**

C++

```
Sprite sprite(&scene);
sprite.setTexture("ui/button.png");
sprite.setSize(128, 64);
sprite.createSprite();
```

getOccluder2D

- `Occluder2D getOccluder2D()`
- `void removeOccluder2D()`

Attaches an Occluder2D (p. 462) component to the sprite's entity if one does not already exist, then returns an `Occluder2D` handle for configuration. The default `AUTO_QUAD` shape uses the sprite's mesh bounds, so it is the quickest way to make a sprite cast shadows from Light2D (p. 430).

Use `removeOccluder2D()` to remove the occluder component from the sprite.

C++

```
Sprite crate(&scene);
crate.setTexture("crate.png");
crate.setSize(96, 96);

Occluder2D occluder = crate.getOccluder2D();
occluder.setShape(Occluder2DShape::AUTO_QUAD);
```

Lua

```lua
local crate = Sprite(scene)
crate:setTexture("crate.png")
crate:setSize(96, 96)

local occluder = crate:getOccluder2D()
occluder.shape = Occluder2DShape.AUTO_QUAD
```

setSize

- `void setSize(unsigned int width, unsigned int height)`

Sets both width and height in a single call. Equivalent to calling `setWidth` and `setHeight` separately.

setTextureRect

- `void setTextureRect(float x, float y, float width, float height)`
- `void setTextureRect(Rect textureRect)`

Restricts rendering to a sub-region of the texture using **UV coordinates**. Divide pixel x/width by the texture width and pixel y/height by the texture height. This setter applies immediately and cancels any pending frame selection waiting for texture dimensions. `getTextureRect()` returns the currently applied UV rectangle.

C++

```cpp
// Show the top-left 64×64 region of a 256×256 atlas.
sprite.setTextureRect(0.0f, 0.0f, 0.25f, 0.25f);
```

Lua

```lua
sprite:setTextureRect(0, 0, 0.25, 0.25)
```

addFrame

- `void addFrame(int id, const std::string& name, Rect rect)`
- `void addFrame(const std::string& name, float x, float y, float width, float height)`
- `void addFrame(float x, float y, float width, float height)` *(auto-increment id)*
- `void addFrame(Rect rect)` *(auto-increment id)*

Registers a named frame in the sprite sheet. Frames are used by setFrame and startAnimation.

Frame rectangles may use pixels or normalized UVs. If all four rectangle values are in `[0, 1]`, the engine treats them as UVs; otherwise it converts them from pixels using the texture dimensions when the frame is applied.

C++

```cpp
sprite.addFrame(0, "idle",  Rect(  0, 0, 64, 64));
sprite.addFrame(1, "run_1", Rect( 64, 0, 64, 64));
sprite.addFrame(2, "run_2", Rect(128, 0, 64, 64));
```

Lua

```lua
sprite:addFrame(0, "idle",  Rect(  0, 0, 64, 64))
sprite:addFrame(1, "run_1", Rect( 64, 0, 64, 64))
sprite:addFrame(2, "run_2", Rect(128, 0, 64, 64))
```

removeFrame

- `void removeFrame(int id)`
- `void removeFrame(const std::string& name)`

Removes a registered frame by ID or name.

setFrame

- `void setFrame(int id)`
- `void setFrame(const std::string& name)`

Selects the registered frame with the given ID or name. You can call this while configuring a new sprite, before its geometry has been built.

UV rectangles and pixel rectangles whose texture dimensions are known apply immediately. If dimensions are unavailable, the sprite retains the requested rectangle and retries during mesh updates. The currently applied rectangle stays unchanged until conversion succeeds.

Each sprite stores one pending rectangle: repeated calls replace it, so the latest selection wins. For example, `setFrame(1)` followed by `setFrame(2)` before loading finishes eventually displays frame 2. Different sprites keep independent pending selections. Use startAnimation for a timed sequence of frames.

Calling `setFrame()` does not stop an active animation; its next frame update can replace the selection. A direct setTextureRect call cancels a pending selection. Stopping editor play also clears pending runtime selections when the saved sprite state is restored.

C++

```
sprite.setFrame("idle");
```

Lua

```
sprite:setFrame("idle")
```

startAnimation

- `void startAnimation(std::vector<int> frames, std::vector<int> framesTime, bool loop)`
- `void startAnimation(int startFrame, int endFrame, int interval, bool loop)`
- `void startAnimation(const std::string& name, int interval, bool loop)`

Plays a frame animation. Time values are in milliseconds.

- The first overload accepts explicit frame ID lists and per-frame durations.
- The second overload plays a contiguous range of frame IDs at a fixed interval. The range includes both ends, so `startAnimation(0, 0, 100, false)` shows frame 0 alone.
- The third overload plays all frames whose names begin with `name` at a fixed interval.

C++

```
// Play frames 1 and 2 alternating, 150 ms each, looping
sprite.startAnimation({1, 2}, {150, 150}, true);
// or range:
sprite.startAnimation(1, 2, 150, true);
```

Lua

```
sprite:startAnimation(1, 2, 150, true)
```

pauseAnimation

- `void pauseAnimation()`

Freezes the animation at the current frame. Call resumeAnimation to continue.

resumeAnimation

- `void resumeAnimation()`

Resumes a paused animation from where it was stopped.

stopAnimation

- `void stopAnimation()`

Stops the animation and resets to the first frame of the sequence.

Terrain

Inherits: Mesh (p. 436)
C++ type: `Terrain`

Description

Generates and renders a heightmap-based terrain mesh using a chunked LOD (Level of Detail) approach. The terrain is built from a greyscale heightmap texture where brighter pixels correspond to higher elevation. Up to three blend maps control how up to nine detail layers are mixed over the base texture, three layers per blend map. A layer blends its colour alone, or carries its own PBR surface — see TerrainSurfaceLayer.

The terrain geometry is a CDLOD quadtree: `rootGridSize` × `rootGridSize` root nodes cover the terrain and each node subdivides into four until `levels` is reached. Every node is drawn with the same `resolution` × `resolution` grid and morphs into its coarser neighbour, which keeps the polycount roughly constant regardless of terrain size.

Properties

Type	Name	Default	Langs
float	size	`200.0`	C++ | Lua
float	maxHeight	`5.0`	C++ | Lua
int	resolution	`32`	C++ | Lua
int	textureBaseTiles	`1`	C++ | Lua
int	textureDetailTiles	`20`	C++ | Lua
int	rootGridSize	`2`	C++ | Lua
int	levels	`6`	C++ | Lua
float	foliageDistanceScale	`1.0`	C++ | Lua

Methods

Type	Name	Langs
bool	createTerrain	C++
void	setHeightMap	C++ | Lua
void	setBlendMap	C++ | Lua
void	setTextureLayer	C++ | Lua
void	setSurfaceLayer	C++ | Lua
TerrainSurfaceLayer	getSurfaceLayer	C++ | Lua
void	setLayerFromMaterial	C++ | Lua
void	removeSurfaceLayer	C++ | Lua
unsigned int	getNumLayers	C++ | Lua
void	setTextureDetailRed	C++ | Lua
void	setTextureDetailGreen	C++ | Lua
void	setTextureDetailBlue	C++ | Lua
TerrainComponent (p. 692)	`getTerrainComponent`	Lua

Property details

size

- *Setter*: void **setSize**(float size)
- *Getter*: float **getSize**() const

World-space width and depth of the terrain in units. The terrain is always square.

maxHeight

- *Setter*: void **setMaxHeight**(float maxHeight)
- *Getter*: float **getMaxHeight**() const

The maximum elevation in world units corresponding to a fully white (255) heightmap pixel.

resolution

- *Setter*: void **setResolution**(int resolution)
- *Getter*: int **getResolution**() const

Number of grid segments per side of a single LOD node (not the heightmap size). Every node in the quadtree is drawn with this grid, so raising it densifies the geometry everywhere.

Rounded to the nearest multiple of 4 (minimum 4): nodes stitch to coarser neighbours by morphing only their odd vertices, and the internal half-resolution grid (`resolution / 2`) must keep the same parity. A value that is not a multiple of 4 is rewritten on the component and logs a warning, so read the property back if the exact value matters.

textureBaseTiles / textureDetailTiles

- *Setter/Getter*: **setTextureBaseTiles** / **getTextureBaseTiles**, **setTextureDetailTiles** / **getTextureDetailTiles**

How many times the base and detail textures tile across the full terrain. Higher values produce smaller, more repetitive texture patterns; lower values stretch the texture.

rootGridSize

- *Setter*: void **setRootGridSize**(int rootGridSize)
- *Getter*: int **getRootGridSize**() const

The number of root quadtree nodes per side of the terrain. The coarsest node covers `size / rootGridSize` world units, so larger values start the quadtree with smaller nodes. At most 4, and capped by the terrain node budget together with `levels`: a larger value leaves the terrain unbuilt and logs an error.

levels

- *Setter*: void **setLevels**(int levels)
- *Getter*: int **getLevels**() const

Depth of the LOD quadtree. Each level halves the node size, so the leaf node covers `size / (rootGridSize * 2^(levels-1))` world units — more levels means finer geometry near the camera, not a longer view distance.

The quadtree materializes `rootGridSize^2 * (4^levels - 1) / 3` nodes, so the node count grows exponentially with `levels`; past the engine's budget of two million nodes, or above 20 levels, the terrain refuses to build and logs an error.

With automatic ranges the LOD distances follow from these node sizes: the first range is twice the leaf node size and each level doubles it, with only the last range stretched to the camera's far clip.

foliageDistanceScale

- *Setter*: void **setFoliageDistanceScale**(float scale)
- *Getter*: float **getFoliageDistanceScale**() const

Multiplies the draw distance and min distance of every foliage layer, for a quality setting: `0.5` draws foliage half as far, `1` as authored. Negative values clamp to `0`, which hides the foliage. It applies on the next frame, is not saved with the scene, and the editor's foliage preview ignores it.

Method details

createTerrain

- bool **createTerrain**()

Explicitly builds the terrain geometry and GPU buffers after its source properties have been configured. This method is **C++ only** and is normally optional because the mesh system builds and rebuilds terrain automatically.

C++

```
Terrain ground(&scene);
ground.setHeightMap("terrain/heightmap.png");
ground.setBlendMap("terrain/blendmap.png");
ground.setTexture("terrain/grass.png");
ground.setTextureLayer(0, "terrain/rock.png");
ground.setTextureLayer(1, "terrain/sand.png");
ground.setTextureLayer(2, "terrain/snow.png");
ground.setSize(1000.0f);
ground.setMaxHeight(80.0f);
ground.createTerrain();
```

Lua

```
local ground = Terrain(scene)
ground:setHeightMap("terrain/heightmap.png")
ground:setBlendMap("terrain/blendmap.png")
ground:setTexture("terrain/grass.png")
ground:setTextureLayer(0, "terrain/rock.png")
ground:setTextureLayer(1, "terrain/sand.png")
ground:setTextureLayer(2, "terrain/snow.png")
ground.size = 1000
ground.maxHeight = 80
```

setHeightMap

- void **setHeightMap**(const std::string& path)
- void **setHeightMap**(Framebuffer* framebuffer)

Sets the greyscale heightmap image. Each pixel's brightness maps linearly to height: black = 0, white = `maxHeight`. 8-bit and 16-bit images are both read (the editor's sculpting tools write 16-bit to avoid terracing). The image is stretched once over the terrain and sampled clamp-to-edge, so its outer texels define the terrain border.

Each quadtree node keeps the min/max height of the heightmap texels under its footprint, and that range is the node's bounding box for frustum culling. A framebuffer heightmap has no CPU-side pixels, so its nodes get a flat (zero-height) box and CPU-side height queries such as terrain picking read as flat — the

displacement still renders, but do not rely on culling or picking accuracy for procedurally generated heightmaps.

setBlendMap

- void **setBlendMap**(const std::string& path)
- void **setBlendMap**(Framebuffer* framebuffer)
- void **setBlendMap**(unsigned int index, const std::string& path) — Lua: **setBlendMapIndex**

Sets a blend map. Each blend map weights three detail layers through its RGB channels: blend map 0 weights layers 0–2, blend map 1 layers 3–5, and blend map 2 layers 6–8. The one-argument form is blend map 0. Whatever weight the layers leave unclaimed goes to the base texture, so areas with no blend-map colour show the base alone.

Alpha is not a fourth weight — saved maps are opaque, which would read as full strength. Indexes at or above three are rejected with an error.

setTextureLayer

- void **setTextureLayer**(unsigned int index, const std::string& path)

Sets the **colour** of one detail layer, 0 to 8, and nothing else — a layer that carries a PBR surface keeps its other maps and factors. The vector of layers grows as needed, so assigning layer 4 also creates the empty layers below it; an unassigned layer renders as white.

Every map of every layer is uploaded as one slice of a single texture array, so the whole terrain costs one sampler. Sources are converted to RGBA and resized to a common size, capped at 2048 px. The sampler filters and wrap modes come from the first assigned layer colour; a terrain with any PBR layer or custom tiling also gets mipmaps, which tiled surface maps need to stop aliasing at distance.

On a plain colour layer the texture's **alpha is read as layer height**, not opacity: where two layers overlap, the taller one takes the contact zone. Opaque textures have no height and blend on their blend-map weight alone. A PBR layer takes its height from its height map instead, leaving its colour alpha unused. Layers are also projected triplanarly as the surface turns vertical, and mixed with a coarser tiling rate at a distance to break up repetition.

C++

```
ground.setBlendMap("terrain/blendmap.png");
ground.setTextureLayer(0, "terrain/rock.png");
ground.setTextureLayer(1, "terrain/sand.png");
ground.setTextureLayer(2, "terrain/snow.png");
```

Lua

```
ground:setBlendMap("terrain/blendmap.png")
ground:setTextureLayer(0, "terrain/rock.png")
ground:setTextureLayer(1, "terrain/sand.png")
ground:setTextureLayer(2, "terrain/snow.png")
```

setSurfaceLayer / getSurfaceLayer

- void **setSurfaceLayer**(unsigned int index, const TerrainSurfaceLayer& layer)
- TerrainSurfaceLayer **getSurfaceLayer**(unsigned int index) const

Replaces or reads one whole layer, maps and factors together. `getSurfaceLayer` returns a **copy**, so editing it changes nothing until it is passed back to `setSurfaceLayer` — read, change, write.

C++

```
TerrainSurfaceLayer rock;
rock.pbr = true;
rock.colorTexture = Texture("terrain/rock_color.png");
rock.normalTexture = Texture("terrain/rock_normal.png");
rock.roughnessTexture = Texture("terrain/rock_orm.png");
rock.metallicTexture = Texture("terrain/rock_orm.png");
rock.roughnessFactor = 0.9f;
ground.setSurfaceLayer(0, rock);
```

Lua

```
local rock = TerrainSurfaceLayer()
rock.pbr = true
rock.colorTexture = Texture("terrain/rock_color.png")
rock.normalTexture = Texture("terrain/rock_normal.png")
rock.roughnessFactor = 0.9
ground:setSurfaceLayer(0, rock)

-- Read, change, write back
local layer = ground:getSurfaceLayer(0)
layer.uvScale = Vector2(2, 2)
ground:setSurfaceLayer(0, layer)
```

setLayerFromMaterial

• void **setLayerFromMaterial**(unsigned int index, const Material& material)

Fills a layer from a Material (p. 734) and turns on its PBR surface: base colour and factor, normal, metallic/roughness (assigned to both the roughness and metallic slots), occlusion, and the roughness and metallic factors.

Emission, alpha mode and secondary UV sets have no meaning on a terrain layer and are dropped. The layer keeps the UV scale and offset it already had, since a material carries no tiling of its own.

removeSurfaceLayer

• void **removeSurfaceLayer**(unsigned int index)

Removes a layer. Blend map channels are positional, so every later layer moves down one channel — onto the paint the removed layer left behind. Only removing the last layer is lossless, which is all the Terrain Editor offers.

getNumLayers

• unsigned int **getNumLayers**() const

How many layers the terrain currently holds, painted or empty.

TerrainSurfaceLayer

One painted surface. The maps are separate inputs so an existing material can fill them without repacking images; the renderer packs them into slices of the terrain's texture array.

Type	Name	Default	Meaning
bool	pbr	`false`	Off blends only the colour and leaves the rest to the terrain material
Texture (p. 738)	colorTexture	empty	Surface colour. sRGB-decoded and tinted on a PBR layer; used raw on a colour layer

Type	Name	Default	Meaning
Texture (p. 738)	normalTexture	empty	Tangent-space normal map, projected with the layer
Texture (p. 738)	roughnessTexture	empty	Read from green, or the only channel of a grayscale map
Texture (p. 738)	metallicTexture	empty	Read from blue, same fallback
Texture (p. 738)	occlusionTexture	empty	Read from red; white is unoccluded
Texture (p. 738)	heightTexture	empty	Biases blending toward this layer where it is taller; never moves geometry
Vector4	colorFactor	`(1,1,1,1)`	Linear tint multiplied with `colorTexture`. PBR layers only
float	normalStrength	`1.0`	How far the normal map tilts the surface
float	roughnessFactor	`1.0`	Multiplies `roughnessTexture`
float	metallicFactor	`0.0`	Multiplies `metallicTexture`
float	occlusionStrength	`1.0`	How far `occlusionTexture` darkens ambient light
Vector2	uvScale	`(1,1)`	Multiplies the shared detail tiling, for this layer alone
Vector2	uvOffset	`(0,0)`	Shifts this layer inside its tile

All fields are readable and writable from both C++ and Lua, and Lua can construct one with `TerrainSurfaceLayer()`.

`uvScale` and `uvOffset` apply whether or not `pbr` is on. Every other field beyond `colorTexture` needs `pbr`.

setTextureDetailRed / setTextureDetailGreen / setTextureDetailBlue

- void **setTextureDetailRed**(const std::string& path)
- void **setTextureDetailGreen**(const std::string& path)
- void **setTextureDetailBlue**(const std::string& path)

Layers 0, 1 and 2 under the names they had when a terrain had a single blend map. Equivalent to `setTextureLayer(0/1/2, path)`, so they set only the layer colour.

Tilemap

Inherits: Mesh (p. 436)
C++ type: `Tilemap`

Description

Renders a 2D tile grid as batched geometry. `Tilemap` works with two concepts:

- **Rect** — a named sub-region of a texture atlas (the source rectangle of a tile graphic).
- **Tile** — an instance placed in the world, referencing a `Rect` and specifying its position and size.

This two-level design lets you define each atlas frame once and place it any number of times with no additional GPU cost. All tiles sharing the same texture atlas are batched together, and larger maps are additionally split into spatial chunks that are frustum-culled (p. 258) individually, so tiles outside the camera view are never submitted.

A single tilemap can render at most **16 383** tiles — the index buffer is 16-bit and each tile is a four-vertex quad. Tiles beyond that limit are dropped with an error in the log; split larger worlds across several tilemaps or scenes.

Properties

Type	Name	Default	Langs
float	textureScaleFactor	`0.0`	C++ | Lua
unsigned int	reserveTiles	`10`	C++ | Lua

Methods

Type	Name	Langs
bool	createTilemap	C++
int	findRectByString	C++ | Lua
int	findTileByString	C++ | Lua
void	addRect	C++ | Lua
void	removeRect	C++ | Lua
void	clearRects	C++ | Lua
TileRectData (p. 719)	getRect	C++ | Lua
void	addTile	C++ | Lua
void	removeTile	C++ | Lua
void	clearTiles	C++ | Lua
TileData (p. 719)	getTile	C++ | Lua
unsigned int	getWidth	C++ | Lua
unsigned int	getHeight	C++ | Lua
void	clearAll	C++ | Lua
TilemapComponent (p. 695)	`getTilemapComponent`	Lua

Property details

textureScaleFactor

- *Setter*: void **setTextureScaleFactor**(float textureScaleFactor)
- *Getter*: float **getTextureScaleFactor**() const

Insets each tile's UV rectangle by this many texels on every side, `0.0` by default. A small value such as `0.5` keeps bilinear filtering from bleeding neighbouring tiles of the atlas into the edges of a tile. It is only exact while the texture keeps its size: the `RESIZE` texture strategy (p. 364) blurs the bleed further into the tiles.

reserveTiles

- *Setter*: void **setReserveTiles**(unsigned int reserveTiles)
- *Getter*: unsigned int **getReserveTiles**() const

Pre-allocates GPU buffer space for this many tiles. If you know the maximum tile count in advance, set this before the tilemap is first built to avoid mid-game reallocations. Changing it later rebuilds the tilemap.

Method details

createTilemap

- bool **createTilemap**()

Explicitly builds or rebuilds the tilemap's geometry and GPU buffers. Configure the texture, rects, and tiles first, then call this method if C++ code needs the geometry immediately. It is **not bound to Lua**; the mesh system builds and rebuilds tilemaps automatically during rendering. Adding, removing, or clearing rects and tiles, and setting `textureScaleFactor` or `reserveTiles`, marks the tilemap to be rebuilt on the next frame.

C++

```
Tilemap map(&scene);
map.setTexture("tiles.png");
map.addRect("grass", 0, 0, 32, 32);
map.addTile("grass", Vector2(64, 64), 32, 32);
map.createTilemap();
```

Lua

```
local map = Tilemap(scene)
map:setTexture("tiles.png")
map:addRect("grass", 0, 0, 32, 32)
map:addTile("grass", Vector2(64, 64), 32, 32)
```

findRectByString / findTileByString

- int **findRectByString**(const std::string& name)
- int **findTileByString**(const std::string& name)

Look up the integer ID of a rect or tile by name. Returns `-1` if not found.

addRect

Several overloads are available:

- void **addRect**(int id, const std::string& name, const std::string& texture, TextureFilter (p. 739) texFilter, Rect (p. 652) rect)
- void **addRect**(int id, const std::string& name, const std::string& texture, Rect (p. 652) rect)
- void **addRect**(int id, const std::string& name, Rect (p. 652) rect)
- void **addRect**(const std::string& name, float x, float y, float width, float height)
- void **addRect**(float x, float y, float width, float height)
- void **addRect**(Rect (p. 652) rect)

Defines a named atlas frame. The `texture` path specifies which file to sample from; when omitted, the tilemap uses the default texture set on the mesh. `rect` is the pixel region (x, y, width, height) within the atlas.

The `id` parameter is an explicit integer key; use the simpler overloads when you don't need manual IDs.

C++

```
map.addRect(0, "water",  "tiles.png", Rect(0,  0,  32, 32));
map.addRect(1, "ground", "tiles.png", Rect(32, 0,  32, 32));
map.addRect(2, "wall",   "tiles.png", Rect(64, 0,  32, 32));
```

Lua

```
map:addRect(0, "water",  "tiles.png", Rect(0,  0,  32, 32))
map:addRect(1, "ground", "tiles.png", Rect(32, 0,  32, 32))
map:addRect(2, "wall",   "tiles.png", Rect(64, 0,  32, 32))
```

removeRect

- void **removeRect**(int id)
- void **removeRect**(const std::string& name)

Removes an atlas-frame definition. Tiles that used the removed rect stop drawing. Rects with higher IDs move down by one, and the tiles that used them follow, so they keep their image.

clearRects

- void **clearRects**()

Removes all rect definitions. Tiles keep their rect IDs, so they stop drawing until rects with those IDs are added again.

getRect

- TileRectData (p. 719)& **getRect**(int id)
- TileRectData (p. 719)& **getRect**(const std::string& name)

Returns a reference to the TileRectData (p. 719) struct for the given rect. Editing the struct in place does not rebuild the tilemap. To change a frame at runtime, call addRect again with the same ID.

addTile

Several overloads are available:

- void **addTile**(int id, const std::string& name, int rectId, Vector2 position, float width, float height)
- void **addTile**(const std::string& name, int rectId, Vector2 position, float width, float height)
- void **addTile**(int rectId, Vector2 position, float width, float height)
- void **addTile**(const std::string& name, const std::string& rectString, Vector2 position, float width, float height)
- void **addTile**(const std::string& rectString, Vector2 position, float width, float height)

Places a tile instance in world space. `rectId` (or `rectString`) identifies which atlas frame to use; `position` is the 2D world-space centre; `width` and `height` are the tile's display dimensions.

C++

```cpp
// Build a simple 3x3 ground patch
for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
        map.addTile("ground", Vector2(x * 32.0f, y * 32.0f), 32.0f, 32.0f);
    }
}
```

Lua

```lua
for x = 0, 2 do
    for y = 0, 2 do
        map:addTile("ground", Vector2(x * 32, y * 32), 32, 32)
    end
end
```

removeTile

- void **removeTile**(int id)
- void **removeTile**(const std::string& name)

Removes a specific tile instance.

clearTiles

- void **clearTiles**()

Removes all tile instances. The rect definitions are preserved.

getTile

- TileData (p. 719)& **getTile**(int id)
- TileData (p. 719)& **getTile**(const std::string& name)

Returns a reference to the TileData (p. 719) struct for the given tile. Editing the struct in place does not rebuild the tilemap. To move, resize, or remap a tile at runtime, call addTile again with the same ID.

getWidth / getHeight

- unsigned int **getWidth**()
- unsigned int **getHeight**()

Returns the overall pixel dimensions of the tilemap bounding box (the extent covered by all placed tiles).

clearAll

- void **clearAll**()

Removes all tiles and all rect definitions in one call.

Water

Inherits: Object (p. 455)
C++ type: `Water`

Description

A water surface: a flat grid at the entity's height, displaced by Gerstner waves and shaded with scrolling ripples, sky or planar reflections, refraction of the scene behind it, shore and crest foam, and a fade seen from below the surface. Constructing a `Water` attaches a `WaterComponent` to the entity. Water has its own render path, so it is not a Mesh (p. 436) and needs no material, texture, or camera wiring.

The waves are computed in world space, so moving, rotating, or scaling the entity changes the area the water covers but never stretches the waves. The surface always stays level at the entity's world height, even when the entity is tilted.

See Rendering Pipeline — Water (p. 270) for how the surface is drawn and what each feature costs.

C++

```
Water water(&scene);
water.setPosition(0.0f, -0.5f, 0.0f);
water.setSize(80.0f, 80.0f);
water.setWaveHeight(0.4f);
water.setWaveDirection(30.0f);
water.setPlanarReflection(true);
```

Lua

```
local water = Water(scene)
water:setPosition(0, -0.5, 0)
water:setSize(80, 80)
water.waveHeight = 0.4
water.waveDirection = 30
water.planarReflection = true
```

A dynamic Body3D (p. 554) with a **buoyancy** above `0` floats on the waves without any script, while the water's own buoyancy is on; see Physics — Buoyancy (p. 319). A WaterExclusion (p. 503) keeps the water out of a boat, a cabin, or a cave.

Properties

Type	Name	Default	Langs
Vector2 (p. 656)	size	`Vector2(50, 50)`	C++ | Lua
unsigned int	subdivisions	`128`	C++ | Lua
Vector3 (p. 660)	shallowColor	`Vector3(0.20, 0.62, 0.62)`	C++ | Lua
Vector3 (p. 660)	deepColor	`Vector3(0.02, 0.13, 0.22)`	C++ | Lua
float	depthFade	`3.0`	C++ | Lua
float	waveHeight	`0.2`	C++ | Lua
float	waveLength	`6.0`	C++ | Lua
float	waveSpeed	`1.0`	C++ | Lua
float	waveDirection	`0.0`	C++ | Lua
float	waveSteepness	`0.6`	C++ | Lua
float	normalScale	`4.0`	C++ | Lua
float	normalStrength	`0.4`	C++ | Lua
float	rippleSpeed	`0.3`	C++ | Lua

Type	Name	Default	Langs
float	reflectivity	`1.0`	C++ | Lua
float	specularIntensity	`1.0`	C++ | Lua
float	roughness	`0.06`	C++ | Lua
bool	receiveShadows	`true`	C++ | Lua
bool	planarReflection	`false`	C++ | Lua
float	reflectionDistortion	`0.04`	C++ | Lua
bool	refraction	`true`	C++ | Lua
float	refractionDistortion	`0.03`	C++ | Lua
float	caustics	`0.5`	C++ | Lua
float	causticsScale	`2.0`	C++ | Lua
bool	underwater	`true`	C++ | Lua
bool	buoyancy	`true`	C++ | Lua
float	buoyancyDepth	`0.0`	C++ | Lua
float	wakeArea	`64.0`	C++ | Lua
float	wakeFade	`8.0`	C++ | Lua
bool	depthEffects	`true`	C++ | Lua
Vector3 (p. 660)	foamColor	`Vector3(1, 1, 1)`	C++ | Lua
float	shoreFoam	`0.4`	C++ | Lua
float	crestFoam	`0.15`	C++ | Lua
string	customShader	`""`	C++ | Lua
string	customUnderwaterShader	`""`	C++ | Lua

Methods

Type	Name	Langs
bool	load	C++ | Lua
void	setSize	C++ | Lua
void	setShallowColor	C++ | Lua
void	setDeepColor	C++ | Lua
void	setFoamColor	C++ | Lua
void	setNormalTexture	C++ | Lua
void	setShaderUniform	C++ | Lua
Vector4	getShaderUniform	C++ | Lua
bool	removeShaderUniform	C++ | Lua
float	getHeight	C++ | Lua
Vector3	getNormal	C++ | Lua
WaterComponent (p. 701)	`getWaterComponent`	Lua

Property details

size / subdivisions

- *Setter*: void **setSize**(Vector2 (p. 656) size)
- *Setter*: void **setSize**(const float width, const float depth)
- *Getter*: Vector2 (p. 656) **getSize**() const

- *Setter*: void **setSubdivisions**(unsigned int subdivisions)
- *Getter*: unsigned int **getSubdivisions**() const

`size` is the width (X) and depth (Z) of the surface, centered on the entity and scaled by its transform. `subdivisions` is the number of grid cells along each side, capped at `512`. Short waves need more cells to keep their shape; the grid has `(subdivisions + 1)²` vertices, so raise it only as far as the waves need. Changing either rebuilds the grid.

shallowColor / deepColor / depthFade

- *Setter*: void **setShallowColor**(Vector3 color)
- *Setter*: void **setShallowColor**(const float r, const float g, const float b)
- *Getter*: Vector3 **getShallowColor**() const
- *Setter*: void **setDeepColor**(Vector3 color)
- *Setter*: void **setDeepColor**(const float r, const float g, const float b)
- *Getter*: Vector3 **getDeepColor**() const
- *Setter*: void **setDepthFade**(float depthFade)
- *Getter*: float **getDepthFade**() const

The water colour, in sRGB (stored linear, so the value read back is the sRGB form of the stored one). Shallow water takes `shallowColor` and turns to `deepColor` as it deepens; `depthFade` is the depth in world units where the deep colour takes over. Shallower water is lighter and clearer.

With refraction or the underwater view, `shallowColor` is also the tint of the scene seen through the water: its clearest channel fades out over `depthFade`, and the other channels fade faster, while `deepColor` fills in the light scattered by the water.

waves

- *Setter*: void **setWaveHeight**(float waveHeight)
- *Setter*: void **setWaveLength**(float waveLength)
- *Setter*: void **setWaveSpeed**(float waveSpeed)
- *Setter*: void **setWaveDirection**(float waveDirection)
- *Setter*: void **setWaveSteepness**(float waveSteepness)
- *Getters*: **getWaveHeight**(), **getWaveLength**(), **getWaveSpeed**(), **getWaveDirection**(), **getWaveSteepness**()

The surface is the sum of four Gerstner waves derived from these values:

Property	Meaning
`waveHeight`	How far the crests rise above the water level, in world units.
`waveLength`	Length of the longest wave. The other three are shorter and turned a little away from it. Longer waves also travel faster, as on deep water.
`waveSpeed`	Multiplier on the wave travel speed. `0` freezes the waves; a negative value runs them backwards.
`waveDirection`	Angle around Y the waves travel toward, `0` = `+X`. Degrees, or radians when Engine::useDegrees (p. 365) is off.
`waveSteepness`	`0` gives round waves, `1` sharp crests. It is lowered automatically where the waves would fold over.

Wave changes take effect on the next frame without a reload.

ripples

- void **setNormalTexture**(const std::string& path)

- *Setter*: void **setNormalScale**(float normalScale)
- *Setter*: void **setNormalStrength**(float normalStrength)
- *Setter*: void **setRippleSpeed**(float rippleSpeed)
- *Getters*: **getNormalScale**(), **getNormalStrength**(), **getRippleSpeed**()

Small ripples come from a tiling normal map, sampled twice in two layers that drift across each other. An empty path restores the built-in ripple map. Give a custom map a mipmap filter so distant ripples do not shimmer.

`normalScale` is the world size of one tile, `normalStrength` how much the ripples bend the lighting and reflections, and `rippleSpeed` how fast the layers drift (world units per second).

reflectivity / specularIntensity / roughness

- *Setter*: void **setReflectivity**(float reflectivity)
- *Setter*: void **setSpecularIntensity**(float specularIntensity)
- *Setter*: void **setRoughness**(float roughness)
- *Getters*: **getReflectivity**(), **getSpecularIntensity**(), **getRoughness**()

`reflectivity` scales the reflection (sky, or the scene with planarReflection), `specularIntensity` scales the sun glints, and `roughness` (`0`–`1`) spreads the highlight: low values give sharp glints.

Without planar reflection the water reflects the scene's Sky (p. 472) (its texture, colour, and rotation), or the background colour when the scene has no sky, and takes its ambient light from the sky's environment when IBL is available.

receiveShadows

- *Setter*: void **setReceiveShadows**(bool receiveShadows)
- *Getter*: bool **isReceiveShadows**() const

Shadows of the scene lights darken the sun glints and the light under the surface. Takes effect only when the scene has a shadow-casting light. Deep, clear water shows shadows only faintly, because most of its colour is reflected sky.

planarReflection / reflectionDistortion

- *Setter*: void **setPlanarReflection**(bool planarReflection)
- *Getter*: bool **isPlanarReflection**() const
- *Setter*: void **setReflectionDistortion**(float reflectionDistortion)
- *Getter*: float **getReflectionDistortion**() const

Reflects the scene, not only the sky, like a Mirror (p. 448): the scene is drawn a second time, at half resolution, from a camera reflected across the water plane. `reflectionDistortion` is how much the ripples bend the reflection.

Only the scene's main camera sees the planar reflection; render-to-texture cameras, mirrors, and other waters' reflection passes see the water reflect the sky.

refraction / refractionDistortion

- *Setter*: void **setRefraction**(bool refraction)
- *Getter*: bool **isRefraction**() const
- *Setter*: void **setRefractionDistortion**(float refractionDistortion)
- *Getter*: float **getRefractionDistortion**() const

Shows the scene behind the water, bent by the ripples and tinted by the water it crosses (see shallowColor). The engine copies the scene once per frame for it. `refractionDistortion` is how much the ripples bend what is seen through.

Refraction is drawn for the main camera. Other cameras show the water alpha-blended over the scene instead, as does a water with refraction off.

caustics / causticsScale

- *Setter*: void **setCaustics**(float caustics)
- *Getter*: float **getCaustics**() const
- *Setter*: void **setCausticsScale**(float causticsScale)
- *Getter*: float **getCausticsScale**() const

Brightness of the sunlight patterns on the floor seen through the water; `0` turns them off. `causticsScale` is the world size of one cell of the pattern. They need refraction and depthEffects, are brightest in shallow water, and scale with the directional lights reaching the surface.

Lua

```lua
local water = Water(self.scene, self.entity)
water.caustics = 0.9
water.causticsScale = 1.6
```

underwater

- *Setter*: void **setUnderwater**(bool underwater)
- *Getter*: bool **isUnderwater**() const

When the main camera goes below the surface, inside the water's area, the scene fades into the water with distance, using the same colours and depth fade, unless the camera is in a WaterExclusion (p. 503). With depthEffects off the fade is even, as if everything were `depthFade` away.

buoyancy / buoyancyDepth

- *Setter*: void **setBuoyancy**(bool buoyancy)
- *Getter*: bool **isBuoyancy**() const
- *Setter*: void **setBuoyancyDepth**(float buoyancyDepth)
- *Getter*: float **getBuoyancyDepth**() const

`buoyancy` lets the water float the dynamic bodies whose Body3D buoyancy (p. 554) is above `0`; turn it off for water that is only for show. `buoyancyDepth` gives the water a bottom: a body whose centre of mass goes deeper than that below the surface no longer floats in it. `0` has no bottom.

Where waters overlap, a body floats in the one with the highest bottom that it is still above, so a pool with a `buoyancyDepth` over a lake floats only the bodies inside it and the lake floats the rest. The bottom is not a floor: a body that plunges below it sinks through, so give a pool a collider too.

Lua

```lua
local pool = Water(self.scene, self.entity)
pool.buoyancyDepth = 1.5
```

wakeArea / wakeFade

- *Setter*: void **setWakeArea**(float wakeArea)
- *Getter*: float **getWakeArea**() const

- *Setter*: void **setWakeFade**(float wakeFade)
- *Getter*: float **getWakeFade**() const

The wakes of the Water Exclusions (p. 504) moving across the water are kept in a square around what the main camera looks at, `wakeArea` world units on a side, in 256 × 256 texels: the default 64 resolves a quarter of a unit, enough for the foam along a small boat. A larger area reaches farther with coarser foam. The square moves with the camera without moving what it holds; a wake that leaves it is dropped. `wakeFade` is the seconds a wake takes to fade, foam and waves alike.

The map exists only while a wake is being made or fading, and costs one texture update a frame then. Stopping Play in the editor clears it.

depthEffects

- *Setter*: void **setDepthEffects**(bool depthEffects)
- *Getter*: bool **isDepthEffects**() const

Uses the scene depth for shore foam, soft edges where the water meets geometry, and the shallow-to-deep tint. The depth comes from SSR or SSAO when the scene already has one of them on; otherwise the engine adds a depth pre-pass for the water. Only the main camera has the scene depth.

foamColor / shoreFoam / crestFoam

- *Setter*: void **setFoamColor**(Vector3 color)
- *Setter*: void **setFoamColor**(const float r, const float g, const float b)
- *Getter*: Vector3 **getFoamColor**() const
- *Setter*: void **setShoreFoam**(float shoreFoam)
- *Setter*: void **setCrestFoam**(float crestFoam)
- *Getters*: **getShoreFoam**(), **getCrestFoam**()

`foamColor` is in sRGB. `shoreFoam` is the water depth the shore foam covers (needs depthEffects); `crestFoam` (`0`–`1`) is how much of the surface breaks into whitecaps, `0` for none. See Rendering Pipeline — Foam (p. 271) for how the foam is drawn.

customShader / customUnderwaterShader

- *Setter*: void **setCustomShader**(const std::string& path)
- *Getter*: std::string **getCustomShader**() const
- *Setter*: void **setCustomUnderwaterShader**(const std::string& path)
- *Getter*: std::string **getCustomUnderwaterShader**() const

Base paths (no extension) of forked shaders. `customShader` replaces the water surface shader (`water.vert` / `water.frag`); `customUnderwaterShader` replaces the fullscreen fade of the underwater view (`fullscreen.vert` / `underwater.frag`). An empty path uses the scene's default water shader (p. 401) for `customShader`, then the built-in. See Custom Shaders — Water shaders (p. 145).

Method details

load

- bool **load**()

Builds the grid and loads the shaders now instead of on the next frame. Not needed in normal use: the render system loads a water the first time it updates.

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`
- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the forks declare with `#pragma param`, by name. The water fork and the underwater fork read the same values. Setting a value only rewrites the uploaded block, so it can run every frame. Same semantics as Mesh.setShaderUniform (p. 445); see Custom Shaders — Shader parameters (p. 147).

getHeight / getNormal

- float **getHeight**(float x, float z) const
- Vector3 **getNormal**(float x, float z) const

The world height and the unit normal of the surface over the world point `(x, z)`, waves included. The result matches the surface the next frame draws, so calling it in `onUpdate` keeps a buoy, a boat without physics, or a camera on the waves:

C++

```
Vector3 pos = buoy.getPosition();
pos.y = water.getHeight(pos.x, pos.z);
buoy.setPosition(pos);
```

Lua

```
local pos = buoy.position
pos.y = water:getHeight(pos.x, pos.z)
buoy:setPosition(pos)
```

Neither call checks the water's area: a point outside it gets the waves as if the water went on. Both follow the built-in wave math, so a custom shader that moves the vertices differently no longer matches them.

description: WaterExclusion API reference (C++ and Lua) — a volume that keeps water out, like a boat: no water surface, underwater fade, or buoyancy inside it.

WaterExclusion

Inherits: Object (p. 455)
C++ type: `WaterExclusion`

Description

A volume that keeps Water (p. 496) out, like the inside of a boat. Inside it the water surface is not drawn, a camera sees no underwater (p. 500) fade, and dynamic bodies get no buoyancy (p. 554), so cargo rests on the floor of a boat instead of floating on the sea outside. Constructing a `WaterExclusion` attaches a `WaterExclusionComponent` to the entity, and the volume moves, rotates, and scales with it.

The default **HULL** shape wraps the convex hull of the entity's meshes, or of its model's mesh nodes, and is rebuilt when they change. On a boat it is the outer hull closed over the top at the gunwale, so the component goes on the boat itself. **BOX** and **SPHERE** fill a volume of their own, for a cabin, a cave, or a diving bell.

C++

```cpp
// keeps the water out of a boat
boat.addComponent<WaterExclusionComponent>();

// a dry room below the sea
WaterExclusion room(&scene);
room.setShape(WaterExclusionShape::BOX);
room.setPosition(0.0f, -3.0f, 0.0f);
room.setSize(6.0f, 3.0f, 4.0f);
```

Lua

```lua
-- a dry room below the sea
local room = WaterExclusion(scene)
room.shape = WaterExclusionShape.BOX
room:setPosition(0, -3, 0)
room:setSize(6, 3, 4)
```

A body that owns the volume, on its own entity or on a child of it, still floats: the boat keeps its buoyancy while what it carries loses it. See Rendering Pipeline — Water exclusions (p. 272) for how the surface is cut.

Hidden, on its own entity or through an ancestor, the volume no longer cuts the surface, stops the underwater fade or leaves a wake, and its wake starts afresh when it shows again. Bodies inside it still lose their buoyancy, as hiding leaves physics running.

Properties

Type	Name	Default	Langs
WaterExclusionShape	shape	`HULL`	C++ | Lua
Vector3 (p. 660)	center	`Vector3(0, 0, 0)`	C++ | Lua
Vector3 (p. 660)	size	`Vector3(1, 1, 1)`	C++ | Lua
float	wakeFoam	`0.0`	C++ | Lua
float	wakeWaves	`0.0`	C++ | Lua

Methods

Type	Name	Langs
void	setCenter	C++ | Lua
void	setSize	C++ | Lua
bool	contains	C++ | Lua
WaterExclusionComponent (p. 702)	`getWaterExclusionComponent`	Lua

Enumerations

WaterExclusionShape

- **BOX** — A box of size around center.
- **SPHERE** — An ellipsoid of size around center.
- **HULL** — The convex hull of the entity's meshes.

Property details

shape

- *Setter*: void **setShape**(WaterExclusionShape shape)
- *Getter*: WaterExclusionShape **getShape**() const

A hull is convex, so a boat with a dip in its outline, like a catamaran, takes one volume per hull, each on its own mesh. Building a hull needs 3D physics, which projects have unless they turn `DORIAX_PHYSICS_3D` off (see Build Options (p. 357)).

center / size

- *Setter*: void **setCenter**(Vector3 (p. 660) center)
- *Setter*: void **setCenter**(const float x, const float y, const float z)
- *Getter*: Vector3 (p. 660) **getCenter**() const
- *Setter*: void **setSize**(Vector3 (p. 660) size)
- *Setter*: void **setSize**(const float x, const float y, const float z)
- *Getter*: Vector3 (p. 660) **getSize**() const

The box or sphere in the entity's local space: `size` is its width, height, and depth, centred on `center`. A hull ignores them.

wakeFoam / wakeWaves

- *Setter*: void **setWakeFoam**(float wakeFoam)
- *Getter*: float **getWakeFoam**() const
- *Setter*: void **setWakeWaves**(float wakeWaves)
- *Getter*: float **getWakeWaves**() const

What the volume leaves on a Water (p. 496) while it moves across its surface, from `0` (nothing, the default) to `2`. The wake follows the motion of the entity, whatever moves it — physics, a script or an animation — so a boat only needs these set:

- **Foam** stays where it forms and fades there: churned behind the stern and, closer to hull speed, along the sides and at the bow, where the bow waves of a fast hull break.
- **Waves** spread from where the bow passed, at the Kelvin angle of 19.5° on each side, narrowing past a length Froude number of about 0.5, with the transverse waves crossing between them. They bend the surface, so they show in its light and reflections.

Both grow with the speed relative to the hull speed of the volume's length, so a small boat drifting leaves almost nothing and one near its hull speed a full wake. The water keeps the wakes in a square around the camera and fades them over its wakeFade (p. 500). Waves stay on the water they were made on, even after the volume leaves it. The waterline is taken as an ellipse inside the volume's bounds, facing the motion.

C++

```
WaterExclusion hull(&scene, boat);
hull.setWakeFoam(1.0f);
hull.setWakeWaves(1.0f);
```

Lua

```
local hull = WaterExclusion(self.scene, self.entity)
hull.wakeFoam = 1
hull.wakeWaves = 1
```

The wake is drawn by the built-in water shader; a fork of an older version draws none until it is forked again.

Method details

contains

- bool **contains**(Vector3 (p. 660) point) const

Whether the world point is inside the volume, tested as buoyancy tests it. Gameplay that should agree with the water, like the air of a diver, can use it:

Lua

```
local cabin = WaterExclusion(self.scene, self.entity)
if cabin:contains(diver:getWorldPosition()) then
    oxygen = maxOxygen -- breathing in the dry cabin
end
```

The volume is placed by the render update, so a new one contains nothing until the next frame.

UI

Button

Inherits: Image (p. 513)
C++ type: `Button`

Description

An interactive UI button widget. `Button` extends Image (p. 513) with four visual states — *normal*, *hovered*, *pressed*, and *disabled* — each of which can have a distinct texture or tint colour. An optional text label can be embedded inside the button.

Respond to a click with the `onClick` event of its UIComponent (p. 698), which fires only when the pointer is released over the button. `onPress` / `onRelease` follow the press itself, for actions held while the button is down.

Buttons can also be focused and pressed with the keyboard or a gamepad. See Keyboard and gamepad navigation (p. 295).

Properties

Type	Name	Default	Langs
std::string	label	`""`	C++ | Lua
Vector4	labelColor	`(1,1,1,1)`	C++ | Lua
std::string	labelFont	system font	C++ | Lua
unsigned int	labelFontSize	`16`	C++ | Lua
Vector4	colorNormal	`(1,1,1,1)`	C++ | Lua
Vector4	colorHovered	`(1,1,1,1)`	C++ | Lua
Vector4	colorPressed	`(1,1,1,1)`	C++ | Lua
Vector4	colorDisabled	`(1,1,1,1)`	C++ | Lua
float	scaleHovered	`1.0`	C++ | Lua
float	scalePressed	`1.0`	C++ | Lua
float	transitionTime	`0.0`	C++ | Lua
bool	disabled	`false`	C++ | Lua
bool	focusable	`true`	C++ | Lua

Methods

Type	Name	Langs
bool	hasLabel	C++
Text (p. 528)	getLabelObject	C++ | Lua
void	setTextureNormal	C++ | Lua
void	setTextureHovered	C++ | Lua
void	setTexturePressed	C++ | Lua
void	setTextureDisabled	C++ | Lua
void	setFocus	C++ | Lua
bool	isFocused	C++ | Lua

Callback events

Callback	Name	Langs
void()	onPress	C++ | Lua
void()	onRelease	C++ | Lua

All pointer events from Image (p. 513) are also available.

Property details

label

- *Setter*: void **setLabel**(const std::string& text)
- *Getter*: std::string **getLabel**() const
- *Setter*: void **setLabelColor**(Vector4 color)
- *Setter*: void **setLabelFont**(const std::string& font)
- *Setter*: void **setLabelFontSize**(unsigned int fontSize)

The text displayed inside the button. Setting a non-empty label automatically creates a child Text (p. 528) entity. Font and colour apply to that label text. The button's 9-patch margins do not clip the label, but a parent that clips the button still clips it.

stateColors

- *Setter/Getter*: void **setColorNormal** / **getColorNormal**
- *Setter/Getter*: void **setColorHovered** / **getColorHovered**
- *Setter/Getter*: void **setColorPressed** / **getColorPressed**
- *Setter/Getter*: void **setColorDisabled** / **getColorDisabled**

Per-state tint colours multiplied with the current state texture. Use these to create hover and press effects with a single texture. The current state colour replaces the colour of the button's UIComponent (p. 698), so change `colorNormal` rather than the image colour.

C++

```
Button btn(&scene);
btn.createImage();
btn.setTexture("ui/button.png");
btn.setPatchMargin(12);
btn.setSize(160, 48);
btn.setLabel("Play");
btn.setColorNormal(Vector4(1.0f, 1.0f, 1.0f, 1.0f));
btn.setColorHovered(Vector4(0.85f, 0.85f, 1.0f, 1.0f));
btn.setColorPressed(Vector4(0.7f, 0.7f, 1.0f, 1.0f));
```

Lua

```
local btn = Button(scene)
btn:setTexture("ui/button.png")
btn:setPatchMargin(12)
btn:setSize(160, 48)
btn.label = "Play"
btn.colorNormal = Vector4(1.0, 1.0, 1.0, 1.0)
btn.colorHovered = Vector4(0.85, 0.85, 1.0, 1.0)
btn.colorPressed = Vector4(0.7, 0.7, 1.0, 1.0)
```

transitions

- *Setter/Getter*: void **setScaleHovered** / float **getScaleHovered**

- *Setter/Getter*: void **setScalePressed** / float **getScalePressed**
- *Setter/Getter*: void **setTransitionTime** / float **getTransitionTime**

A hovered or pressed button grows by `scaleHovered` or `scalePressed` (`1.1` is 10% bigger), around its pivot (p. 539), so set the pivot to `(0.5, 0.5)` to grow it from the middle. With a `transitionTime` in seconds, the state colour and scale change over that time when the pointer enters, leaves, presses or releases the button, instead of at once. Textures still swap at once.

C++

```
btn.setScaleHovered(1.08f);
btn.setScalePressed(0.95f);
btn.setTransitionTime(0.12f);
```

Lua

```
btn.scaleHovered = 1.08
btn.scalePressed = 0.95
btn.transitionTime = 0.12
```

disabled

- *Setter*: void **setDisabled**(bool disabled)
- *Getter*: bool **getDisabled**() const

When `true`, the button ignores pointer events and displays with `colorDisabled` / `textureDisabled`. Use to grey out unavailable actions.

focusable

- *Setter*: void **setFocusable**(bool focusable)
- *Getter*: bool **isFocusable**() const

When `false`, keys and gamepads leave this button alone: the arrow keys, the d-pad, and the left stick skip it, and Enter, Space, or the A button never press it, so it can only be clicked. Turn it off on a button the focus shouldn't reach, like a HUD button in the same scene as a menu. Hidden and disabled buttons are always skipped.

statetextures

- void **setTextureNormal**(const std::string& path)
- void **setTextureHovered**(const std::string& path)
- void **setTexturePressed**(const std::string& path)
- void **setTextureDisabled**(const std::string& path)

Assigns a unique texture for each interaction state. If not set, all states use the base texture from Image (p. 513) tinted by the state colour. Whenever the button loads or changes state, the state texture replaces the image texture, so change the button's look with `setTextureNormal`, which sets both, rather than `setTexture`.

Method details

hasLabel

- bool **hasLabel**() const

Returns `true` if a label child entity has been created for this button.

getLabelObject

• Text (p. 528) **getLabelObject**() const

Returns a handle to the child Text (p. 528) entity. Useful for advanced label manipulation such as alignment or size.

setFocus

• void **setFocus**()

Moves the keyboard and gamepad focus to this button and shows it, so Enter, Space, or the A button presses it. Call it when a menu opens to start on its main button: keys and gamepads stay with gameplay until a button has the focus. A button with focusable turned off takes the focus but doesn't show it or react to those keys.

C++

```
playButton->setFocus();
```

Lua

```
self.playButton:setFocus()
```

isFocused

• bool **isFocused**() const

Returns `true` while this button has the focus, even when the mouse has hidden it. A click also gives a button the focus, but keys and gamepads only use it once a script has called setFocus.

Callback event details

onPress / onRelease

• *Property*: FunctionSubscribe<void()> **onPress**

• *Property*: FunctionSubscribe<void()> **onRelease**

`onPress` fires when the mouse button is pressed over the button element, or when Enter, Space, or the gamepad A button is pressed while the button has the focus. `onRelease` fires when that press ends, even if the pointer has left the button. For a click, use the `onClick` event of the button's UIComponent (p. 698).

C++

```
btn.getComponent<ButtonComponent>().onPress.add("pressed", [&](){
    Log::print("Button pressed!");
});
```

Lua

```
local btnComp = btn:getButtonComponent()
btnComp.onPress = function()
    Log.print("Button pressed!")
end
```

Container

Inherits: UILayout (p. 537)
C++ type: `Container`

Description

An invisible layout container that automatically arranges child UI elements in a vertical or horizontal stack, with optional wrapping. Add UI children as child entities of the container's Object (p. 455); the container measures and positions them each frame.

A container lays out up to 30 visible children (`MAX_CONTAINER_BOXES`). A child past that loses its parent and an error is logged.

Properties

Type	Name	Default	Langs
ContainerType	type	`VERTICAL`	C++ | Lua
bool	useAllWrapSpace	`false`	C++ | Lua
unsigned int	wrapCellWidth	`0`	C++ | Lua
unsigned int	wrapCellHeight	`0`	C++ | Lua

Methods

Type	Name	Langs
void	resize	C++ | Lua
unsigned int	getContentWidth	C++
unsigned int	getContentHeight	C++
void	setBoxExpand	C++ | Lua
UIContainerComponent (p. 699)	`getUIContainerComponent`	Lua

Enumerations

ContainerType

- **VERTICAL** — Children stacked top to bottom.
- **HORIZONTAL** — Children placed left to right.
- **VERTICAL_WRAP** — Children stacked vertically; new columns start when height is exhausted.
- **HORIZONTAL_WRAP** — Children placed horizontally; new rows start when width is exhausted.

Property details

type

- *Setter*: void **setType**(ContainerType type)
- *Getter*: ContainerType **getType**() const

Layout direction.

C++

```
Container menu(&scene);
menu.setAnchorPreset(AnchorPreset::CENTER);
menu.setType(ContainerType::VERTICAL);

Button btnPlay(&scene);
btnPlay.setSize(200, 48);
menu.addChild(&btnPlay);

Button btnQuit(&scene);
btnQuit.setSize(200, 48);
menu.addChild(&btnQuit);
```

Lua

```
local menu = Container(scene)
menu.anchorPreset = AnchorPreset.CENTER
menu.type = ContainerType.VERTICAL

local btnPlay = Button(scene)
btnPlay:setSize(200, 48)
menu:addChild(btnPlay)

local btnQuit = Button(scene)
btnQuit:setSize(200, 48)
menu:addChild(btnQuit)
```

useAllWrapSpace

- *Setter*: void **setUseAllWrapSpace**(bool useAllWrapSpace)
- *Getter*: bool **isUseAllWrapSpace**() const

When `true` and the layout is a wrap type, the cells are distributed evenly to fill the available space.

wrapCellWidth / wrapCellHeight

- *Setters/Getters*: **setWrapCellWidth** / **getWrapCellWidth**, **setWrapCellHeight** / **getWrapCellHeight**

Fixed cell size used when the layout type is `VERTICAL_WRAP` or `HORIZONTAL_WRAP`.

Method details

resize

- void **resize**()

Forces an immediate layout recalculation. Normally the container updates automatically each frame, but you can call this after batch-adding children to avoid a one-frame lag.

getContentWidth / getContentHeight

- unsigned int **getContentWidth**() const
- unsigned int **getContentHeight**() const

Returns the total size of all laid-out children. Use this to size a Scrollbar (p. 525) relative to the visible area.

setBoxExpand

- void **setBoxExpand**(bool expand) — applies to all children

- void **setBoxExpand**(size_t id, bool expand)
- bool **isBoxExpand**(size_t id) const

Controls whether a specific child slot expands to fill available space. When `expand` is `true`, the slot grows to fill the remaining parent area not occupied by fixed-size children.

Image

Inherits: UILayout (p. 537)
C++ type: `Image`

Description

The foundational 2D UI visual element. `Image` renders a textured or colour-filled rectangle with optional 9-patch scaling. It is the base class for Button (p. 506), Panel (p. 518), Progressbar (p. 523), Scrollbar (p. 525), and TextEdit (p. 533).

9-patch scaling (also called *scale-9* or *nine-slice*) divides the image into a 3×3 grid. The four corner pieces are never stretched, while the edges and centre fill the available space. This allows buttons and panel backgrounds to scale to any size without distorting the corners.

All UI elements also expose a rich set of pointer/focus callback events (inherited via `UIComponent`). See the Callback events section.

Properties

Type	Name	Default	Langs
Vector4	color	`(1,1,1,1)`	C++ | Lua
float	alpha	`1.0`	C++ | Lua
bool	flipY	`false`	C++ | Lua
unsigned int	patchMarginLeft	`0`	C++ | Lua
unsigned int	patchMarginRight	`0`	C++ | Lua
unsigned int	patchMarginTop	`0`	C++ | Lua
unsigned int	patchMarginBottom	`0`	C++ | Lua
float	textureScaleFactor	`0.0`	C++ | Lua
string	customShader	`""`	C++ | Lua

Methods

Type	Name	Langs
bool	createImage	C++
void	setTexture	C++ | Lua
void	setPatchMargin	C++ | Lua
void	setColor	C++ | Lua
void	setAlpha	C++
AABB	getAABB	C++ | Lua
AABB	getWorldAABB	C++ | Lua
void	setShaderUniform	C++ | Lua
Vector4	getShaderUniform	C++ | Lua
bool	removeShaderUniform	C++ | Lua
ImageComponent (p. 672)	`getImageComponent`	Lua

Callback events

Callback	Name	Langs
void()	onGetFocus	C++ | Lua
void()	onLostFocus	C++ | Lua

Callback	Name	Langs
void(float,float)	onPointerEnter	C++ | Lua
void(float,float)	onPointerLeave	C++ | Lua
void(float,float)	onPointerMove	C++ | Lua
void(float,float)	onPointerDown	C++ | Lua
void(float,float)	onPointerUp	C++ | Lua
void(float,float)	onClick	C++ | Lua
void(float,float)	onDoubleClick	C++ | Lua
void(float,float)	onDragStart	C++ | Lua
void(float,float)	onDrag	C++ | Lua
void(float,float)	onDragEnd	C++ | Lua

Property details

color

- *Setter*: void **setColor**(Vector4 color)
- *Setter*: void **setColor**(float red, float green, float blue, float alpha)
- *Setter*: void **setColor**(float red, float green, float blue)
- *Setter*: void **setAlpha**(float alpha)
- *Getter*: Vector4 **getColor**() const
- *Getter*: float **getAlpha**() const

The tint colour multiplied with the texture, in sRGB. `(1,1,1,1)` renders the texture as-is. Changing the colour is a cheap way to create hover/pressed states without separate textures.

It is stored linear internally, so the value read back is the sRGB form of the stored one. Writing `UIComponent.color` (p. 699) instead skips the conversion and takes the linear value directly — which is also what a colour authored in a `.scene` file is.

---

flipY

- *Setter*: void **setFlipY**(bool flipY)
- *Getter*: bool **isFlipY**() const

Flips the texture vertically. Useful when a texture is loaded upside-down or for certain rendering passes.

---

patchMargin

- *Setter*: void **setPatchMargin**(unsigned int margin)
- *Setter*: void **setPatchMargin**(unsigned int left, unsigned int right, unsigned int top, unsigned int bottom)
- Individual getters: **getPatchMarginLeft**, **getPatchMarginRight**, **getPatchMarginTop**, **getPatchMarginBottom**

Sets the 9-patch insets. When all four margins are zero (the default), the image is scaled uniformly. Setting the margins enables 9-patch mode where the corners are preserved.

An image clips its child elements to its rectangle inside these margins, except on a Button (p. 506). A child that should draw past that area, such as text in a short panel, sets ignoreScissor (p. 539).

C++

```
Image panel(&scene);
panel.createImage();
panel.setTexture("ui/panel_bg.png");
panel.setPatchMargin(20);  // 20px on all sides
panel.setSize(300, 200);
```

Lua

```
local panel = Image(scene)
panel:setTexture("ui/panel_bg.png")
panel:setPatchMargin(20)
panel:setSize(300, 200)
```

textureScaleFactor

- *Setter*: void **setTextureScaleFactor**(float textureScaleFactor)
- *Getter*: float **getTextureScaleFactor**() const

Insets the UV rectangle by this many texels on every side, `0.0` by default. A small value such as `0.5` pulls the sampled area away from the edges of a packed atlas cell, hiding the neighbouring pixels that bilinear filtering would otherwise bleed in.

Method details

createImage

- bool **createImage**()

Explicitly builds the image geometry and GPU buffers. This method is **C++ only** and is normally optional because the UI system builds and rebuilds images automatically. Use it when C++ code needs the geometry immediately.

setTexture

- void **setTexture**(const std::string& path)
- void **setTexture**(const std::string& id, TextureData data)
- void **setTexture**(Framebuffer* framebuffer)

Assigns the texture to display. Pass a file path, in-memory pixel data with a cache id, or a framebuffer for render-to-texture UI.

getAABB / getWorldAABB

- AABB (p. 632) **getAABB**() const
- AABB (p. 632) **getWorldAABB**() const

Returns the axis-aligned bounding box in local or world space. Useful for hit-testing or layout calculations.

customShader

- *Setter:* `void setCustomShader(const std::string& path)`
- *Getter:* `std::string getCustomShader() const`

Project-relative base path of a forked shader (`"shaders/glow"` resolves to `shaders/glow.vert` and `shaders/glow.frag`, or `"a.vert|b.frag"` names the files separately). Empty (default) uses the scene default shader

for this type, or the engine built-in. Changing it reloads the component. Same semantics as Mesh.customShader (p. 441); see Custom Shaders (p. 143).

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`
- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the custom shader declares with `#pragma param`, by name. Setting a value only rewrites the uploaded block, so it can run every frame; a parameter with no value reads the scene's value under a scene default shader (p. 405), else its declared default. Same semantics as Mesh.setShaderUniform (p. 445); see Custom Shaders — Shader parameters (p. 147).

C++

```
image.setCustomShader("shaders/glow");
image.setShaderUniform("strength", 0.8f);
image.setShaderUniform("glowColor", Vector4(1.0f, 0.8f, 0.2f, 1.0f));
```

Lua

```
image.customShader = "shaders/glow"
image:setShaderUniform("strength", 0.8)
image:setShaderUniform("glowColor", Vector4(1, 0.8, 0.2, 1))
```

Callback event details

onGetFocus / onLostFocus

- *Property*: FunctionSubscribe<void()> **onGetFocus**
- *Property*: FunctionSubscribe<void()> **onLostFocus**

Called when this element gains or loses keyboard focus.

onPointerEnter / onPointerLeave

- *Property*: FunctionSubscribe<void(float,float)> **onPointerEnter**
- *Property*: FunctionSubscribe<void(float,float)> **onPointerLeave**
- *Callback*: void(float x, float y)

Called when the mouse pointer enters or leaves the element boundary.

C++

```
Image btn(&scene);
btn.createImage();
btn.getComponent<UIComponent>().onPointerEnter.add("enter", [&](float x, float y){
    btn.setColor(Vector4(0.8f, 0.8f, 1.0f, 1.0f));  // highlight
});
btn.getComponent<UIComponent>().onPointerLeave.add("leave", [&](float x, float y){
    btn.setColor(Vector4(1.0f, 1.0f, 1.0f, 1.0f));  // reset
});
```

Lua

```lua
local ui = btn:getUIComponent()
ui.onPointerEnter = function(x, y)
    btn.color = Vector4(0.8, 0.8, 1.0, 1.0)
end
ui.onPointerLeave = function(x, y)
    btn.color = Vector4(1.0, 1.0, 1.0, 1.0)
end
```

onPointerMove

- *Property*: FunctionSubscribe<void(float,float)> **onPointerMove**
- *Callback*: void(float x, float y)

Called each frame the pointer moves while over this element.

onPointerDown / onPointerUp

- *Property*: FunctionSubscribe<void(float,float)> **onPointerDown**
- *Property*: FunctionSubscribe<void(float,float)> **onPointerUp**
- *Callback*: void(float x, float y)

Called on mouse/touch press and release.

onClick / onDoubleClick

- *Property*: FunctionSubscribe<void(float,float)> **onClick**
- *Property*: FunctionSubscribe<void(float,float)> **onDoubleClick**
- *Callback*: void(float x, float y)

Called on a complete click (press + release inside the element) or double click.

onDragStart / onDrag / onDragEnd

- *Property*: FunctionSubscribe<void(float,float)> **onDragStart**
- *Property*: FunctionSubscribe<void(float,float)> **onDrag**
- *Property*: FunctionSubscribe<void(float,float)> **onDragEnd**
- *Callback*: void(float x, float y)

Fired when the user begins dragging, continues dragging, or releases while dragging this element.

Panel

Inherits: Image (p. 513)
C++ type: `Panel`

Description

A draggable, resizable floating window panel. `Panel` extends Image (p. 513) with a header bar that can display a title, and an optional container area below it. Users can drag the header to move the panel, drag the edges to resize it, and click to bring it to the front of other panels.

All child UI elements placed inside the panel are automatically clipped to the panel's content area.

Properties

Type	Name	Default	Langs
std::string	title	`""`	C++ | Lua
AnchorPreset (p. 538)	titleAnchorPreset	`CENTER`	C++ | Lua
Vector4	titleColor	`(1,1,1,1)`	C++ | Lua
std::string	titleFont	system default	C++ | Lua
unsigned int	titleFontSize	`16`	C++ | Lua
Vector4	headerColor	`(1,1,1,1)`	C++ | Lua
int	headerMarginLeft	`0`	C++ | Lua
int	headerMarginTop	`0`	C++ | Lua
int	headerMarginRight	`0`	C++ | Lua
int	headerMarginBottom	`0`	C++ | Lua
int	minWidth	`0`	C++ | Lua
int	minHeight	`0`	C++ | Lua
int	resizeMargin	`6`	C++ | Lua
bool	canMove	`true`	C++ | Lua
bool	canResize	`true`	C++ | Lua
bool	canBringToFront	`true`	C++ | Lua

Methods

Type	Name	Langs
bool	hasHeader	C++
Image (p. 513)	getHeaderImageObject	C++ | Lua
Container (p. 510)	getHeaderContainerObject	C++ | Lua
Text (p. 528)	getHeaderTextObject	C++ | Lua
void	setHeaderPatchMargin	C++ | Lua
void	setHeaderTexture	C++ | Lua
void	setHeaderMargin	C++ | Lua
void	setMinSize	C++ | Lua
void	setWindowProperties	C++ | Lua

Property details

title

- *Setter*: void **setTitle**(const std::string& text)
- *Getter*: std::string **getTitle**() const
- *Setter*: void **setTitleAnchorPreset**(AnchorPreset (p. 538) preset)
- *Setter*: void **setTitleColor**(Vector4 color)
- *Setter*: void **setTitleFont**(const std::string& font)
- *Setter*: void **setTitleFontSize**(unsigned int fontSize)

The window title shown in the header bar. Setting a non-empty title automatically creates child Text (p. 528) and Container (p. 510) entities for the header.

C++

```cpp
Panel win(&scene);
win.createImage();
win.setTexture("ui/window_bg.png");
win.setHeaderTexture("ui/window_header.png");
win.setHeaderPatchMargin(8);
win.setTitle("Settings");
win.setSize(400, 300);
win.setAnchorPreset(AnchorPreset::CENTER);
```

Lua

```lua
local win = Panel(scene)
win:setTexture("ui/window_bg.png")
win:setHeaderTexture("ui/window_header.png")
win:setHeaderPatchMargin(8)
win.title = "Settings"
win:setSize(400, 300)
win.anchorPreset = AnchorPreset.CENTER
```

header

- *Setter*: void **setHeaderTexture**(const std::string& path)
- *Setter*: void **setHeaderColor**(Vector4 color)
- *Setter*: void **setHeaderPatchMargin**(int margin)
- *Setter*: void **setHeaderMargin**(int left, int top, int right, int bottom)
- *Getters*: **getHeaderMarginLeft**, **getHeaderMarginTop**, **getHeaderMarginRight**, **getHeaderMarginBottom**
- *Accessors*: **getHeaderImageObject**, **getHeaderContainerObject**, **getHeaderTextObject**

Configuration for the panel's header bar. `setHeaderMargin()` defines the pixel inset from the panel edges to where the header bar begins.

minWidth / minHeight

- *Setter*: void **setMinSize**(int minWidth, int minHeight)
- *Setter*: void **setMinWidth**(int minWidth) / **setMinHeight**(int minHeight)
- *Getter*: int **getMinWidth**() / **getMinHeight**() const

Minimum dimensions when canResize is `true`.

resizeMargin

- *Setter*: void **setResizeMargin**(int resizeMargin)
- *Getter*: int **getResizeMargin**() const

The width in pixels of the invisible hit area at the panel edges that triggers resize drag. Default is `6`.

windowProperties

- *Setter*: void **setWindowProperties**(bool canMove, bool canResize, bool canBringToFront)
- *Setter/Getter*: **setCanMove** / **isCanMove**
- *Setter/Getter*: **setCanResize** / **isCanResize**
- *Setter/Getter*: **setCanBringToFront** / **isCanBringToFront**

Controls interactive window behaviours:

- **canMove** — user can drag the header to move the window.
- **canResize** — user can drag the edges to resize the window.
- **canBringToFront** — clicking anywhere on the panel raises its Z-order above siblings.

Method details

hasHeader

- bool **hasHeader**() const

Returns `true` if a header bar has been created (i.e. `setTitle()` has been called or a header texture has been set).

Polygon

Inherits: UILayout (p. 537)
C++ type: `Polygon`

`Polygon` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`color`	C++ | Lua
`alpha`	C++ | Lua
`flipY`	C++ | Lua
`customShader`	C++ | Lua

Methods

Name	Languages
`addVertex`	C++ | Lua
`setColor`	C++ | Lua
`setTexture`	C++ | Lua
`getAABB`	C++ | Lua
`getWorldAABB`	C++ | Lua
`setShaderUniform`	C++ | Lua
`getShaderUniform`	C++ | Lua
`removeShaderUniform`	C++ | Lua
`getPolygonComponent` (PolygonComponent (p. 684))	Lua
`getUIComponent` (UIComponent (p. 698))	Lua

Method details

customShader

- *Setter:* `void setCustomShader(const std::string& path)`
- *Getter:* `std::string getCustomShader() const`

Project-relative base path of a forked shader (`"shaders/glow"` resolves to `shaders/glow.vert` and `shaders/glow.frag`, or `"a.vert|b.frag"` names the files separately). Empty (default) uses the scene default shader for this type, or the engine built-in. Changing it reloads the component. Same semantics as Mesh.customShader (p. 441); see Custom Shaders (p. 143).

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`
- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the custom shader declares with `#pragma param`, by name. Setting a value only rewrites the uploaded block, so it can run every frame; a parameter with no value reads the scene's value under a scene default shader (p. 405), else its declared default. Same semantics as Mesh.setShaderUniform (p. 445); see Custom Shaders — Shader parameters (p. 147).

C++

```
polygon.setCustomShader("shaders/glow");
polygon.setShaderUniform("strength", 0.8f);
polygon.setShaderUniform("glowColor", Vector4(1.0f, 0.8f, 0.2f, 1.0f));
```

Lua

```
polygon.customShader = "shaders/glow"
polygon:setShaderUniform("strength", 0.8)
polygon:setShaderUniform("glowColor", Vector4(1, 0.8, 0.2, 1))
```

Progressbar

Inherits: Image (p. 513)
C++ type: `Progressbar`

Description

A UI widget that displays a value in the `[0, 1]` range as a filled bar. The bar can grow horizontally (left to right) or vertically (bottom to top). Both the background and the fill area support 9-patch textures and independent tint colours.

Properties

Type	Name	Default	Langs
ProgressbarType	type	`HORIZONTAL`	C++ | Lua
float	value	`0.0`	C++ | Lua
Vector4	fillColor	`(1,1,1,1)`	C++ | Lua
float	fillAlpha	`1.0`	C++ | Lua

Methods

Type	Name	Langs
Image (p. 513)	getFillObject	C++ | Lua
void	setFillTexture	C++ | Lua
void	setFillColor	C++ | Lua
void	setFillPatchMargin	C++ | Lua
void	setFillMargin	C++ | Lua

Enumerations

ProgressbarType

- **HORIZONTAL** — Bar fills from left to right.
- **VERTICAL** — Bar fills from bottom to top.

Property details

type

- *Setter*: void **setType**(ProgressbarType type)
- *Getter*: ProgressbarType **getType**() const

Direction of growth.

value

- *Setter*: void **setValue**(float value)
- *Getter*: float **getValue**() const

Current fill amount. `0.0` = empty, `1.0` = full. Clamped to `[0, 1]`.

C++

```cpp
Progressbar hp(&scene);
hp.createImage();
hp.setTexture("ui/progressbar_bg.png");
hp.setFillTexture("ui/progressbar_fill.png");
hp.setPatchMargin(6);
hp.setFillColor(Vector4(0.2f, 0.8f, 0.2f, 1.0f));  // green
hp.setSize(200, 24);
hp.setValue(0.75f);
```

Lua

```lua
local hp = Progressbar(scene)
hp:setTexture("ui/progressbar_bg.png")
hp:setFillTexture("ui/progressbar_fill.png")
hp:setPatchMargin(6)
hp.fillColor = Vector4(0.2, 0.8, 0.2, 1.0)
hp:setSize(200, 24)
hp.value = 0.75
```

Method details

getFillObject

- Image (p. 513) **getFillObject**() const

Returns the child Image (p. 513) entity that renders the fill area. Use this to set fill properties directly if you need more control than the convenience setters provide.

setFillTexture

- void **setFillTexture**(const std::string& path)
- void **setFillTexture**(Framebuffer* framebuffer)

Sets the texture for the fill portion of the bar.

setFillColor

- void **setFillColor**(Vector4 color)
- void **setFillColor**(float red, float green, float blue, float alpha)
- void **setFillAlpha**(float alpha)

Tint colour applied to the fill texture.

setFillPatchMargin

- void **setFillPatchMargin**(int margin)
- void **setFillPatchMargin**(int marginLeft, int marginRight, int marginTop, int marginBottom)

9-patch insets for the fill image.

setFillMargin

- void **setFillMargin**(int margin)
- void **setFillMargin**(int marginLeft, int marginRight, int marginTop, int marginBottom)

Pixel offset between the background edges and the fill image.

Scrollbar

Inherits: Image (p. 513)
C++ type: `Scrollbar`

Description

A UI scrollbar widget. The scrollbar consists of a track (the background Image (p. 513)) and a draggable bar handle (a child Image). Both can have independent textures and tint colours. The current scroll position is reported through the `onChange` callback.

Properties

Type	Name	Default	Langs
ScrollbarType	type	`VERTICAL`	C++ | Lua
float	barSize	`0.5`	C++ | Lua
float	step	`0.0`	C++ | Lua
Vector4	barColor	`(1,1,1,1)`	C++ | Lua
float	barAlpha	`1.0`	C++ | Lua

Methods

Type	Name	Langs
Image (p. 513)	getBarObject	C++ | Lua
void	setBarTexture	C++ | Lua
void	setBarColor	C++ | Lua
void	setBarPatchMargin	C++ | Lua
void	setBarMargin	C++ | Lua

Callback events

Callback	Name	Langs
void(float)	onChange	C++ | Lua

Enumerations

ScrollbarType

- **VERTICAL** — Bar slides up and down.
- **HORIZONTAL** — Bar slides left and right.

Property details

type

- *Setter*: void **setType**(ScrollbarType type)
- *Getter*: ScrollbarType **getType**() const

Orientation of the scrollbar.

barSize

- *Setter*: void **setBarSize**(float size)
- *Getter*: float **getBarSize**() const

The relative size of the draggable handle as a fraction of the track length, in `[0, 1]`. Larger values represent a proportionally large viewport relative to the content (e.g. `0.5` means the viewport covers half the content).

step

- *Setter*: void **setStep**(float step)
- *Getter*: float **getStep**() const

Snap increment in `[0, 1]`. When non-zero, the scroll position snaps to multiples of this value. `0` means continuous scrolling.

Method details

getBarObject

- Image (p. 513) **getBarObject**() const

Returns the child Image (p. 513) handle entity for the draggable bar.

setBarTexture

- void **setBarTexture**(const std::string& path)
- void **setBarTexture**(Framebuffer* framebuffer)

Sets the texture for the bar handle.

setBarColor

- void **setBarColor**(Vector4 color)
- void **setBarColor**(float red, float green, float blue, float alpha)
- void **setBarAlpha**(float alpha)

Tint colour applied to the bar handle texture.

setBarPatchMargin

- void **setBarPatchMargin**(int margin)
- void **setBarPatchMargin**(int marginLeft, int marginRight, int marginTop, int marginBottom)

9-patch insets for the bar handle image.

setBarMargin

- void **setBarMargin**(int margin)
- void **setBarMargin**(int marginLeft, int marginRight, int marginTop, int marginBottom)

Pixel offset between the track edges and the bar handle.

C++

```cpp
Scrollbar vsb(&scene);
vsb.createImage();
vsb.setTexture("ui/scrollbar_track.png");
vsb.setBarTexture("ui/scrollbar_handle.png");
vsb.setBarPatchMargin(4);
vsb.setBarSize(0.3f);
vsb.setSize(16, 300);

vsb.getComponent<ScrollbarComponent>().onChange.add("scroll", [&](float value){
    // value is in [0, 1]
    contentContainer.setPositionYOffset(-value * contentHeight);
});
```

Lua

```lua
local vsb = Scrollbar(scene)
vsb:setTexture("ui/scrollbar_track.png")
vsb:setBarTexture("ui/scrollbar_handle.png")
vsb:setBarPatchMargin(4)
vsb.barSize = 0.3
vsb:setSize(16, 300)

local vsbComp = vsb:getScrollbarComponent()
vsbComp.onChange = function(value)
    contentContainer.positionYOffset = -value * contentHeight
end
```

Callback event details

onChange

- *Property*: FunctionSubscribe<void(float)> **onChange**
- *Callback*: void(float value)

Called whenever the scroll position changes. `value` is in `[0, 1]` where `0` is the start of the track and `1` is the end.

Text

Inherits: UILayout (p. 537)
C++ type: `Text`

Description

Renders a unicode string with a TrueType or OpenType font. `Text` produces a quad mesh from glyph data at the specified font size. It supports multi-line wrapping, fixed-size constraints, and exposes per-character layout information for custom cursor or selection rendering.

Text is shaped before it is drawn, so scripts that need contextual glyph forms work without any extra setup: Arabic letters join, ligatures and kerning are applied, and right-to-left runs are reordered by the Unicode Bidirectional Algorithm. A line mixing Arabic and Latin is laid out with each run in its own direction.

Properties

Type	Name	Default	Langs
std::string	text	`""`	C++ | Lua
std::string	font (slot 0)	`""` (built-in)	C++ | Lua
unsigned int	fontSize	`20`	C++ | Lua
bool	multiline	`true`	C++ | Lua
unsigned int	maxTextSize	`100`	C++ | Lua
Vector4	color	`(1,1,1,1)`	C++ | Lua
float	alpha	`1.0`	C++ | Lua
bool	fixedWidth	`false`	C++ | Lua
bool	fixedHeight	`false`	C++ | Lua
bool	flipY	`false`	C++ | Lua
bool	pivotBaseline	`false`	C++ | Lua
bool	pivotCentered	`false`	C++ | Lua
string	customShader	`""`	C++ | Lua

Methods

Type	Name	Langs
bool	createText	C++
float	getAscent	C++ | Lua
float	getDescent	C++ | Lua
float	getLineGap	C++ | Lua
int	getLineHeight	C++ | Lua
unsigned int	getNumChars	C++ | Lua
Vector2	getCharPosition	C++ | Lua
float	getCharWidth	C++ | Lua
AABB (p. 632)	getAABB	C++ | Lua
AABB (p. 632)	getWorldAABB	C++ | Lua
void	setShaderUniform	C++ | Lua
Vector4	getShaderUniform	C++ | Lua
bool	removeShaderUniform	C++ | Lua
TextComponent (p. 693)	`getTextComponent`	Lua

Type	Name	Langs
UIComponent (p. 698)	`getUIComponent`	Lua

Property details

text

- *Setter*: void **setText**(const std::string& text)
- *Getter*: std::string **getText**() const

The string to render. Supports UTF-8 encoded unicode.

C++

```
Text label(&scene);
label.createText();
label.setFont("fonts/Roboto-Regular.ttf");
label.setFontSize(24);
label.setText("Score: 0");
label.setAnchorPreset(AnchorPreset::TOP_LEFT);
```

Lua

```
local label = Text(scene)
label.font = "fonts/Roboto-Regular.ttf"
label.fontSize = 24
label.text = "Score: 0"
label.anchorPreset = AnchorPreset.TOP_LEFT
```

font

- *Setter*: void **setFont**(const std::string& font)
- *Setter*: void **setFont**(unsigned int index, const std::string& font)
- *Getter*: std::string **getFont**() const
- *Getter*: std::string **getFont**(unsigned int index) const

File path to a TTF/OTF font file, or a TrueType collection (`.ttc`). The `font` property and the overloads without an index read or write slot `0`, the main font. Setting the main font does not clear the fallback slots.

Doriax stores a fixed `FontArray` with `MAX_TEXT_FONTS` slots (four by default). Index `0` is the main font; indexes `1` through `3` are tried in order when an earlier font does not contain a character. Empty slots are skipped. Indexes are zero-based in both C++ and Lua.

Two fonts are built into the engine and always close the chain: a Roboto subset covering Latin, Cyrillic, Greek and Vietnamese, and an Arabic subset of Noto Sans Arabic. Any codepoint the custom fonts do not cover is drawn from them, so setting an Arabic-only font still renders Latin, and leaving slot `0` unset still renders Arabic. A codepoint no font in the chain covers is drawn as the missing-glyph box.

A missing main-font file prevents that atlas from loading. A missing fallback is logged and skipped, so later fallbacks and the built-in fonts can still render the text.

Use the indexed overloads to assign fallback fonts:

C++

```
Text label(&scene);
label.setFont("fonts/MyLatin.ttf");
label.setFont(1, "fonts/NotoKufiArabic.ttf");
label.setFont(2, "fonts/NotoSansCJK.ttc");
label.setText("hello مرحبا 你好");
```

Lua

```lua
local label = Text(scene)
label.font = "fonts/MyLatin.ttf"
label:setFont(1, "fonts/NotoKufiArabic.ttf")
label:setFont(2, "fonts/NotoSansCJK.ttc")
label.text = "hello مرحبا 你好"
```

> **Note**
> Texts only share a glyph atlas when their whole chain matches, so keep the slots identical across elements that use the same font.

An out-of-range index is rejected and logged. `getFont(index)` returns an empty string for an invalid index.

At the component level, `TextComponent::font` is the same `FontArray`, so direct C++ component access uses `text.font[0]` for the main font and `text.font[1]` onward for fallbacks.

> **Migrating from fontFallbacks**
> The `fontFallbacks` property and `setFontFallbacks` / `getFontFallbacks` methods no longer exist. Replace a semicolon-separated fallback list with one indexed `setFont` call per font. The editor still accepts the former scalar `font` scene field as slot `0`; newly saved scenes write `font` as an ordered sequence.

fontSize

- *Setter*: void **setFontSize**(unsigned int fontSize)
- *Getter*: unsigned int **getFontSize**() const

The glyph render size in pixels (points at 72 DPI).

multiline

- *Setter*: void **setMultiline**(bool multiline)
- *Getter*: bool **getMultiline**() const

When `true`, the text wraps to multiple lines at the element's width boundary.

maxTextSize

- *Setter*: void **setMaxTextSize**(unsigned int maxTextSize)
- *Getter*: unsigned int **getMaxTextSize**() const

Number of characters the vertex buffer is preallocated for. It is not a limit: a longer string grows the buffer automatically and logs a warning. Raise it up front for text that is known to be long, to avoid the reallocation.

color

- *Setter*: void **setColor**(Vector4 color)
- *Setter*: void **setAlpha**(float alpha)
- *Getter*: Vector4 **getColor**() const / float **getAlpha**() const

RGBA tint colour applied to all glyphs, in sRGB. It is stored linear internally, so the value read back is the sRGB form of the stored one.

fixedWidth / fixedHeight

- *Setter*: void **setFixedWidth**(bool fixedWidth) / **setFixedHeight**(bool fixedHeight)

- *Setter*: void **setFixedSize**(bool fixedSize) — sets both at once
- *Getter*: bool **isFixedWidth**() / **isFixedHeight**() const

When `fixedWidth` is `false`, the text element automatically resizes its width to fit the rendered text. Same applies for height. Set to `true` to keep a fixed size and clip/wrap text instead.

flipY

- *Setter*: void **setFlipY**(bool flipY)
- *Getter*: bool **isFlipY**() const

Flips glyph texture coordinates vertically. The engine normally picks this automatically from the camera; calling the setter takes over and pins the value.

pivotBaseline / pivotCentered

- *Setter*: void **setPivotBaseline**(bool pivotBaseline) / **setPivotCentered**(bool pivotCentered)
- *Getter*: bool **isPivotBaseline**() / **isPivotCentered**() const

Controls the local-space pivot point. `pivotBaseline` moves the pivot to the text baseline; `pivotCentered` centres it on the bounding box.

Method details

createText

- bool **createText**()

Explicitly builds the text geometry and GPU buffers. This method is **C++ only** and is normally optional because the UI system builds and rebuilds text automatically. Use it when C++ code needs the geometry immediately.

getAscent / getDescent / getLineGap / getLineHeight

- float **getAscent**() const
- float **getDescent**() const
- float **getLineGap**() const
- int **getLineHeight**() const

Font metrics in pixels for the current `fontSize`. Useful for precise cursor or selection rendering.

getNumChars

- unsigned int **getNumChars**() const

Number of codepoints in the current string, excluding line breaks. This differs from `strlen` for multi-byte UTF-8 input, and from the number of glyphs drawn: shaping can merge codepoints into one ligature or expand one into several marks.

getCharPosition

- Vector2 **getCharPosition**(unsigned int index) const

Returns the local-space pen position after the codepoint at `index`, in logical (string) order. Codepoints that shaping merged into a single cluster share one position.

For a caret or a selection in text that can be right-to-left, this is not enough on its own, because logical order and screen order differ. TextEdit (p. 533) handles that case internally.

getCharWidth

- float **getCharWidth**(uint32_t codepoint) const

Returns the advance width of `codepoint` in pixels at the current font size, taken from the first font in the chain that covers it. Returns `0` when the font is not loaded yet.

This is the advance of the glyph in isolation. It is not the width the codepoint takes inside a string, where shaping may change the form, apply kerning, or merge it into a ligature.

getAABB / getWorldAABB

- AABB (p. 632) **getAABB**() const
- AABB (p. 632) **getWorldAABB**() const

Returns the axis-aligned bounding box of the rendered text in local or world space.

customShader

- *Setter:* `void setCustomShader(const std::string& path)`
- *Getter:* `std::string getCustomShader() const`

Project-relative base path of a forked shader (`"shaders/outline"` resolves to `shaders/outline.vert` and `shaders/outline.frag`, or `"a.vert|b.frag"` names the files separately). Empty (default) uses the scene default shader for this type, or the engine built-in. Changing it reloads the component. Same semantics as Mesh.customShader (p. 441); see Custom Shaders (p. 143).

setShaderUniform

- `void setShaderUniform(const std::string& name, const Vector4& value)`
- `void setShaderUniform(const std::string& name, const Vector3& value)`
- `void setShaderUniform(const std::string& name, const Vector2& value)`
- `void setShaderUniform(const std::string& name, float value)`
- `Vector4 getShaderUniform(const std::string& name) const`
- `bool removeShaderUniform(const std::string& name)`

Values of the parameters the custom shader declares with `#pragma param`, by name. Setting a value only rewrites the uploaded block, so it can run every frame; a parameter with no value reads the scene's value under a scene default shader (p. 405), else its declared default. Same semantics as Mesh.setShaderUniform (p. 445); see Custom Shaders — Shader parameters (p. 147).

C++

```
text.setCustomShader("shaders/outline");
text.setShaderUniform("strength", 0.8f);
text.setShaderUniform("glowColor", Vector4(1.0f, 0.8f, 0.2f, 1.0f));
```

Lua

```
text.customShader = "shaders/outline"
text:setShaderUniform("strength", 0.8)
text:setShaderUniform("glowColor", Vector4(1, 0.8, 0.2, 1))
```

TextEdit

Inherits: Image (p. 513)
C++ type: `TextEdit`

Description

An interactive single-line text input field. `TextEdit` composes an Image (p. 513) background, a Text (p. 528) display layer, and polygon overlays for the cursor and text selection highlight. It handles keyboard input, clipboard operations, cursor navigation, and optional password masking.

Editing works in text whose direction is not left to right. **Left** and **Right** move the caret one stop across the screen rather than one position along the string, matching where a click would put it, and **Home** and **End** go to the start and end of the line, which in a right-to-left line are its right and left edges. A selection that is contiguous in the string can appear as more than one highlighted block when it crosses a direction change.

Properties

Type	Name	Default	Langs
std::string	text	`""`	C++ | Lua
std::string	textFont	`""` (built-in)	C++ | Lua
unsigned int	fontSize	`20`	C++ | Lua
Vector4	textColor	`(1,1,1,1)`	C++ | Lua
unsigned int	maxTextSize	`100`	C++ | Lua
std::string	placeholder	`""`	C++ | Lua
Vector4	placeholderColor	`(0.76,0.76,0.76,1)`	C++ | Lua
bool	password	`false`	C++ | Lua
char	passwordChar	`'*'`	C++ | Lua
bool	disabled	`false`	C++ | Lua
int	cursorIndex	`0`	C++ | Lua
float	cursorBlink	`0.6`	C++ | Lua
float	cursorWidth	`2.0`	C++ | Lua
Vector4	cursorColor	`(0,0,0,1)`	C++ | Lua
Vector4	selectionColor	`(0.53,0.70,0.93,0.35)`	C++ | Lua

Methods

Type	Name	Langs
Text (p. 528)	getTextObject	C++ | Lua
void	setSelection	C++ | Lua
int	getSelectionAnchor	C++ | Lua
int	getSelectionFocus	C++ | Lua

Property details

text

- *Setter*: void **setText**(const std::string& text)
- *Getter*: std::string **getText**() const

The current content of the input field.

C++

```cpp
TextEdit nameField(&scene);
nameField.createImage();
nameField.setTexture("ui/input_bg.png");
nameField.setPatchMargin(8);
nameField.setSize(280, 36);
nameField.setPlaceholder("Enter your name...");
nameField.setFontSize(18);
```

Lua

```lua
local nameField = TextEdit(scene)
nameField:setTexture("ui/input_bg.png")
nameField:setPatchMargin(8)
nameField:setSize(280, 36)
nameField.placeholder = "Enter your name..."
nameField.fontSize = 18
```

textColor

- *Setter*: void **setTextColor**(Vector4 color)
- *Getter*: Vector4 **getTextColor**() const

RGBA tint colour applied to the input text glyphs.

textFont

- *Setter*: void **setTextFont**(const std::string& font)
- *Getter*: std::string **getTextFont**() const

Path to the TTF/OTF font file used by the inner Text (p. 528). Falls back to the built-in fonts for codepoints it does not cover, see Text.font (p. 529).

fontSize

- *Setter*: void **setFontSize**(unsigned int fontSize)
- *Getter*: unsigned int **getFontSize**() const

Glyph size in pixels.

maxTextSize

- *Setter*: void **setMaxTextSize**(unsigned int maxTextSize)
- *Getter*: unsigned int **getMaxTextSize**() const

Maximum allowed number of characters. `0` means no limit.

placeholder

- *Setter*: void **setPlaceholder**(const std::string& placeholder)
- *Getter*: std::string **getPlaceholder**() const
- *Setter*: void **setPlaceholderColor**(Vector4 color)
- *Getter*: Vector4 **getPlaceholderColor**() const

Ghost text displayed when the field is empty. Typically a prompt like `"Type here…"`.

password

- *Setter*: void **setPassword**(bool password)
- *Getter*: bool **getPassword**() const
- *Setter*: void **setPasswordChar**(char passwordChar)
- *Getter*: char **getPasswordChar**() const

When `password` is `true`, each character is displayed as `passwordChar` (default `'*'`). The actual text is still accessible via `getText()`.

disabled

- *Setter*: void **setDisabled**(bool disabled)
- *Getter*: bool **getDisabled**() const

When `true`, the field ignores keyboard and pointer events.

cursorIndex

- *Setter*: void **setCursorIndex**(int cursorIndex)
- *Getter*: int **getCursorIndex**() const

Current insertion-point position in the string, not on screen (0 = before the first character, `getNumChars()` = after the last).

Where the direction changes, two neighbouring indices can sit on the same pixel. They are still separate insertion points, and typing at one does not give the same result as typing at the other, so arrow keys step through both.

cursorBlink / cursorWidth / cursorColor

- *Setters/Getters*: **setCursorBlink** / **getCursorBlink**, **setCursorWidth** / **getCursorWidth**, **setCursorColor** / **getCursorColor**

Visual properties of the cursor caret. `cursorBlink` is the on/off period in seconds.

Colours are sRGB, matching what a colour picker shows. They are stored linear internally, so the value read back is the sRGB form of the stored one.

selectionColor

- *Setter*: void **setSelectionColor**(Vector4 color)
- *Getter*: Vector4 **getSelectionColor**() const

Background colour of the text selection highlight polygon, in sRGB. The default is a translucent blue.

A selection that crosses a direction change is drawn as several blocks rather than one, since the selected characters are not contiguous on screen.

Method details

getTextObject

- Text (p. 528) **getTextObject**() const

Returns the child Text (p. 528) entity used for rendering the input content.

setSelection

- void **setSelection**(int anchor, int focus)
- int **getSelectionAnchor**() const
- int **getSelectionFocus**() const

Programmatically set or query the text selection. `anchor` is the fixed end of the selection; `focus` is the cursor end that moves with keyboard navigation. Pass equal values to collapse the selection to a cursor.

UILayout

Inherits: Object (p. 455)
C++ type: `UILayout`

Description

Base class for all UI elements. `UILayout` extends Object (p. 455) with a pixel-based sizing and anchor system that lets widgets automatically resize and reposition relative to their parent container or the screen edges.

The anchor system works in two modes:

- **Fixed size** — the widget has an explicit `width`/`height`. The anchor points define which corner(s) of the parent the widget is attached to; anchor offsets define the pixel distance from those corners.
- **Stretch mode** — by setting anchor points on all four edges (e.g. `FULL_LAYOUT`), the widget automatically fills the available space, optionally with pixel margins.

A UI element's position is its top-left corner, and by default it scales and rotates around that corner too. Set a pivot to grow it from its center instead.

Properties

Type	Name	Default	Langs
unsigned int	width	`0`	C++ | Lua
unsigned int	height	`0`	C++ | Lua
float	anchorPointLeft	`0.0`	C++ | Lua
float	anchorPointTop	`0.0`	C++ | Lua
float	anchorPointRight	`0.0`	C++ | Lua
float	anchorPointBottom	`0.0`	C++ | Lua
int	anchorOffsetLeft	`0`	C++ | Lua
int	anchorOffsetTop	`0`	C++ | Lua
int	anchorOffsetRight	`0`	C++ | Lua
int	anchorOffsetBottom	`0`	C++ | Lua
AnchorPreset	anchorPreset	`NONE`	C++ | Lua
Vector2	positionOffset	`(0,0)`	C++ | Lua
float	positionXOffset	`0.0`	C++ | Lua
float	positionYOffset	`0.0`	C++ | Lua
Vector2	pivot	`(0,0)`	C++ | Lua
bool	usingAnchors	`true`	C++ | Lua
bool	ignoreScissor	`false`	C++ | Lua

Methods

Type	Name	Langs
void	setSize	C++ | Lua
void	setAnchorPoints	C++ | Lua
void	setAnchorOffsets	C++ | Lua
void	setAnchorPreset	C++
void	setPositionOffset	C++
void	setPivot	C++ | Lua

Enumerations

AnchorPreset

Convenience presets for common anchor configurations:

- **NONE** — No anchor preset applied.
- **TOP_LEFT / TOP_RIGHT / BOTTOM_LEFT / BOTTOM_RIGHT** — Fixed-size widget anchored to a corner.
- **CENTER_LEFT / CENTER_TOP / CENTER_RIGHT / CENTER_BOTTOM** — Fixed-size widget centred on an edge.
- **CENTER** — Fixed-size widget centred in the parent.
- **LEFT_WIDE / TOP_WIDE / RIGHT_WIDE / BOTTOM_WIDE** — Stretches along one axis, fixed on the other.
- **VERTICAL_CENTER_WIDE** — Stretches vertically, centred horizontally.
- **HORIZONTAL_CENTER_WIDE** — Stretches horizontally, centred vertically.
- **FULL_LAYOUT** — Stretches to fill the entire parent.

Property details

width / height

- *Setter*: void **setWidth**(unsigned int width) / **setHeight**(unsigned int height)
- *Setter*: void **setSize**(unsigned int width, unsigned int height)
- *Getter*: unsigned int **getWidth**() / **getHeight**() const

Pixel dimensions of the UI element. Only used when the anchor configuration does not stretch the element.

anchorPoints

- *Setter*: void **setAnchorPoints**(float left, float top, float right, float bottom)
- Individual setters/getters: **setAnchorPointLeft**, **setAnchorPointTop**, **setAnchorPointRight**, **setAnchorPointBottom**

Normalised values in `[0, 1]` defining where each edge of this element is attached within the parent. For example, `(0, 0, 0, 0)` pins the top-left corner; `(0, 0, 1, 1)` stretches to fill the parent.

anchorOffsets

- *Setter*: void **setAnchorOffsets**(int left, int top, int right, int bottom)
- Individual setters/getters: **setAnchorOffsetLeft**, **setAnchorOffsetTop**, **setAnchorOffsetRight**, **setAnchorOffsetBottom**

Pixel margins from the anchor points. For a `FULL_LAYOUT` preset with a 10-pixel margin, set all offsets to `10`.

anchorPreset

- *Setter*: void **setAnchorPreset**(AnchorPreset anchorPreset)
- *Getter*: AnchorPreset **getAnchorPreset**() const

Applies a AnchorPreset that sets `anchorPoints` automatically.

C++

```
Image overlay(&scene);
overlay.createImage();
overlay.setAnchorPreset(AnchorPreset::FULL_LAYOUT);
```

Lua

```lua
local overlay = Image(scene)
overlay.anchorPreset = AnchorPreset.FULL_LAYOUT
```

positionOffset

- *Setter*: void **setPositionOffset**(Vector2 positionOffset)
- Individual: **setPositionXOffset**, **setPositionYOffset**
- *Getter*: Vector2 **getPositionOffset**() const

An additional pixel offset applied to the final computed position. Useful for fine-tuning placement without modifying the anchor configuration.

pivot

- *Setter*: void **setPivot**(Vector2 pivot)
- *Setter*: void **setPivot**(float x, float y)
- *Getter*: Vector2 **getPivot**() const

The point the element scales and rotates around, as a share of its size: `(0, 0)` is the top-left corner (the default), `(0.5, 0.5)` the center and `(1, 1)` the bottom-right corner. It does not move the element: the position stays the top-left corner of the unscaled element, so anchors and layouts are unaffected. `getWorldPosition()` returns where the top-left corner is after scaling and rotation.

A Text (p. 528) with its own *Center* or *Baseline* option moves its glyphs away from the layout rectangle, so a pivot is measured from that rectangle, not from the glyphs.

C++

```cpp
// a button that grows from its center on hover
Button play(&scene);
play.setPivot(0.5f, 0.5f);
play.getComponent<UIComponent>().onPointerEnter.add("grow", [&](float x, float y){
    play.setScale(1.05f);
});
play.getComponent<UIComponent>().onPointerLeave.add("shrink", [&](float x, float y){
    play.setScale(1.0f);
});
```

Lua

```lua
play:setPivot(0.5, 0.5)
local ui = play:getUIComponent()
ui.onPointerEnter = function(x, y) play:setScale(1.05) end
ui.onPointerLeave = function(x, y) play:setScale(1.0) end
```

usingAnchors

- *Setter*: void **setUsingAnchors**(bool usingAnchors)
- *Getter*: bool **isUsingAnchors**() const

When `false`, the element is positioned using the standard 3D transform from Object (p. 455) rather than the anchor system.

ignoreScissor

- *Setter*: void **setIgnoreScissor**(bool ignoreScissor)
- *Getter*: bool **isIgnoreScissor**() const

When `true`, this element is not clipped to the parent's scissor rectangle. Use for tooltips or overlays that must extend beyond a scrollable container.

Physics

Body2D

Description

`Body2D` is the 2D rigid-body handle built on top of the **Box2D v3** physics library. Obtain one by calling `object.getBody2D()`. After creating shapes and setting properties, call `load()` to register the body with the physics world. The body position is synchronized with the parent Object (p. 455) transform automatically each physics step.

Inherits: EntityHandle (p. 711)

C++

```cpp
Sprite ball(&scene);
ball.setTexture("ball.png");
ball.setSize(32, 32);
ball.createSprite();

Body2D body = ball.getBody2D();
body.createCircleShape(Vector2(16, 16), 16.0f);
body.setType(BodyType::DYNAMIC);
body.load();
```

Lua

```lua
local ball = Sprite(scene)
ball:setTexture("ball.png")
ball:setSize(32, 32)

local body = ball:getBody2D()
body:createCircleShape(Vector2(16, 16), 16)
body.type = BodyType.DYNAMIC
body:load()
```

Properties

Type	Name	Default	Languages
BodyType	type	`STATIC`	C++ | Lua
Vector2	position	—	C++ | Lua
float	angle	—	C++ | Lua
Vector2	linearVelocity	`(0,0)`	C++ | Lua
float	angularVelocity	`0`	C++ | Lua
float	linearDamping	`0`	C++ | Lua
float	angularDamping	`0`	C++ | Lua
bool	enableSleep	`true`	C++ | Lua
bool	awake	`true`	C++ | Lua
bool	fixedRotation	`false`	C++ | Lua
bool	linearXLocked	`false`	C++ | Lua *(read-only)*
bool	linearYLocked	`false`	C++ | Lua *(read-only)*
bool	bullet	`false`	C++ | Lua
bool	enabled	`true`	C++ | Lua
float	gravityScale	`1.0`	C++ | Lua
float	mass	—	C++ | Lua

Type	Name	Default	Languages
float	rotationalInertia	—	C++ | Lua

Methods

Returns	Name	Languages
void	load	C++ | Lua
int	createBoxShape	C++ | Lua
int	createCenteredBoxShape	C++ | Lua
int	createRoundedBoxShape	C++ | Lua
int	createPolygonShape	C++ | Lua
int	createCircleShape	C++ | Lua
int	createCapsuleShape	C++ | Lua
int	createSegmentShape	C++ | Lua
int	createChainShape	C++ | Lua
int	createTilemapShape	C++ | Lua
void	removeAllShapes	C++ | Lua
size_t	getNumShapes	C++ | Lua
Shape2DType	getShapeType	C++ | Lua
void	setShapeDensity	C++ | Lua
float	getShapeDensity	C++ | Lua
void	setShapeFriction	C++ | Lua
float	getShapeFriction	C++ | Lua
void	setShapeRestitution	C++ | Lua
float	getShapeRestitution	C++ | Lua
void	setShapeRollingResistance	C++ | Lua
float	getShapeRollingResistance	C++ | Lua
void	setShapeTangentSpeed	C++ | Lua
float	getShapeTangentSpeed	C++ | Lua
void	setShapeSensor	C++ | Lua
bool	isShapeSensor	C++ | Lua
void	setShapeEnableHitEvents	C++ | Lua
void	setShapeContactEvents	C++ | Lua
void	setShapePreSolveEvents	C++ | Lua
void	setShapeSensorEvents	C++ | Lua
void	setBitsFilter	C++ | Lua
void	setCategoryBitsFilter	C++ | Lua
void	setMaskBitsFilter	C++ | Lua
void	setGroupIndexFilter	C++ | Lua
void	setMotionLocks	C++ | Lua
void	applyMassFromShapes	C++ | Lua
void	applyForce	C++ | Lua
void	applyForceToCenter	C++ | Lua
void	applyTorque	C++ | Lua
void	applyLinearImpulse	C++ | Lua
void	applyLinearImpulseToCenter	C++ | Lua

Returns	Name	Languages
void	applyAngularImpulse	C++ | Lua
std::vector<Contact2D>	getBodyContacts	C++ | Lua
std::vector<Contact2D>	getShapeContacts	C++ | Lua
Object	getAttachedObject	C++ | Lua
float	getPointsToMeterScale	C++
Body2DComponent (p. 667)	`getBody2DComponent`	Lua

Enumerations

BodyType

- **STATIC** — Does not move; infinite mass. Used for ground, walls, and platforms.
- **KINEMATIC** — Moved by setting velocity directly, not by forces. Collides with dynamic bodies.
- **DYNAMIC** — Fully simulated; responds to forces, torques, and gravity.

Shape2DType

- **BOX** — Rectangular polygon shape.
- **CIRCLE** — Circle shape defined by center and radius.
- **CAPSULE** — Two circles connected by a rectangle.
- **POLYGON** — Arbitrary convex polygon.
- **SEGMENT** — Line segment (edge shape, no volume). Collides on both sides.
- **CHAIN** — Connected chain of line segments; can be open or closed. Collides on one side only, see createChainShape.

Property details

type

- *Setter:* `void setType(BodyType type)`
- *Getter:* `BodyType getType() const`

Sets the body simulation mode. See BodyType. Must be set before calling load or changed at runtime by re-loading the body.

position / angle

- *Setters:* `void setPosition(Vector2)` / `void setAngle(float angle)`
- *Getters:* `Vector2 getPosition() const` / `float getAngle() const`

The body's own pose in **world space**, read from and written straight to the simulation, and the only way to place a body from `onFixedUpdate`. Setting either also updates the entity Object (p. 455) transform so rendering follows and the next step does not undo the move. Both wake the body. Box2D owns only X and Y, so the Transform's Z is left untouched.

Use these for teleports — respawns, checkpoints, portals. For continuous movement prefer linearVelocity or applyForce, which let the solver resolve collisions instead of pushing bodies through walls.

> **Parented bodies**
> The world pose is converted to the entity's local space using the parent's transform as of the last variable-timestep sync. Moving a body's **parent** and teleporting the body in the same callback therefore places it relative to the parent's *previous* pose — do one or the other per frame. Bodies with no parent, the common case, are exact.

C++

```cpp
Body2D body = object.getBody2D();

body.setPosition(Vector2(0, 240));     // respawn
body.setLinearVelocity(Vector2::ZERO); // stop residual motion
```

Lua

```lua
local body = object:getBody2D()

body.position = Vector2(0, 240)
body.linearVelocity = Vector2(0, 0)
```

> **Do not use Object::setPosition from onFixedUpdate**
> Writing an entity's Object (p. 455) transform only reaches the body between fixed steps. Inside `onFixedUpdate` the physics system syncs the body from the transform *before* stepping and writes the stepped pose back *after*, so a transform written there is discarded unless you also call updateTransform (p. 460) to refresh the world transform the sync reads. Prefer `Body2D::setPosition` / `setAngle`: one call, and it writes the simulation directly. `Object::setPosition` is fine from `onUpdate` and outside the update callbacks.

linearVelocity

- *Setter:* `void setLinearVelocity(Vector2 linearVelocity)`
- *Getter:* `Vector2 getLinearVelocity() const`

Current velocity of the center of mass in points per second — the same units as positions, so `setLinearVelocity(Vector2(300, 0))` crosses 300 points in one second. Forces and impulses are not converted: they stay in Box2D units (Newtons on the meter-scaled body).

angularVelocity

- *Setter:* `void setAngularVelocity(float angularVelocity)`
- *Getter:* `float getAngularVelocity() const`

Current angular velocity in radians per second (regardless of `Engine::useDegrees`).

linearDamping / angularDamping

- *Setters:* `void setLinearDamping(float)` / `void setAngularDamping(float)`
- *Getters:* `float getLinearDamping() const` / `float getAngularDamping() const`

Drag coefficients that slow down translation and rotation over time. Values in `[0, ∞)`. Zero means no damping.

fixedRotation

- *Setter:* `void setFixedRotation(bool fixedRotation)`
- *Getter:* `bool isFixedRotation() const`

When `true`, the body cannot rotate. Useful for top-down characters that should always stay upright. It is the rotation lock of setMotionLocks.

bullet

- *Setter:* `void setBullet(bool bullet)`
- *Getter:* `bool isBullet() const`

Enables continuous collision detection (CCD) for fast-moving objects that might tunnel through thin geometry in a single time step.

gravityScale

- *Setter:* `void setGravityScale(float gravityScale)`
- *Getter:* `float getGravityScale() const`

Multiplier applied to global gravity for this body. `0` means the body is unaffected by gravity; negative values invert gravity.

mass / rotationalInertia

- *Getter:* `float getMass() const` / `float getRotationalInertia() const`

Computed mass and moment of inertia. Updated automatically when applyMassFromShapes is called.

Method details

load

- `void load()`

Registers all shapes with the Box2D world and activates the body. Must be called after creating shapes. Call again to rebuild the body after runtime shape changes.

createBoxShape

- `int createBoxShape(float width, float height)`

Creates a rectangular polygon shape with corner at the entity's local origin. Returns the shape index (0-based).

createCenteredBoxShape

- `int createCenteredBoxShape(float width, float height)`
- `int createCenteredBoxShape(float width, float height, Vector2 center, float angle)`

Creates a rectangular polygon centered at `(width/2, height/2)` relative to the entity origin. The second overload allows an explicit center offset and pre-rotation.

C++

```
body.createCenteredBoxShape(64.0f, 64.0f);
```

Lua

```
body:createCenteredBoxShape(64, 64)
```

createRoundedBoxShape

- `int createRoundedBoxShape(float width, float height, float radius)`

Creates a rounded rectangle (box with chamfered corners). `radius` specifies the corner rounding amount.

createPolygonShape

- `int createPolygonShape(std::vector<Vector2> vertices)`

Creates a convex hull from the given vertices. Box2D requires the shape to be convex; provide vertices in counter-clockwise order.

createCircleShape

- `int createCircleShape(Vector2 center, float radius)`

Creates a circle shape at `center` (in local space) with the given `radius`.

createCapsuleShape

- `int createCapsuleShape(Vector2 center1, Vector2 center2, float radius)`

Creates a capsule defined by two center points and a radius. Good for characters and rounded projectiles.

createSegmentShape

- `int createSegmentShape(Vector2 point1, Vector2 point2)`

Creates a line segment between `point1` and `point2`. It collides on both sides and has no mass. For outlines made of many edges, use a chain instead.

createChainShape

- `int createChainShape(std::vector<Vector2> vertices, bool loop)`

Creates a chain of connected edge segments. When `loop` is `true`, the last point connects back to the first forming a closed boundary.

A chain collides on one side only, to the right of each edge going from one vertex to the next. With counter-clockwise vertices a closed chain blocks bodies outside it; with clockwise vertices it keeps them inside, like a ring or a bowl. Rotating or mirroring the body doesn't change the side. In the editor, ticks on the edges show the colliding side and **Collision Side → Reverse** flips it.

A chain needs at least 4 vertices, and an open chain doesn't collide on its first and last edges. Chains have no mass, so use them on static or kinematic bodies.

C++

```cpp
// clockwise ring: keeps a ball inside
std::vector<Vector2> ring;
for (int i = 0; i < 16; i++){
    float a = -2.0f * M_PI * i / 16.0f;
    ring.push_back(Vector2(170.0f * cos(a), 170.0f * sin(a)));
}
body.createChainShape(ring, true);
body.load();
```

Lua

```lua
-- clockwise ring: keeps a ball inside
local ring = {}
for i = 0, 15 do
    local a = -2 * math.pi * i / 16
    ring[#ring + 1] = Vector2(170 * math.cos(a), 170 * math.sin(a))
end
body:createChainShape(ring, true)
body:load()
```

createTilemapShape

- `int createTilemapShape()`

Creates a polygon shape built from the tiles of the Tilemap (p. 492) on the same entity. Tiles that touch are merged, first into rows and then into blocks of rows with the same width, and each block becomes one box. Tiles whose rect has `collision` (p. 719) set to `false` are left out.

The tiles are read when the body loads. After adding or removing tiles from a script, set `needUpdateShapes` on the Body2DComponent (p. 667) to rebuild the boxes. The shape setters, like `setShapeFriction` or `setBitsFilter`, apply to every box, and getShapeContacts returns the contacts of all of them. Removing the Tilemap removes the boxes.

The shape's own vertices aren't used, so don't read them to find where a body touched it. Use the point and normal of a contact or a Ray (p. 649) hit instead. In C++, `PhysicsSystem::getTilemapCollisionRects` returns the boxes in the tilemap's local space.

C++

```cpp
Tilemap level(&scene);
level.addRect(0, 0, 32, 32);
for (int i = 0; i < 10; i++){
    level.addTile(0, Vector2(i * 32, 0), 32, 32);
}

Body2D body = level.getBody2D();
body.createTilemapShape(); // one 320x32 box
```

Lua

```lua
local level = Tilemap(scene)
level:addRect(0, 0, 32, 32)
for i = 0, 9 do
    level:addTile(0, Vector2(i * 32, 0), 32, 32)
end

local body = level:getBody2D()
body:createTilemapShape() -- one 320x32 box
```

removeAllShapes

- `void removeAllShapes()`

Removes all shapes from the body. Call load again after adding new shapes.

setShapeDensity / setShapeFriction / setShapeRestitution

Per-shape material properties:

- **density** — Mass per unit area (affects computed mass).
- **friction** — Coulomb friction coefficient `[0, ∞)`. `0` = frictionless, `1` = high friction.
- **restitution** — Bounciness `[0, 1]`. `0` = no bounce, `1` = perfectly elastic.

All three come in two overloads: `setXxx(value)` applies to the first shape; `setXxx(index, value)` applies to the shape at `index`.

A friction or restitution change during play reaches bodies already resting on the shape: they wake up and their contacts take the new value on the next step. Rolling resistance and tangent speed work the same way, so a conveyor switched on under a sleeping crate starts carrying it.

setShapeRollingResistance / setShapeTangentSpeed

- **rollingResistance** — Slows circles and capsules rolling on the shape, `[0, ∞)`. `0` (the default) lets them roll freely.
- **tangentSpeed** — Moves bodies along the shape's surface, in points per second, like a conveyor belt. Positive carries them clockwise around the shape: to the right on top of a box. `0` by default.

Both have the same two overloads as friction, and are fields of each shape in the editor (**Rolling Resistance**, **Tangent Speed**). A chain takes them on every edge.

Lua

```lua
local belt = beltObject:getBody2D()
belt:setShapeTangentSpeed(120)
```

setShapeSensor

- `void setShapeSensor(bool sensor)`
- `void setShapeSensor(size_t index, bool sensor)`
- `bool isShapeSensor() const`
- `bool isShapeSensor(size_t index) const`

Turns a shape into a **sensor**: an overlap-only shape that never pushes anything and is reported through `beginSensorContact2D` / `endSensorContact2D` instead of the contact events. Pickups, triggers and hurt zones are the typical use. A sensor always has its sensor events enabled, so the callbacks arrive without further setup; two sensors never report each other. The flag is also a checkbox on the shape in the editor (**Sensor**) and the `shapeSensor` property in Lua. Box2D has no live toggle for it, so changing it rebuilds the body's shapes on the next physics step.

C++

```cpp
Body2D coin = coinSprite.getBody2D();
coin.createCircleShape(Vector2(0, 0), 22.0f);
coin.setShapeSensor(true);
coin.setCategoryBitsFilter(0x0010); // a category the hero can recognize
```

Lua

```lua
local coin = coinSprite:getBody2D()
coin:createCircleShape(Vector2(0, 0), 22)
coin.shapeSensor = true
```

setShapeEnableHitEvents / setShapeContactEvents / setShapePreSolveEvents / setShapeSensorEvents

Enable specific collision event callbacks for a shape. See PhysicsSystem events (p. 567) and Events manual (p. 219).

- **HitEvents** — Fired when two shapes collide (one-shot, not sustained), as `hitContact2D`.
- **ContactEvents** — Fired when shapes start and stop touching, as `beginContact2D` and `endContact2D`.
- **PreSolveEvents** — Fired before the physics impulse is resolved, as `preSolve2D`; allows modifying or cancelling the collision. It runs every step for an awake body, even one resting still, so a filter that changes

its answer (drop through a platform while holding down) takes effect at once. Turning the flag on or off also works at once, on contacts that already touch and from inside a physics callback. For this Doriax turns off Box2D's contact reuse, so every touching contact is updated each step.

- **SensorEvents** — Lets the shape be reported when it overlaps a sensor shape. It does *not* turn the shape itself into a sensor, and it does not change the collision response. A sensor always has them on: `isShapeSensorEvents` returns `true` for it and the editor hides the box.

Hit, contact and pre-solve events are off by default, both in code and in the shape's **Enable Hit Events**, **Contact Events** and **PreSolve Events** boxes in the editor. Box2D reports one for a pair of shapes only when at least one of the two has it on, so subscribing to `beginContact2D` alone reports nothing until a shape turns **ContactEvents** on.

setBitsFilter

- `void setBitsFilter(uint16_t categoryBits, uint16_t maskBits)`
- `void setBitsFilter(size_t shapeIndex, uint16_t categoryBits, uint16_t maskBits)`

Configures collision filtering using bitmasks. A body collides with another only if `(a.category & b.mask) != 0` and `(b.category & a.mask) != 0`.

C++

```
const uint16_t LAYER_PLAYER   = 0x0001;
const uint16_t LAYER_ENEMY    = 0x0002;
const uint16_t LAYER_WALL     = 0x0004;

// Player collides with enemies and walls
playerBody.setBitsFilter(LAYER_PLAYER, LAYER_ENEMY | LAYER_WALL);
// Enemy collides with player and walls
enemyBody.setBitsFilter(LAYER_ENEMY, LAYER_PLAYER | LAYER_WALL);
```

setMotionLocks

- `void setMotionLocks(bool linearX, bool linearY, bool angularZ)`
- `bool isLinearXLocked() const`
- `bool isLinearYLocked() const`

Stops the body moving along world X, along world Y, or rotating. A locked axis ignores gravity, forces and contacts; the others move as usual. A character that only falls, or a piston that only slides sideways, is one lock away. The rotation lock is fixedRotation. The locks live on the loaded body, so set them after load.

Lua

```
body:load()
body:setMotionLocks(true, false, true) -- falls straight down without spinning
```

applyMassFromShapes

- `void applyMassFromShapes()`

Recomputes mass and inertia from the attached shapes' density values. Call after changing `setShapeDensity` at runtime.

applyForce

- `void applyForce(const Vector2& force, const Vector2& point, bool wake)`

Applies a force (in Newtons) at a given world-space `point`. Pass `wake = true` to activate a sleeping body.

applyForceToCenter

- `void applyForceToCenter(const Vector2& force, bool wake)`

Applies a force at the body's center of mass; produces linear acceleration but no torque.

applyTorque

- `void applyTorque(float torque, bool wake)`

Applies a rotational force. Positive values spin counter-clockwise.

applyLinearImpulse / applyLinearImpulseToCenter

Instantaneous velocity changes (Δv = impulse / mass). Prefer impulses over forces for one-shot events like jumps.

applyAngularImpulse

- `void applyAngularImpulse(float impulse, bool wake)`

Instantaneous angular velocity change.

getBodyContacts / getShapeContacts

- `std::vector<Contact2D> getBodyContacts()`
- `std::vector<Contact2D> getShapeContacts(size_t index)`

Returns currently active contact manifolds for the entire body or for a specific shape. Useful for manual overlap queries without subscribing to per-frame events. `getShapeContacts` doesn't work on chain shapes; use `getBodyContacts` for those.

getAttachedObject

- `Object getAttachedObject()`

Returns the Object (p. 455) whose entity owns this body.

getPointsToMeterScale

- `float getPointsToMeterScale() const`

Returns the pixels-to-meters conversion scale used by Box2D. The physics simulation uses SI units (meters/kg/seconds); divide pixel distances by this value before passing them to physics methods, or use the engine's automatic scaling.

Body3D

Description

`Body3D` is the 3D rigid-body handle built on top of the **Jolt Physics** library. Obtain one by calling `object.getBody3D()`. Create one or more shapes, set the body type and material properties, then call `load()` to register with the physics world. The body transform is synchronized with the parent Object (p. 455) each physics step.

Inherits: EntityHandle (p. 711)

C++

```
Mesh cube(&scene);
cube.setTexture("textures/box.png");

Body3D body = cube.getBody3D();
body.createBoxShape(1.0f, 1.0f, 1.0f);
body.setType(BodyType::DYNAMIC);
body.load();
```

Lua

```
local cube = Mesh(scene)
cube:setTexture("textures/box.png")

local body = cube:getBody3D()
body:createBoxShape(1, 1, 1)
body.type = BodyType.DYNAMIC
body:load()
```

Properties

Type	Name	Default	Languages
BodyType	type	`STATIC`	C++ | Lua
Body3DMotionQuality	motionQuality	`DISCRETE`	C++ | Lua
Vector3	position	—	C++ | Lua
Quaternion	rotation	—	C++ | Lua
float	mass	—	C++ | Lua
float	gravityFactor	`1.0`	C++ | Lua
float	buoyancy	`0.0`	C++ | Lua
float	waterDrag	`0.5`	C++ | Lua
float	waterAngularDrag	`0.01`	C++ | Lua
float	friction	`0.2`	C++ | Lua
float	restitution	`0.0`	C++ | Lua
Vector3	linearVelocity	`(0,0,0)`	C++ | Lua
Vector3	angularVelocity	`(0,0,0)`	C++ | Lua
bool	allowSleeping	`true`	C++ | Lua
bool	sensor	`false`	C++ | Lua
bool	collideKinematicVsNonDynamic	`false`	C++ | Lua
uint16_t	categoryBitsFilter	`0xFFFF`	C++ | Lua
uint16_t	maskBitsFilter	`0xFFFF`	C++ | Lua
uint32_t	collisionGroupID	`0`	C++ | Lua

Methods

Returns	Name	Languages
void	load	C++ | Lua
int	createBoxShape	C++ | Lua
int	createSphereShape	C++ | Lua
int	createCapsuleShape	C++ | Lua
int	createTaperedCapsuleShape	C++ | Lua
int	createCylinderShape	C++ | Lua
int	createConvexHullShape	C++ | Lua
int	createMeshShape	C++ | Lua
int	createHeightFieldShape	C++ | Lua
size_t	getNumShapes	C++ | Lua
Shape3DType	getShapeType	C++ | Lua
void	setShapeDensity	C++ | Lua
float	getShapeDensity	C++ | Lua
void	setShapeBitsPerSample	C++ | Lua
unsigned int	getShapeBitsPerSample	C++ | Lua
void	setShapeBuildQuality	C++ | Lua
Shape3DBuildQuality	getShapeBuildQuality	C++ | Lua
void	activate	C++ | Lua
void	deactivate	C++ | Lua
bool	canBeKinematicOrDynamic	C++ | Lua
void	setAllowedDOFsAll	C++ | Lua
void	setAllowedDOFs2DPlane	C++ | Lua
void	setAllowedDOFs	C++ | Lua
void	setMass	C++
void	setOverrideMassAndInertia	C++ | Lua
void	setBitsFilter	C++ | Lua
void	setLinearVelocity	C++
void	setLinearVelocityClamped	C++ | Lua
void	setAngularVelocity	C++
void	setAngularVelocityClamped	C++ | Lua
Vector3	getPointVelocity	C++ | Lua
Vector3	getAccumulatedForce	C++ | Lua
Vector3	getAccumulatedTorque	C++ | Lua
void	applyForce	C++ | Lua
void	applyTorque	C++ | Lua
void	applyImpulse	C++ | Lua
void	applyAngularImpulse	C++ | Lua
bool	applyBuoyancyImpulse	C++ | Lua
SubmergedVolume3D	getSubmergedVolume	C++ | Lua
Vector3	getCenterOfMassPosition	C++ | Lua
Object	getAttachedObject	C++ | Lua
Body3DComponent (p. 668)	`getBody3DComponent`	Lua

Enumerations

BodyType

- **STATIC** — Infinite mass; never moves. Walls, floors, and terrain.
- **KINEMATIC** — Controlled by velocity, not forces. Elevators and moving platforms.
- **DYNAMIC** — Fully simulated by Jolt; reacts to forces and gravity.

Body3DMotionQuality

- **DISCRETE** — Standard collision detection per time step (default). Suitable for most objects.
- **LINEAR_CAST** — Continuous collision detection: sweeps the shape along its path each step. Use for fast-moving objects that may tunnel through thin geometry.

Shape3DType

`BOX`, `SPHERE`, `CAPSULE`, `TAPERED_CAPSULE`, `CYLINDER`, `CONVEX_HULL`, `MESH`, `HEIGHTFIELD`

Shape3DBuildQuality

How a mesh shape builds its collision tree. See setShapeBuildQuality.

- **FAVOR_RUNTIME_PERFORMANCE** — Builds a better tree that answers collisions and ray casts faster (default).
- **FAVOR_BUILD_SPEED** — Builds faster, so the mesh loads sooner, but each query costs more.

Structures

SubmergedVolume3D

What getSubmergedVolume returns.

Type	Name	Description
float	`totalVolume`	Volume of the body's shapes, in m^3
float	`submergedVolume`	Volume below the surface, in m^3
float	`fraction`	`submergedVolume / totalVolume`: `0` out of the fluid, `1` all under
Vector3	`centerOfBuoyancy`	World-space centre of the part under the surface; the centre of mass when nothing is under

Property details

type

- *Setter:* `void setType(BodyType type)`
- *Getter:* `BodyType getType() const`

Simulation mode. See BodyType. Can be changed at runtime to switch between, e.g., KINEMATIC and DYNAMIC.

motionQuality

- *Setter:* `void setMotionQuality(Body3DMotionQuality motionQuality)`
- *Getter:* `Body3DMotionQuality getMotionQuality() const`

Collision detection mode. See Body3DMotionQuality.

mass

- *Setter:* `void setMass(float mass)`
- *Getter:* `float getMass() const`

The body's mass in kilograms. The setter scales the inertia by the same factor, so a body made heavier also gets harder to spin. Set it after calling load; see setMass. The getter returns the mass the body has now, including a change from `setMass` or setOverrideMassAndInertia; for a static body it returns the mass its shapes would give.

gravityFactor

- *Setter:* `void setGravityFactor(float gravityFactor)`
- *Getter:* `float getGravityFactor() const`

Scales the global gravity for this body. `0` = gravity-free, `2` = double gravity.

Stored on the body component, so it can be set **before** load and survives a body reload. In the editor it is the **Gravity Factor** field of the Body3D component, saved with the scene and applied in exported projects.

buoyancy

- *Setter:* `void setBuoyancy(float buoyancy)`
- *Getter:* `float getBuoyancy() const`
- *Setter:* `void setWaterDrag(float waterDrag)`
- *Getter:* `float getWaterDrag() const`
- *Setter:* `void setWaterAngularDrag(float waterAngularDrag)`
- *Getter:* `float getWaterAngularDrag() const`

How the body floats in a Water (p. 496). `buoyancy` compares the water's density to the body's: `1` is neutral, more floats and less sinks. `0` (the default) ignores the water. `waterDrag` is how much the water slows the body down, and `waterAngularDrag` how much it slows its rotation.

Every fixed step, a **dynamic** body with a `buoyancy` above `0` whose centre of mass is inside a water's area is pushed up by the part of its shape below the wave surface there, and tilted by the wave normal, so it bobs and rolls with the waves. No script or applyBuoyancyImpulse call is needed. Static and kinematic bodies are not affected, and waters with their own buoyancy (p. 500) off float nothing.

Like gravityFactor, the values are stored on the body component, so they can be set before load. In the editor they are the **Buoyancy** section of the Body3D component (shown for dynamic bodies). See Physics — Buoyancy (p. 319).

C++

```
Body3D body = crate.getBody3D();
body.createBoxShape(1.0f, 1.0f, 1.0f);
body.setType(BodyType::DYNAMIC);
body.setBuoyancy(1.4f);   // floats, about 70% under
body.load();
```

Lua

```
local body = crate:getBody3D()
body:createBoxShape(1, 1, 1)
body.type = BodyType.DYNAMIC
body.buoyancy = 1.4
body:load()
```

friction / restitution

- *Setters:* `void setFriction(float)` / `void setRestitution(float)`
- *Getters:* `float getFriction() const` / `float getRestitution() const`

Surface material properties shared across all shapes on this body. Friction `[0, ∞)` controls sliding resistance; restitution `[0, 1]` controls bounciness.

position / rotation

- *Setters:* `void setPosition(Vector3)` / `void setRotation(Quaternion)`
- *Getters:* `Vector3 getPosition() const` / `Quaternion getRotation() const`

The body's own pose in **world space**, read from and written straight to the simulation, and the only way to place a body from `onFixedUpdate`. Setting either also updates the entity Object (p. 455) transform so rendering follows and the next step does not undo the move. Both wake the body.

Use these for teleports — respawns, checkpoints, portals. For continuous movement prefer linearVelocity or applyForce, which let the solver resolve collisions instead of pushing bodies through walls.

> **Parented bodies**
> The world pose is converted to the entity's local space using the parent's transform as of the last variable-timestep sync. Moving a body's **parent** and teleporting the body in the same callback therefore places it relative to the parent's *previous* pose — do one or the other per frame. Bodies with no parent, the common case, are exact.

C++

```
Body3D body = object.getBody3D();

body.setPosition(Vector3(0, 10, 0));   // respawn
body.setLinearVelocity(Vector3::ZERO); // stop residual motion
```

Lua

```
local body = object:getBody3D()

body.position = Vector3(0, 10, 0)
body.linearVelocity = Vector3(0, 0, 0)
```

> **Do not use Object::setPosition from onFixedUpdate**
> Writing an entity's Object (p. 455) transform only reaches the body between fixed steps. Inside `onFixedUpdate` the physics system syncs the body from the transform *before* stepping and writes the stepped pose back *after*, so a transform written there is discarded unless you also call updateTransform (p. 460) to refresh the world transform the sync reads. Prefer `Body3D::setPosition` / `setRotation`: one call, and it writes the simulation directly. `Object::setPosition` is fine from `onUpdate` and outside the update callbacks.

linearVelocity / angularVelocity

- *Setters:* `void setLinearVelocity(Vector3)` / `void setAngularVelocity(Vector3)`
- *Getters:* `Vector3 getLinearVelocity() const` / `Vector3 getAngularVelocity() const`

Current velocity vectors. Linear velocity is in world units/second; angular velocity is in radians/second around each world axis.

sensor

- *Setter:* `void setIsSensor(bool sensor)`

- *Getter:* `bool isSensor() const`
- *Lua property:* `body.sensor`

When `true`, the body acts as a trigger volume: overlapping bodies generate contact events but no physical impulse is applied.

The flag is stored on the body component, not only on the live simulation body, so it can be set **before** `load()` and survives a body reload (a shape change, a body type change, or reopening the scene). In the editor it is the **Sensor** checkbox of the Body3D component, saved with the scene and applied in exported projects.

C++

```
Body3D trigger = checkpoint.getBody3D();
trigger.createBoxShape(2, 2, 2);
trigger.setIsSensor(true);   // before load()
trigger.load();
```

Lua

```
local trigger = checkpoint:getBody3D()
trigger:createBoxShape(2, 2, 2)
trigger.sensor = true
trigger:load()
```

> **Which bodies a sensor detects**
>
> A `STATIC` sensor is the cheapest option, but it only detects **awake** dynamic and kinematic bodies — the contact is lost as soon as the other body falls asleep. Making the sensor dynamic or kinematic (and activating it) also detects sleeping bodies, and such a sensor never sleeps on its own. Keep dynamic and kinematic sensors on a category/mask that excludes static bodies and other sensors to avoid broad-phase work.

collideKinematicVsNonDynamic

- *Setter:* `void setCollideKinematicVsNonDynamic(bool)`
- *Getter:* `bool isCollideKinematicVsNonDynamic() const`

When `true`, this kinematic body generates collision reports against static and other kinematic bodies (normally skipped for performance).

categoryBitsFilter / maskBitsFilter

Collision layer filtering bitmasks. A collision occurs only when `(a.category & b.mask) != 0`.

collisionGroupID

Group index for disabling self-collision within a group (e.g. ragdoll bones).

Method details

load

- `void load()`

Registers the body and all its shapes with the Jolt physics world. Must be called after defining shapes. Re-call if shapes change at runtime.

createBoxShape

- `int createBoxShape(float width, float height, float depth)`
- `int createBoxShape(Vector3 position, Quaternion rotation, float width, float height, float depth)` *(with local offset)*

Creates a solid box collider. Returns the shape index. The offset overload positions the shape relative to the entity origin.

createSphereShape

- `int createSphereShape(float radius)`
- `int createSphereShape(Vector3 position, Quaternion rotation, float radius)` *(with local offset)*

Creates a sphere collider. Spheres are the cheapest shape to simulate.

createCapsuleShape

- `int createCapsuleShape(float halfHeight, float radius)`
- `int createCapsuleShape(Vector3 position, Quaternion rotation, float halfHeight, float radius)`

Creates a vertical capsule (hemisphere-capped cylinder). Standard character controller shape.

C++

```
Body3D body = player.getBody3D();
body.createCapsuleShape(0.9f, 0.3f);   // 1.8 m tall, 0.3 m radius
body.setType(BodyType::DYNAMIC);
body.setAllowedDOFs2DPlane();           // constrain to XZ plane
body.load();
```

Lua

```
local body = player:getBody3D()
body:createCapsuleShape(0.9, 0.3)
body.type = BodyType.DYNAMIC
body:setAllowedDOFs2DPlane()
body:load()
```

createTaperedCapsuleShape

- `int createTaperedCapsuleShape(float halfHeight, float topRadius, float bottomRadius)`
- `int createTaperedCapsuleShape(Vector3 position, Quaternion rotation, float halfHeight, float topRadius, float bottomRadius)`

Capsule with different radii at each end. Useful for cones, bullets, and stylised characters.

createCylinderShape

- `int createCylinderShape(float halfHeight, float radius)`
- `int createCylinderShape(Vector3 position, Quaternion rotation, float halfHeight, float radius)`

Upright cylinder collider. Less stable than capsules on uneven ground; prefer capsules for characters.

createConvexHullShape

- `int createConvexHullShape()`
- `int createConvexHullShape(std::vector<Vector3> vertices)`
- `int createConvexHullShape(Vector3 position, Quaternion rotation, std::vector<Vector3> vertices)`

Creates a convex hull from the mesh vertices (no-argument form) or from explicit vertices. Must be convex; Jolt computes the hull automatically.

createMeshShape

- `int createMeshShape()`
- `int createMeshShape(std::vector<Vector3> vertices, std::vector<uint16_t> indices)`
- `int createMeshShape(Vector3 position, Quaternion rotation, std::vector<Vector3> vertices, std::vector<uint16_t> indices)`

Creates a static triangle-mesh collider. **Mesh shapes can only be STATIC.** Use convex hull for dynamic objects. For a large mesh, setShapeBuildQuality trades query speed for a faster build.

createHeightFieldShape

- `int createHeightFieldShape()`
- `int createHeightFieldShape(unsigned int samplesSize)`

Creates a height-field collider sized to the terrain mesh in the entity. Used with the Terrain (p. 486) class. `samplesSize` controls the resolution grid. setShapeBitsPerSample sets how precisely each height is stored.

activate / deactivate

- `void activate()` — Wakes the body if it is sleeping.
- `void deactivate()` — Forces the body to sleep immediately.

setAllowedDOFsAll / setAllowedDOFs2DPlane / setAllowedDOFs

Constrain which degrees of freedom the body can move in.

- **setAllowedDOFsAll** — All 6 DOF (default).
- **setAllowedDOFs2DPlane** — Restricts to translation on X/Z and rotation on Y. Use for top-down or side-view games.
- **setAllowedDOFs(translationX, translationY, translationZ, rotationX, rotationY, rotationZ)** — Fine-grained control per axis.

Locking all three rotation axes is the usual way to keep a character upright; drive its facing with rotation instead of letting the solver spin it.

Like gravityFactor, the mask is stored on the body component, so it survives a body reload and is saved with the scene and applied in exported projects. Leaving every axis locked is invalid in Jolt — use a static body to freeze one completely.

setMass / setOverrideMassAndInertia

- `void setMass(float mass)` — Sets the mass in kilograms and scales the inertia by the same factor, so the body keeps how its mass is spread: twice as heavy also takes twice the torque to spin. The mass must be above `0`, and the body must be loaded and not static, with at least one translation axis free.
- `void setOverrideMassAndInertia(Vector3 solidBoxSize, float solidBoxDensity)` — Computes mass and inertia for a solid box of the given size and density (kg/m^3), then applies those values.

`setMass` changes only the live body: a body reload computes the mass from the shapes again. To keep a mass, set it through the shape density or the solid-box override.

The solid-box override is stored on the body component, so it can be set **before** load and survives a body reload; on a loaded dynamic body it takes effect at once. In the editor it is **Override Mass** on the Body3D

component, with **Solid Box Size** and **Solid Box Density** (1000 kg/m³ by default, about the density of water), saved with the scene and applied in exported projects.

setShapeDensity

- `void setShapeDensity(float density)` *(first shape)*
- `void setShapeDensity(size_t index, float density)` *(specific shape)*

Sets the density (kg/m³) used to compute mass. Call after adding shapes and before load, or call load again after runtime changes.

setShapeBitsPerSample

- `void setShapeBitsPerSample(size_t index, unsigned int bitsPerSample)`
- `unsigned int getShapeBitsPerSample(size_t index) const`

How many bits a height-field shape stores for each height, from `1` to `16` (default `8`). The heights are kept in blocks of samples, and the bits divide each block's own height range, so 8 bits already give small steps on gentle ground. More bits make collision follow steep terrain closer and take more memory. The shape is rebuilt on the next update. In the editor it is **Bits Per Sample** on a heightfield shape.

setShapeBuildQuality

- `void setShapeBuildQuality(size_t index, Shape3DBuildQuality buildQuality)`
- `Shape3DBuildQuality getShapeBuildQuality(size_t index) const`

How a mesh shape builds its collision tree: see Shape3DBuildQuality. The default favors runtime performance, which suits level geometry built once. `FAVOR_BUILD_SPEED` makes a large mesh load sooner, for example on the web, at a cost on every collision and ray cast against it. The shape is rebuilt on the next update. In the editor it is **Build Quality** on a mesh shape.

C++

```
Body3D body = house.getBody3D();
int shape = body.createMeshShape();
body.setShapeBuildQuality(shape, Shape3DBuildQuality::FAVOR_BUILD_SPEED);
body.load();
```

Lua

```
local body = house:getBody3D()
local shape = body:createMeshShape()
body:setShapeBuildQuality(shape, Shape3DBuildQuality.FAVOR_BUILD_SPEED)
body:load()
```

setLinearVelocityClamped / setAngularVelocityClamped

Velocity setters that clamp the new value to the body's configured speed limits (set in Jolt body settings). Safer for kinematic controllers.

getPointVelocity

- `Vector3 getPointVelocity(Vector3 point) const`

Returns the velocity of a world-space point on the body, including the contribution of angular velocity. Use for precise impact and dragging calculations.

applyForce

- `void applyForce(const Vector3& force)`
- `void applyForce(const Vector3& force, const Vector3& point)`

Applies a force in Newtons each simulation step. The single-argument overload applies force at the center of mass (no torque). The two-argument overload applies force at a world-space point, potentially generating torque.

applyTorque

- `void applyTorque(const Vector3& torque)`

Applies a torque (Nm) to rotate the body. The vector direction defines the rotation axis (right-hand rule).

applyImpulse

- `void applyImpulse(const Vector3& impulse)`
- `void applyImpulse(const Vector3& impulse, const Vector3& point)`

Instantaneous velocity change. Prefer impulses over forces for one-shot events (jumps, explosions, hits).

applyAngularImpulse

- `void applyAngularImpulse(const Vector3& angularImpulse)`

Instantaneous angular velocity change. Useful for spinning objects on impact.

applyBuoyancyImpulse

- `bool applyBuoyancyImpulse(Vector3 surfacePosition, Vector3 surfaceNormal, float buoyancy, float linearDrag, float angularDrag, Vector3 fluidVelocity, Vector3 gravity, float deltaTime)`

Simulates buoyancy for a body partially submerged in a fluid. Returns `true` if the body intersects the water plane. Only a dynamic body is pushed, and a sleeping one wakes up. Call once per physics step with the water plane data. A body in a Water (p. 496) gets this every fixed step from its buoyancy settings, so call it yourself only for fluids that are not a Water component.

getSubmergedVolume

- `SubmergedVolume3D getSubmergedVolume(const Vector3& surfacePosition, const Vector3& surfaceNormal) const`

How much of the body lies below a fluid surface, given as a plane through `surfacePosition` whose `surfaceNormal` points up, out of the fluid. Returns a SubmergedVolume3D with the volumes, the submerged `fraction` and the centre of buoyancy. It only reads the body, so it applies no force: use it for gameplay, such as a splash when `fraction` first rises above `0`, or a sinking state once it nears `1`. It works for any body type, but not for a body with a mesh or height-field shape, which has no volume: that call logs an error and returns zeros.

For a Water (p. 496), take the surface from Water.getHeight and getNormal (p. 502) at the body's centre of mass, the point the built-in buoyancy uses.

C++

```
Vector3 center = body.getCenterOfMassPosition();
Vector3 surface(center.x, sea.getHeight(center.x, center.z), center.z);
SubmergedVolume3D under = body.getSubmergedVolume(surface, sea.getNormal(center.x, center.z));
if (under.fraction > 0.0f && !wasInWater){
    // splash
}
wasInWater = under.fraction > 0.0f;
```

Lua

```
local center = body:getCenterOfMassPosition()
local surface = Vector3(center.x, sea:getHeight(center.x, center.z), center.z)
local under = body:getSubmergedVolume(surface, sea:getNormal(center.x, center.z))
if under.fraction > 0 and not self.wasInWater then
    -- splash
end
self.wasInWater = under.fraction > 0
```

getAttachedObject

- `Object getAttachedObject()`

Returns the Object (p. 455) that owns this body.

getCenterOfMassPosition

- `Vector3 getCenterOfMassPosition()`

Returns the world-space position of the center of mass, which may differ from the entity origin if shapes are offset.

CollideShapeResult3D

`CollideShapeResult3D` API exposed to Lua and C++ gameplay code.

Methods

Name	Languages
`getContactPointOnA`	C++ | Lua
`getContactPointOnB`	C++ | Lua
`getPenetrationAxis`	C++ | Lua
`getShapeIndex1`	C++ | Lua
`getShapeIndex2`	C++ | Lua

Contact2D

2D contact manifold data.

Methods

Name	Languages
`getManifold`	C++ | Lua
`getBodyEntityA`	C++ | Lua
`getBodyA`	C++ | Lua
`getShapeIndexA`	C++ | Lua
`getBodyEntityB`	C++ | Lua
`getBodyB`	C++ | Lua
`getShapeIndexB`	C++ | Lua

Contact3D

3D contact data.

Properties

Name	Languages
`combinedFriction`	C++ | Lua
`combinedRestitution`	C++ | Lua
`sensor`	C++ | Lua

Methods

Name	Languages
`getBaseOffset`	C++ | Lua
`getWorldSpaceNormal`	C++ | Lua
`getPenetrationDepth`	C++ | Lua
`getShapeIndex1`	C++ | Lua
`getShapeIndex2`	C++ | Lua
`getRelativeContactPointsOnA`	C++ | Lua
`getRelativeContactPointsOnB`	C++ | Lua

Joint2D

Box2D joint constraint.

Methods

Name	Languages
`setDistanceJoint`	C++ | Lua
`setRevoluteJoint`	C++ | Lua
`setPrismaticJoint`	C++ | Lua
`setMouseJoint`	C++ | Lua
`setWheelJoint`	C++ | Lua
`setWeldJoint`	C++ | Lua
`setMotorJoint`	C++ | Lua
`getType`	C++ | Lua
`getJoint2DComponent` (Joint2DComponent (p. 673))	Lua

Method details

setDistanceJoint

- `void setDistanceJoint(Entity bodyA, Entity bodyB)`
- `void setDistanceJoint(Entity bodyA, Entity bodyB, Vector2 worldAnchorOnBodyA, Vector2 worldAnchorOnBodyB, bool rope)`

Keeps two bodies at the distance their anchors have when the joint is created. The first form anchors at each body's position, the second takes anchors in world space. With `rope` set to `true` the joint acts like a rope: the bodies can move closer, but never farther apart than that distance. This is the **Rope** option of a Distance joint in the editor.

C++

```
// the weight hangs at most 150 points below the hook
Joint2D rope(&scene);
rope.setDistanceJoint(hook.getEntity(), weight.getEntity(), Vector2(0, 200), Vector2(0, 50), true);
```

Lua

```
-- the weight hangs at most 150 points below the hook
local rope = Joint2D(scene)
rope:setDistanceJoint(hook.entity, weight.entity, Vector2(0, 200), Vector2(0, 50), true)
```

setMouseJoint

- `void setMouseJoint(Entity bodyA, Entity bodyB, Vector2 target)`

Grabs the dynamic `bodyA` at `target`, in world space, and holds that point there with a soft spring, so the body hangs from it and swings to rest. The target stays fixed in the world: `bodyB` must be another body but takes no force and doesn't carry the target, and `bodyA` still collides with it. Removing or disabling `bodyB` releases `bodyA`, and the joint comes back when `bodyB` does.

Joint3D

Jolt 3D joint constraint.

Methods

Name	Languages
`setFixedJoint`	C++ | Lua
`setDistanceJoint`	C++ | Lua
`setPointJoint`	C++ | Lua
`setHingeJoint`	C++ | Lua
`setConeJoint`	C++ | Lua
`setPrismaticJoint`	C++ | Lua
`setSwingTwistJoint`	C++ | Lua
`setSixDOFJoint`	C++ | Lua
`setPathJoint`	C++ | Lua
`setGearJoint`	C++ | Lua
`setRackAndPinionJoint`	C++ | Lua
`setPulleyJoint`	C++ | Lua
`getType`	C++ | Lua
`setMotorState` / `getMotorState`	C++ | Lua (property `motorState`)
`setMotorTargetVelocity` / `getMotorTargetVelocity`	C++ | Lua (property `motorTargetVelocity`)
`setMotorTargetPosition` / `getMotorTargetPosition`	C++ | Lua (property `motorTargetPosition`)
`setMotorFrequency` / `getMotorFrequency`	C++ | Lua (property `motorFrequency`)
`setMotorDamping` / `getMotorDamping`	C++ | Lua (property `motorDamping`)
`setMotorMaxForce` / `getMotorMaxForce`	C++ | Lua (property `motorMaxForce`)
`getCurrentPosition`	C++ | Lua
`getJoint3DComponent` (Joint3DComponent (p. 674))	Lua

Enumerations

Joint3DMotorState

- **OFF** — No motor (default). The joint moves freely within its limits. Turning a motor off wakes its bodies, so what it held falls or swings free.
- **VELOCITY** — Drives the joint at `motorTargetVelocity`, as hard as `motorMaxForce` allows.
- **POSITION** — Springs the joint to `motorTargetPosition`, with the spring set by `motorFrequency` and `motorDamping`.
- **POSITION_AND_VELOCITY** — Springs the joint to `motorTargetPosition` while it moves at `motorTargetVelocity`. Move the target a little each step at that velocity for a smooth path that does not lag behind.

Motors

A **hinge** or **prismatic** joint can drive its bodies: a door that swings shut, a lift that rises to a floor, a crank or a wheel that turns on its own. On other joint types the motor values are kept but do nothing.

Property	Default	Hinge	Prismatic
`motorState`	`OFF`	Joint3DMotorState	same
`motorTargetVelocity`	`0`	angle per second	m/s along the axis
`motorTargetPosition`	`0`	angle	m along the axis
`motorFrequency`	`2`	Hz	Hz
`motorDamping`	`1`	ratio	ratio
`motorMaxForce`	`0`	torque limit, N·m	force limit, N

- Angles are in degrees, or radians when `Engine::useDegrees` is `false`. A hinge's angle `0` is the pose its bodies have when the joint is made, and a body reload during play keeps it.
- A prismatic position is measured in the same frame as its `limitsMin` and `limitsMax`, and Jolt keeps the target inside them.
- `motorFrequency` is how fast a position motor springs back, and `motorDamping` how much it resists overshooting: `1` stops at the target without bouncing, less bounces. A frequency of `0` makes the spring as stiff as the solver allows.
- `motorMaxForce` `0` means no limit. With a limit, a heavy door or lift may stop short of its target or move slower than asked, which is often what a mechanism should do. A limited position motor can also swing past its target before it settles, whatever its damping.

The setters change the live joint at once and wake its bodies; one called from a 3D callback takes effect before the next step. The values are also stored on the Joint3DComponent (p. 674), so they survive the joint being rebuilt and are saved with the scene. In the editor they are the **Motor** section of a hinge or prismatic joint, and changing them during Play moves the running joint without rebuilding it.

`getCurrentPosition()` returns the hinge angle or prismatic position now, in the same units, and `0` while the joint is not built.

C++

```
// a door that swings open to 90 degrees
Joint3D hinge(&scene);
hinge.setHingeJoint(frame.getEntity(), door.getEntity(), hingePoint, Vector3(0, 1, 0), Vector3(1, 0,
0));
hinge.setMotorState(Joint3DMotorState::POSITION);
hinge.setMotorMaxForce(200.0f);  // N m: a player can still hold it
hinge.setMotorTargetPosition(90.0f);

// a lift that rises at 1 m/s
Joint3D lift(&scene);
lift.setPrismaticJoint(shaft.getEntity(), cabin.getEntity(), Vector3(0, 1, 0), 0.0f, 12.0f);
lift.setMotorState(Joint3DMotorState::VELOCITY);
lift.setMotorTargetVelocity(1.0f);

// later, in onFixedUpdate: stop at the top
if (lift.getCurrentPosition() >= 12.0f){
    lift.setMotorState(Joint3DMotorState::OFF);
}
```

Lua

```lua
local hinge = Joint3D(scene)
hinge:setHingeJoint(frame.entity, door.entity, hingePoint, Vector3(0, 1, 0), Vector3(1, 0, 0))
hinge.motorState = Joint3DMotorState.POSITION
hinge.motorMaxForce = 200
hinge.motorTargetPosition = 90

local lift = Joint3D(scene)
lift:setPrismaticJoint(shaft.entity, cabin.entity, Vector3(0, 1, 0), 0, 12)
lift.motorState = Joint3DMotorState.VELOCITY
lift.motorTargetVelocity = 1

-- later, in onFixedUpdate: stop at the top
if lift:getCurrentPosition() >= 12 then
    lift.motorState = Joint3DMotorState.OFF
end
```

Manifold2D

`Manifold2D` API exposed to Lua and C++ gameplay code.

Methods

Name	Languages
`getManifoldPointAnchorA`	C++ | Lua
`getManifoldPointAnchorB`	C++ | Lua
`getManifoldPointPosition`	C++ | Lua
`getManifoldPointNormalImpulse`	C++ | Lua
`getManifoldPointNormalVelocity`	C++ | Lua
`getManifoldPointTangentImpulse`	C++ | Lua
`getManifoldPointSeparation`	C++ | Lua
`isManifoldPointPersisted`	C++ | Lua
`getNormal`	C++ | Lua
`getPointCount`	C++ | Lua

Positions, anchors and separation are in points and the normal velocity in points per second, like Body2D (p. 544). Impulses stay in Box2D units (N·s). The normal is a unit vector from shape A to shape B. Anchors start from each body's origin in `preSolve2D` and from its center of mass in a contact from `getBodyContacts`; `getManifoldPointPosition` is the world point either way. A contact keeps the position it had when it was read, even if its body moves or is removed afterwards. A point index from `0` to `getPointCount() - 1` is valid; another one logs an error and returns zero.

PhysicsSystem

Inherits: SubSystem (p. 594)

2D (Box2D) and 3D (Jolt) simulation. `scene:getPhysicsSystem()`.

Properties

Type	Name	Default	Languages
Vector2	gravity2D	`(0,-9.81)`	C++ | Lua
Vector3	gravity3D	`(0,-9.81,0)`	C++ | Lua
Vector3	gravity	`(0,-9.81,0)`	C++ | Lua
float	pointsToMeterScale2D	`64.0`	C++ | Lua
bool	lock3DBodies	`true`	C++ | Lua
bool	interpolation	`true`	C++ | Lua
bool	steppingWorld2D	`false`	C++ | Lua *(read-only)*
bool	steppingWorld3D	`false`	C++ | Lua *(read-only)*

`gravity2D` and `gravity3D` set each world's gravity independently (m/s²); the legacy `gravity` / `setGravity(...)` sets both worlds at once and reads back the 3D value. The same values are exposed directly on Scene (p. 396) — `scene.gravity2D` / `scene.gravity3D` — which is what the editor's scene **Physics** section edits and saves.

Lua

```lua
local physics = scene:getPhysicsSystem()
physics.gravity2D = Vector2(0, -20)
physics.gravity3D = Vector3(0, -3.7, 0)
```

`lock3DBodies` decides whether 3D body access takes Jolt's body locks. Leave it on unless you drive the world from a worker thread. It is ignored while `steppingWorld3D` is true, since the step already holds those locks.

`interpolation` draws each 2D and 3D body between its poses at the last two fixed steps, so bodies move smoothly at frame rates above the step rate. The entity's transform then trails the body by less than one step: read the body pose for gameplay and the transform for what follows the body on screen. Off, the transform holds the body pose of the last step. See Smooth motion between steps (p. 321).

`steppingWorld2D` is true while the 2D world simulates, which is when `preSolve2D` and `shouldCollide2D` run. `steppingWorld3D` is true while the 3D world simulates, which is exactly when the 3D callbacks below run.

Methods

Returns	Name	Languages
void	explode2D	C++ | Lua

explode2D

- `void explode2D(Vector2 position, float radius, float falloff, float impulsePerLength)`
- `void explode2D(Vector2 position, float radius, float falloff, float impulsePerLength, uint16_t maskBits)`

Pushes the 2D dynamic bodies around `position` away from it. Shapes within `radius` take the full impulse, which fades to zero over `falloff` beyond it; both are in points. Each shape is pushed by `impulsePerLength` for every metre of its outline facing the explosion, so big crates fly less than small debris. A negative value

pulls them in. `maskBits` picks the shape categories it reaches, all by default. Circles, capsules and polygons are pushed; segments and chains are not. It is refused while the world is stepping.

Lua

```lua
function Bomb:explode()
    local position = Object(self.scene, self.entity).position
    self.scene:getPhysicsSystem():explode2D(Vector2(position.x, position.y), 120, 80, 2)
end
```

Events

2D: `beginContact2D`, `endContact2D`, `hitContact2D`, sensor contacts, `preSolve2D`, `shouldCollide2D`

3D: `onContactAdded3D`, `onContactPersisted3D`, `onContactRemoved3D`, `onBodyActivated3D`, `onBodyDeactivated3D`, `shouldCollide3D`

Lua

```lua
RegisterEvent(self, physics.beginContact2D, "onHit")
```

See Events (p. 215). Use `onFixedUpdate` for forces.

`clearSubscriptions()` removes every callback from these events, and `removeSubscriptionsByTag(substring)` removes the ones whose tag contains `substring`.

A filter callback must return a value

`preSolve2D`, `shouldCollide2D`, and `shouldCollide3D` are the events whose return value is read. A Lua handler that falls off the end returns `nil`, which is read as `false` and **rejects the contact**. The `true` default only applies when nothing is subscribed at all, so always end the handler with an explicit `return`.

```lua
function Trap:init()
    RegisterEvent(self, self.scene:getPhysicsSystem().shouldCollide3D, "shouldCollide")
end

function Trap:shouldCollide(bodyA, bodyB, offset, result)
    if bodyA.entity == self.ghostEntity then
        return false
    end
    return true          -- without this, nothing collides
end
```

Inside a 3D callback

3D events fire while Jolt is stepping, on a thread that already holds the body locks:

- Reading and moving bodies is fine — velocity, position, ray casts. The engine uses Jolt's lock-free interfaces for the whole step.
- Creating a body or joint is **refused** and logged; the next update retries it.
- Destroying one is **queued** and applied when the step ends, so removing a `Body3DComponent` from a callback tears its body down properly.

Anything else belongs in `onUpdate` or `onFixedUpdate`. Check `steppingWorld3D` when a helper can be called from both places.

Actions

Action

Inherits: EntityHandle (p. 711)
C++ type: `Action`

Description

Base class for all time-driven actions in Doriax. Actions are managed by ActionSystem (p. 590), which advances them each frame. You typically do not instantiate `Action` directly; instead you use concrete subclasses:

- TimedAction (p. 588) — a single tween with start/end values and duration
- Animation (p. 573) — a sequenced timeline of `ActionFrame` entries
- Particles (p. 724) — a particle emitter driven by the action clock

An action operates on a *target* object. When started, the action applies its effect to the target each frame until it completes, is paused, or is stopped.

Properties

Type	Name	Default	Langs
float	speed	`1.0`	C++ | Lua
float	weight	`1.0`	C++ | Lua
float	startOffset	`0.0`	C++ | Lua
bool	randomStart	`false`	C++ | Lua
bool	ownedTarget	`false`	C++ | Lua

Methods

Type	Name	Langs
void	start	C++ | Lua
void	pause	C++ | Lua
void	stop	C++ | Lua
void	restart	C++ | Lua
void	setTarget	C++ | Lua
Entity	getTarget	C++
bool	isRunning	C++ | Lua
bool	isPaused	C++ | Lua
bool	isStopped	C++ | Lua
float	getTimeCount	C++ | Lua

Property details

speed

- *Setter*: void **setSpeed**(float speed)
- *Getter*: float **getSpeed**() const

Playback speed multiplier. `1.0` is normal speed, `2.0` plays twice as fast, `0.5` in slow motion. Negative values play the action in reverse (where supported by the subclass).

weight

- *Setter*: void **setWeight**(float weight)
- *Getter*: float **getWeight**() const

Blend weight (`0.0`–`1.0`) for this action's contribution when several actions drive the same target. When multiple skeletal clips animate the same bone, `ActionSystem` combines their sampled poses as a weight-normalized average rather than letting the last one win, which is what makes crossfades smooth.

You rarely set this by hand. For Animation (p. 573) clips it is driven automatically by fadeIn (p. 574) / fadeOut (p. 574) and by Model::playAnimation() (p. 452); the manual control is Animation::blendWeight (p. 574) on the clip. Set it directly only for advanced layering of individual actions.

startOffset / randomStart

- *Setter*: void **setStartOffset**(float startOffset)
- *Getter*: float **getStartOffset**() const
- *Setter*: void **setRandomStart**(bool randomStart)
- *Getter*: bool **isRandomStart**() const

Where the action starts, in seconds: `startOffset` starts it that far in, and `randomStart` at a random time of its duration instead, so copies of the same animation do not move in sync. Both apply when the action starts from stopped, not when it resumes from a pause, also for an action that an Animation (p. 573) plays. The editor preview always starts at the beginning.

ownedTarget

- *Setter*: void **setOwnedTarget**(bool ownedTarget)
- *Getter*: bool **getOwnedTarget**() const

When `true`, the action takes ownership of its target entity — the target is automatically destroyed when the action is destroyed. Useful for fire-and-forget effects where both the action and the visual entity should be cleaned up together.

Method details

start / pause / stop

- void **start**()
- void **pause**()
- void **stop**()

Control playback:

- `start()` — begins or resumes the action. If the action was stopped, it restarts from the beginning.
- `pause()` — freezes time at the current position; `start()` resumes from here.
- `stop()` — halts and resets the time counter to zero.

The calls take effect on the next update. `start()` does nothing to a running action, so `stop()` followed by `start()` in the same frame leaves it running where it was; use restart to play it again from the beginning.

C++

```
AlphaAction fadeOut(&scene);
fadeOut.setTarget(&mySprite);
fadeOut.setAction(1.0f, 0.0f, 1.0f);
fadeOut.start();
```

Lua

```
local fadeOut = AlphaAction(scene)
fadeOut:setTarget(mySprite)
fadeOut:setAction(1.0, 0.0, 1.0)
fadeOut:start()
```

restart

- void **restart**()

Plays the action again from the beginning: a running or paused action is stopped and started again in the same update (its `onStop` and `onStart` events both fire), and a stopped one just starts.

setTarget

- void **setTarget**(Object* target)
- void **setTarget**(Entity target)
- Entity **getTarget**() const

Assigns the Object (p. 455) or entity that this action modifies. Most actions require a target to be set before `start()` is called.

isRunning / isPaused / isStopped

- bool **isRunning**() const
- bool **isPaused**() const
- bool **isStopped**() const

Query the current playback state. An action is *running* if it has been started and has not yet completed, been paused, or been stopped.

getTimeCount

- float **getTimeCount**() const

Returns the action's internal time counter in seconds, from `0` at the start to the action's total duration at completion.

AlphaAction

Inherits: TimedAction (p. 588)
C++ type: `AlphaAction`

Description

Animates the `alpha` (opacity) property of its target object from a start value to an end value over a fixed duration.

Constructors

Available in C++ and Lua:

- `AlphaAction(Scene* scene)` — creates a new entity.
- `AlphaAction(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `AlphaAction(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Type	Name	Langs
void	setAction	C++ | Lua
AlphaActionComponent (p. 666)	`getAlphaActionComponent`	Lua

Method details

setAction

- void **setAction**(float startAlpha, float endAlpha, float duration, bool loop = false)

Configures the tween. `startAlpha` and `endAlpha` are opacity values in `[0, 1]`. `duration` is in **seconds**.

C++

```cpp
AlphaAction fadeOut(&scene);
fadeOut.setTarget(&sprite);
fadeOut.setAction(1.0f, 0.0f, 0.5f);  // fade out in 0.5 s
fadeOut.start();
```

Lua

```lua
local fadeOut = AlphaAction(scene)
fadeOut:setTarget(sprite)
fadeOut:setAction(1.0, 0.0, 0.5)
fadeOut:start()
```

Set `loop = true` to create a continuous pulsing effect.

Animation

Inherits: Action (p. 569)
C++ type: `Animation`

Description

A timeline-based animation composed of `ActionFrame` entries. Each frame schedules an Action (p. 569) to run against a target entity at a specific time offset, for a specific duration. Frames can overlap to create parallel tweens, producing complex multi-property animations.

`Animation` objects are automatically created when loading a Model (p. 450) from a GLTF file. You can also build animations entirely in code for procedural sequences.

Properties

Type	Name	Default	Langs
bool	loop	`false`	C++ | Lua
bool	ownedActions	`false`	C++ | Lua
std::string	name	`""`	C++ | Lua
float	duration	`0.0`	C++
float	blendWeight	`1.0`	C++ | Lua
float	defaultFadeTime	`0.15`	C++ | Lua

Methods

Type	Name	Langs
void	fadeIn	C++ | Lua
void	fadeOut	C++ | Lua
void	addActionFrame	C++ | Lua
size_t	getActionFrameSize	C++ | Lua
ActionFrame (p. 708)	getActionFrame	C++ | Lua
void	setActionFrameStartTime	C++ | Lua
void	setActionFrameDuration	C++ | Lua
void	setActionFrameEntity	C++ | Lua
void	clearActionFrames	C++
AnimationComponent (p. 667)	`getAnimationComponent`	Lua

Property details

loop

- *Setter*: void **setLoop**(bool loop)
- *Getter*: bool **isLoop**() const

When `true`, the animation restarts from the beginning after playing to completion.

ownedActions

- *Setter*: void **setOwnedActions**(bool ownedActions)
- *Getter*: bool **isOwnedActions**() const

When `true`, all child Action (p. 569) entities referenced by the animation's frames are destroyed when the animation is destroyed. Useful for animations built entirely in code.

name

Inherited from: EntityHandle (p. 711)

- *Setter*: void **setName**(const std::string& name)
- *Getter*: std::string **getName**() const

The animation entity's human-readable name (for example `"Walk"`, `"Attack"`, or `"Idle"`). The editor displays this name in the Structure panel and animation selectors, and Model::findAnimation() (p. 452) and the string overloads of Model::playAnimation() (p. 452) use it for lookup.

Imported GLTF clips initialize their animation entity name from the source clip name. Renaming the entity also changes the string used to find or play that clip. There is no separate name stored in `AnimationComponent`.

duration

- *Setter*: void **setDuration**(const float& duration)
- *Getter*: const float& **getDuration**() const

Total length of the animation in **seconds**. When loading a model, this is set automatically from the GLTF data. The getter and setter are not exposed to Lua; Lua-created sequences derive their duration from their action frames.

blendWeight

- *Setter*: void **setBlendWeight**(float weight)
- *Getter*: float **getBlendWeight**() const

The clip's current blend weight (`0.0`–`1.0`). When several clips animate the same skeleton, each bone's final pose is the weight-normalized average of the running clips, so weights let you layer or crossfade animations smoothly.

Setting `blendWeight` cancels any in-progress fade and holds the weight at the given value. For time-based transitions prefer fadeIn / fadeOut, which animate this value for you. Reading it returns the live weight while a fade is running.

defaultFadeTime

- *Setter*: void **setDefaultFadeTime**(float seconds)
- *Getter*: float **getDefaultFadeTime**() const

The crossfade duration (in **seconds**) used by Model::playAnimation() (p. 452) when it is called without an explicit fade time. Authored per-clip and saved with the scene — set it in the **Properties** window (**AnimationComponent → Fade time**) or in code. Defaults to `0.15`.

Method details

fadeIn / fadeOut

- void **fadeIn**(float duration)
- void **fadeOut**(float duration)

Crossfade primitives that ramp the clip's blendWeight over `duration` seconds.

- `fadeIn(duration)` starts the clip (if not already running) and ramps its weight from `0` up to `1`. A `duration` of `0` starts it instantly at full weight.
- `fadeOut(duration)` ramps a running clip's weight down to `0` and stops it when it reaches zero. A `duration` of `0` stops it immediately.

To transition between two clips, fade one out while fading the other in. For clips on the same Model (p. 450), Model::playAnimation() (p. 452) does this for you.

C++

```
// Manual crossfade from run to idle over 0.25s
runAnim.fadeOut(0.25f);
idleAnim.fadeIn(0.25f);
```

Lua

```
-- Manual crossfade from run to idle over 0.25s
runAnim:fadeOut(0.25)
idleAnim:fadeIn(0.25)
```

addActionFrame

Several overloads are available:

- void **addActionFrame**(float startTime, float duration, Entity action, Entity target)
- void **addActionFrame**(float startTime, Entity timedAction, Entity target)
- void **addActionFrame**(float startTime, float duration, Entity action)
- void **addActionFrame**(float startTime, Entity timedAction)

Adds a frame to the animation timeline. `startTime` is the offset in seconds from the animation start. `duration` overrides the action's own duration; a `duration` of `0` (or lower) means **auto** — the frame follows the action's own duration. The overloads without a `duration` parameter use auto. When `target` is omitted, the animation's own target is used.

An animation cannot contain itself, directly or through nested animations: a call that would create such a cycle is rejected with an error log.

C++

```
// Sequence: move right, then move up
PositionAction moveRight(&scene);
moveRight.setAction(Vector3(0,0,0), Vector3(5,0,0), 1.0f);

PositionAction moveUp(&scene);
moveUp.setAction(Vector3(5,0,0), Vector3(5,5,0), 1.0f);

Animation seq(&scene);
seq.setDuration(2.0f);
seq.addActionFrame(0.0f, moveRight.getEntity(), mySprite.getEntity());
seq.addActionFrame(1.0f, moveUp.getEntity(), mySprite.getEntity());
seq.start();
```

Lua

```lua
local moveRight = PositionAction(scene)
moveRight:setAction(Vector3(0,0,0), Vector3(5,0,0), 1.0)

local moveUp = PositionAction(scene)
moveUp:setAction(Vector3(5,0,0), Vector3(5,5,0), 1.0)

local seq = Animation(scene)
seq:addActionFrame(0.0, moveRight.entity, mySprite.entity)
seq:addActionFrame(1.0, moveUp.entity, mySprite.entity)
seq:start()
```

getActionFrameSize

- size_t **getActionFrameSize**() const

Returns the number of frames currently in the timeline.

getActionFrame

- ActionFrame (p. 708)& **getActionFrame**(unsigned int index)

Returns a reference to the ActionFrame (p. 708) at the given zero-based index.

setActionFrameStartTime / setActionFrameDuration

- void **setActionFrameStartTime**(unsigned int index, float startTime)
- void **setActionFrameDuration**(unsigned int index, float duration)

Modify the start time or duration of an existing frame by index. Setting `duration` to `0` (or lower) switches the frame to **auto**: it follows the action's own duration.

setActionFrameEntity

- void **setActionFrameEntity**(unsigned int index, Entity action)

Replace the action entity for a given frame.

clearActionFrames

- void **clearActionFrames**()

Removes all frames from the timeline. This method is not exposed to Lua.

ColorAction

Inherits: TimedAction (p. 588)
C++ type: `ColorAction`

Description

Animates the `color` property of its target object, interpolating from a start colour to an end colour over a fixed duration. Both RGB (`Vector3`) and RGBA (`Vector4`) variants are available.

Constructors

Available in C++ and Lua:

- `ColorAction(Scene* scene)` — creates a new entity.
- `ColorAction(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `ColorAction(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Type	Name	Langs
void	setAction	C++ | Lua
ColorActionComponent (p. 671)	`getColorActionComponent`	Lua

Method details

setAction

- void **setAction**(Vector3 startColor, Vector3 endColor, float duration, bool loop = false)
- void **setAction**(Vector4 startColor, Vector4 endColor, float duration, bool loop = false)

Configures the tween. The `Vector3` overload interpolates RGB; the `Vector4` overload includes alpha. `duration` is in **seconds**.

C++

```cpp
ColorAction flash(&scene);
flash.setTarget(&sprite);
// Flash from white to original colour in 0.2 s, looping
flash.setAction(Vector3(1, 1, 1), Vector3(1, 0.3f, 0.3f), 0.2f, true);
flash.setFunctionType(EaseType::SINE_IN_OUT);
flash.start();
```

Lua

```lua
local flash = ColorAction(scene)
flash:setTarget(sprite)
flash:setAction(Vector3(1, 1, 1), Vector3(1, 0.3, 0.3), 0.2, true)
flash:setFunctionType(EaseType.SINE_IN_OUT)
flash:start()
```

KeyframeTracks

Inherits: Action (p. 569)
C++ type: `KeyframeTracks`

Description

Base class of the keyframe track actions: TranslateTracks (p. 590), RotateTracks (p. 581), ScaleTracks (p. 585) and MorphTracks (p. 579). It holds what they share: the key times, the per-segment easing, looping and relative values. Each subclass adds `setValues` with one value per key.

You typically do not instantiate `KeyframeTracks` directly.

Properties

Type	Name	Default	Langs
bool	loop	`false`	C++ | Lua
bool	relative	`false`	C++ | Lua

Methods

Type	Name	Langs
void	setTimes	C++ | Lua
void	setEasings	C++ | Lua
void	setEasing	C++ | Lua
KeyframeTracksComponent (p. 675)	`getKeyframeTracksComponent`	Lua

Property details

loop

- *Setter*: void **setLoop**(bool loop)
- *Getter*: bool **isLoop**() const

When `true`, the track starts over after its last key instead of stopping, so a track loops on its own without an Animation (p. 573).

relative

- *Setter*: void **setRelative**(bool relative)
- *Getter*: bool **isRelative**() const

When `true`, the values are offsets from the pose the target has when the track starts: a translation adds to its position, a rotation is applied after its rotation and a scale multiplies its scale. The track then works wherever the target is placed. A UI element with anchors moves by its layout offset. Morph tracks ignore it. See Relative tracks (p. 304).

Method details

setTimes

- void **setTimes**(std::vector<float> times)

Sets the key times in seconds, in ascending order. Trims the easing list when the key count shrinks.

setEasings / setEasing

- void **setEasings**(std::vector<EaseType> easings)
- void **setEasing**(unsigned int segment, EaseType ease)

`setEasing(segment, ease)` sets the easing of a single segment (key `segment` to key `segment + 1`); `setEasings(list)` replaces the whole per-segment list. Missing entries mean linear, `STEP` holds each key's value until the next key, and `CUSTOM` is not storable per segment (treated as linear). See Per-segment easing (p. 304).

MorphTracks

Inherits: KeyframeTracks (p. 578)
C++ type: `MorphTracks`

`MorphTracks` API exposed to Lua and C++ gameplay code.

Constructors

Available in C++ and Lua:

- `MorphTracks(Scene* scene)` — creates a new entity.
- `MorphTracks(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.
- `MorphTracks(Scene* scene, std::vector<float> times, std::vector<std::vector<float>> values)` — creates a track initialized with key times and values.

In Lua, use `MorphTracks(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Name	Languages
`setValues`	C++ | Lua
`getMorphTracksComponent` (MorphTracksComponent (p. 681))	Lua

The key times, easing, `loop` and `relative` come from KeyframeTracks (p. 578). Tracks imported from GLTF `CUBICSPLINE` clips carry per-key Hermite tangents and interpolate from them instead; `setValues` keeps existing tangent arrays sized to the new values. See Interpolation modes (p. 302).

PositionAction

Inherits: TimedAction (p. 588)
C++ type: `PositionAction`

Description

Animates the `position` property of its target object, moving it from a start position to an end position over a fixed duration. Works in both 2D and 3D scenes.

Constructors

Available in C++ and Lua:

- `PositionAction(Scene* scene)` — creates a new entity.
- `PositionAction(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `PositionAction(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Type	Name	Langs
void	setAction	C++ | Lua
PositionActionComponent (p. 685)	`getPositionActionComponent`	Lua

Method details

setAction

- void **setAction**(Vector3 startPosition, Vector3 endPosition, float duration, bool loop = false)

Configures the tween. `startPosition` and `endPosition` are world-space coordinates. `duration` is in **seconds**.

C++

```cpp
PositionAction move(&scene);
move.setTarget(&platform);
move.setAction(Vector3(0, 0, 0), Vector3(5, 0, 0), 2.0f, true);
move.setFunctionType(EaseType::SINE_IN_OUT);
move.start();
```

Lua

```lua
local move = PositionAction(scene)
move:setTarget(platform)
move:setAction(Vector3(0, 0, 0), Vector3(5, 0, 0), 2.0, true)
move:setFunctionType(EaseType.SINE_IN_OUT)
move:start()
```

RotateTracks

Inherits: KeyframeTracks (p. 578)
C++ type: `RotateTracks`

`RotateTracks` API exposed to Lua and C++ gameplay code.

Constructors

Available in C++ and Lua:

- `RotateTracks(Scene* scene)` — creates a new entity.
- `RotateTracks(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.
- `RotateTracks(Scene* scene, std::vector<float> times, std::vector<Quaternion> values)` — creates a track initialized with key times and values.

In Lua, use `RotateTracks(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Name	Languages
`setValues`	C++ | Lua
`getRotateTracksComponent` (RotateTracksComponent (p. 686))	Lua

The key times, easing, `loop` and `relative` come from KeyframeTracks (p. 578). Tracks imported from GLTF `CUBICSPLINE` clips carry per-key Hermite tangents and interpolate from them instead; `setValues` keeps existing tangent arrays sized to the new values. See Interpolation modes (p. 302).

RotationAction

Inherits: TimedAction (p. 588)
C++ type: `RotationAction`

Description

Animates the `rotation` property of its target object using quaternion spherical linear interpolation (slerp) between a start and end rotation. A slerp takes the short way between two rotations, so it never turns more than half a turn; setSpinAction turns by any angle instead, for full spins.

Constructors

Available in C++ and Lua:

- `RotationAction(Scene* scene)` — creates a new entity.
- `RotationAction(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `RotationAction(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Type	Name	Langs
void	setAction	C++ | Lua
void	setSpinAction	C++ | Lua
RotationActionComponent (p. 687)	`getRotationActionComponent`	Lua

Method details

setAction

- void **setAction**(Quaternion startRotation, Quaternion endRotation, float duration, bool loop = false)

Configures the tween. `startRotation` and `endRotation` are quaternion values. `duration` is in **seconds**.

C++

```cpp
RotationAction spin(&scene);
spin.setTarget(&wheel);
spin.setAction(
    Quaternion(0.0f, 0.0f, 0.0f),
    Quaternion(0.0f, 360.0f, 0.0f),
    2.0f, true
);
spin.start();
```

Lua

```lua
local spin = RotationAction(scene)
spin:setTarget(wheel)
spin:setAction(
    Quaternion(0, 0, 0),
    Quaternion(0, 360, 0),
    2.0, true
)
spin:start()
```

setSpinAction

- void **setSpinAction**(Quaternion startRotation, Vector3 axis, float angle, float duration, bool loop = false)

Turns `startRotation` by `angle` around `axis` over `duration` seconds. The angle can be any size, so `360` is a full turn and `720` two turns, and the axis is in the frame of `startRotation`. The angle uses the engine's angle unit (degrees by default). In the editor, check **Spin** in the RotationAction component and set its **Axis** and **Angle**.

C++

```cpp
RotationAction spin(&scene);
spin.setTarget(&coin);
spin.setSpinAction(Quaternion(), Vector3(0, 1, 0), 360.0f, 2.0f, true);
spin.start();
```

Lua

```lua
local spin = RotationAction(scene)
spin:setTarget(coin)
spin:setSpinAction(Quaternion(), Vector3(0, 1, 0), 360, 2.0, true)
spin:start()
```

ScaleAction

Inherits: TimedAction (p. 588)
C++ type: `ScaleAction`

Description

Animates the `scale` property of its target object, interpolating between a start scale and an end scale over a fixed duration.

Constructors

Available in C++ and Lua:

- `ScaleAction(Scene* scene)` — creates a new entity.
- `ScaleAction(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `ScaleAction(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Type	Name	Langs
void	setAction	C++ | Lua
ScaleActionComponent (p. 687)	`getScaleActionComponent`	Lua

Method details

setAction

- void **setAction**(Vector3 startScale, Vector3 endScale, float duration, bool loop = false)

Configures the tween. `startScale` and `endScale` are scale vectors. `duration` is in **seconds**.

C++

```cpp
ScaleAction pop(&scene);
pop.setTarget(&coin);
pop.setAction(Vector3(0, 0, 0), Vector3(1, 1, 1), 0.3f);
pop.setFunctionType(EaseType::BACK_OUT);
pop.start();
```

Lua

```lua
local pop = ScaleAction(scene)
pop:setTarget(coin)
pop:setAction(Vector3(0, 0, 0), Vector3(1, 1, 1), 0.3, false)
pop:setFunctionType(EaseType.BACK_OUT)
pop:start()
```

ScaleTracks

Inherits: KeyframeTracks (p. 578)
C++ type: `ScaleTracks`

`ScaleTracks` API exposed to Lua and C++ gameplay code.

Constructors

Available in C++ and Lua:

- `ScaleTracks(Scene* scene)` — creates a new entity.
- `ScaleTracks(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.
- `ScaleTracks(Scene* scene, std::vector<float> times, std::vector<Vector3> values)` — creates a track initialized with key times and values.

In Lua, use `ScaleTracks(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Name	Languages
`setValues`	C++ | Lua
`getScaleTracksComponent` (ScaleTracksComponent (p. 687))	Lua

The key times, easing, `loop` and `relative` come from KeyframeTracks (p. 578). Tracks imported from GLTF `CUBICSPLINE` clips carry per-key Hermite tangents and interpolate from them instead; `setValues` keeps existing tangent arrays sized to the new values. See Interpolation modes (p. 302).

SpriteAnimation

Inherits: Action (p. 569)
C++ type: `SpriteAnimation`

Description

Cycles through a sequence of texture atlas frames on its target Sprite (p. 480), creating flip-book-style 2D animation. This is distinct from skeletal animation (Animation (p. 573)); `SpriteAnimation` only advances the active frame of a sprite sheet.

Constructors

Available in C++ and Lua:

- `SpriteAnimation(Scene* scene)` — creates a new entity.
- `SpriteAnimation(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `SpriteAnimation(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Properties

Type	Name	Default	Langs
std::string	name	`""`	C++ | Lua

Methods

Type	Name	Langs
void	setAnimation	C++ | Lua
SpriteAnimationComponent (p. 690)	`getSpriteAnimationComponent`	Lua

Property details

name

Inherited from: EntityHandle (p. 711)

- *Setter*: void **setName**(const std::string& name)
- *Getter*: std::string **getName**() const

The sprite animation entity's human-readable name. The editor shows this name in the Structure panel. There is no separate name stored in `SpriteAnimationComponent` (p. 690) — rename the entity (or call `setName`) when you need a label for lookup with `Scene::findEntity` (p. 379).

Method details

setAnimation

- void **setAnimation**(std::vector<int> frames, std::vector<int> framesTime, bool loop)
- void **setAnimation**(int startFrame, int endFrame, int interval, bool loop)

Configure the frame sequence.

First overload — explicit frame list with per-frame durations in milliseconds:

C++

```
SpriteAnimation anim(&scene);
anim.setTarget(&hero);
// Show frames 0, 1, 2, 1 with 100 ms each
anim.setAnimation({0, 1, 2, 1}, {100, 100, 100, 100}, true);
anim.start();
```

Lua

```
local anim = SpriteAnimation(scene)
anim:setTarget(hero)
anim:setAnimation({0, 1, 2, 1}, {100, 100, 100, 100}, true)
anim:start()
```

Second overload — range-based with a fixed interval (milliseconds per frame):

C++

```
SpriteAnimation walk(&scene);
walk.setTarget(&hero);
walk.setAnimation(4, 7, 120, true);  // frames 4-7, 120 ms each
walk.start();
```

Lua

```
local walk = SpriteAnimation(scene)
walk:setTarget(hero)
walk:setAnimation(4, 7, 120, true)
walk:start()
```

The range includes both ends and counts from 0, so `setAnimation(0, 0, 100, false)` shows frame 0 alone. A negative start, or an end before the start, logs an error and keeps the current animation.

Without `loop`, the animation stops when it reaches its last frame and leaves that frame showing. A one-frame animation stops after its interval. A frame time of 0 moves on one frame per update.

See also

- SpriteAnimationComponent (p. 690)
- Sprite (p. 480)
- Action (p. 569)

TimedAction

Inherits: Action (p. 569)
C++ type: `TimedAction`

Description

Base class for all fixed-duration tweens. `TimedAction` drives a single value from `0.0` to `1.0` over a specified number of seconds, optionally with an easing curve. Concrete subclasses (AlphaAction (p. 572), ColorAction (p. 577), PositionAction (p. 580), RotationAction (p. 582), ScaleAction (p. 584)) apply that interpolated value to a specific property of the target object.

You typically do not instantiate `TimedAction` directly.

Constructors

Available in C++ and Lua:

- `TimedAction(Scene* scene)` — creates a new entity.
- `TimedAction(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `TimedAction(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Properties

Type	Name	Default	Langs
float	duration	`1.0`	C++ | Lua
bool	loop	`false`	C++ | Lua
bool	pingPong	`false`	C++ | Lua

Methods

Type	Name	Langs
void	setFunctionType	C++ | Lua
float	getValue	C++ | Lua
float	getTime	C++ | Lua
void	setDuration	C++
float	getDuration	C++
void	setLoop	C++
bool	isLoop	C++
void	setPingPong	C++
bool	isPingPong	C++

Enumerations

EaseType

Controls the interpolation curve applied to the tween.

- **LINEAR** — Constant speed from start to end.
- **QUAD_IN / QUAD_OUT / QUAD_IN_OUT** — Quadratic easing (gentle acceleration or deceleration).
- **CUBIC_IN / CUBIC_OUT / CUBIC_IN_OUT** — Cubic easing (stronger acceleration or deceleration).
- **QUART_IN / QUART_OUT / QUART_IN_OUT** — Quartic easing.

- **QUINT_IN / QUINT_OUT / QUINT_IN_OUT** — Quintic easing.
- **SINE_IN / SINE_OUT / SINE_IN_OUT** — Sinusoidal easing (smooth, natural motion).
- **EXPO_IN / EXPO_OUT / EXPO_IN_OUT** — Exponential easing (very sharp).
- **CIRC_IN / CIRC_OUT / CIRC_IN_OUT** — Circular easing.
- **ELASTIC_IN / ELASTIC_OUT / ELASTIC_IN_OUT** — Elastic spring overshoot.
- **BACK_IN / BACK_OUT / BACK_IN_OUT** — Slight overshoot (anticipation).
- **BOUNCE_IN / BOUNCE_OUT / BOUNCE_IN_OUT** — Bouncing ball effect.
- **STEP** — Hold the start value and jump at the end; on keyframe tracks, holds each key's value until the next key (how GLTF `STEP` clips import).
- **CUSTOM** — Provide your own ease function via the `Ease` overloads.

Property details

duration

- *Setter*: void **setDuration**(float duration)
- *Getter*: float **getDuration**() const

Length of the tween in **seconds**.

loop

- *Setter*: void **setLoop**(bool loop)
- *Getter*: bool **isLoop**() const

When `true`, the tween restarts from the beginning after completing.

pingPong

- *Setter*: void **setPingPong**(bool pingPong)
- *Getter*: bool **isPingPong**() const

When `true`, the tween plays to the end and then back to the start, so one pass takes twice the duration. With loop it goes back and forth for good, which makes a floating bob a single `PositionAction`. The easing applies both ways.

Method details

setFunctionType

- void **setFunctionType**(EaseType functionType)

Apply an easing curve to the tween. See EaseType for available values.

getValue / getTime

- float **getValue**() const
- float **getTime**() const

`getValue()` returns the current eased output value in `[0, 1]`. `getTime()` returns the current elapsed time in seconds.

TranslateTracks

Inherits: KeyframeTracks (p. 578)
C++ type: `TranslateTracks`

`TranslateTracks` API exposed to Lua and C++ gameplay code.

Constructors

Available in C++ and Lua:

- `TranslateTracks(Scene* scene)` — creates a new entity.
- `TranslateTracks(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.
- `TranslateTracks(Scene* scene, std::vector<float> times, std::vector<Vector3> values)` — creates a track initialized with key times and values.

In Lua, use `TranslateTracks(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Methods

Name	Languages
`setValues`	C++ | Lua
`getTranslateTracksComponent` (TranslateTracksComponent (p. 697))	Lua

The key times, easing, `loop` and `relative` come from KeyframeTracks (p. 578). Tracks imported from GLTF `CUBICSPLINE` clips carry per-key Hermite tangents and interpolate from them instead; `setValues` keeps existing tangent arrays sized to the new values. See Interpolation modes (p. 302).

Systems

ActionSystem

Inherits: SubSystem (p. 594)

Updates actions, animations, and particles.

Methods

Name	Languages
`actionStart`	C++ | Lua
`actionStop`	C++ | Lua
`actionPause`	C++ | Lua
`particleActionReset`	C++ | Lua
`getDuration`	C++ | Lua
`getFrameDuration`	C++ | Lua

`actionStart(entity, newRun)` resumes a paused action. With `newRun` true, the default, a stopped action starts over from its `startOffset`, or from a random point with `randomStart`.

`getDuration(entity)` returns an action's length in seconds, and `getFrameDuration(frame)` the length of one ActionFrame (p. 708) of an animation. `particleActionReset(entity)` stops a particle action and clears its particles.

AudioSystem

Inherits: SubSystem (p. 594)

SoLoud playback and 3D audio.

Methods

Name	Languages
`stopAll`	C++ | Lua
`pauseAll`	C++ | Lua
`resumeAll`	C++ | Lua
`checkActive`	C++ | Lua
`stopSceneSounds`	C++ | Lua
`setMuted`	C++

Constants

- `globalVolume`

Platform mute

`AudioSystem::setMuted(bool muted)` silences all audio, whatever `globalVolume` is set to, until it is called again with `false`. It is meant for mutes the platform imposes: web exports for YouTube Playables call it when YouTube turns the game's audio off. See Monetization → Pauses and mute (p. 348).

MeshSystem

Inherits: SubSystem (p. 594)

Creates meshes, loads models, builds sprites and tilemaps.

Methods

Name	Languages
`createPlane`	C++ | Lua
`createWall`	C++ | Lua
`createBox`	C++ | Lua
`createSphere`	C++ | Lua
`createCylinder`	C++ | Lua
`createCapsule`	C++ | Lua
`createTorus`	C++ | Lua
`setSpriteFrameRect`	C++
`canMergeStaticModel`	C++
`canEditModelHierarchy`	C++
`canEditModelPart`	C++
`hasCustomMeshParenting`	C++
`getModelNodeDefaultParents`	C++
`loadGLTF`	C++ | Lua
`loadOBJ`	C++ | Lua
`createInstancedMesh`	C++ | Lua
`removeInstancedMesh`	C++ | Lua
`hasInstancedMesh`	C++ | Lua
`hasPendingAsyncModelLoads`	C++ | Lua
`isAsyncModelLoadPending`	C++ | Lua
`cancelAsyncModelLoad`	C++ | Lua
`cancelAsyncModelLoads`	C++ | Lua
`cancelAllAsyncModelLoads`	C++

Sprite frame rectangles

`static void setSpriteFrameRect(MeshComponent& mesh, SpriteComponent& sprite, Rect frameRect)` applies a frame rectangle to the sprite's first submesh. Both components must belong to the same sprite. This C++ helper is shared by `Sprite::setFrame()` and sprite animation updates; gameplay code normally uses Sprite.setFrame (p. 484).

Rectangles with all four values in `[0, 1]` are treated as UVs. Other rectangles are converted from pixels using the texture dimensions. If dimensions are not yet available, the helper stores one pending rectangle on the sprite for mesh updates to retry. Subsequent calls replace that request; requests are not queued. Frame selection also works before the sprite geometry has been built. Applying a rectangle sets the sprite's `needUpdateBatch`, so a sprite drawn from a kept copy shows the new frame.

Static model hierarchy

`bool canMergeStaticModel(const ModelComponent& model, std::string* reason = nullptr) const` reports whether a loaded GLTF can flatten its mesh nodes into the root entity (`ModelComponent::mergeStaticMeshes`). Returns `false` with an optional reason for skinned, animated, morph-target, or single-node models, and when the flatten would exceed the root submesh limit. Animated models and models with more than one skin import a full node tree rather than mesh children only — see 3D Graphics — GLTF node hierarchy (p. 249) and Merging static model meshes (p. 250).

`bool canEditModelHierarchy(const ModelComponent& model, std::string* reason = nullptr) const` reports whether the model's child mesh entities may be reparented at all. Returns `false` with an optional reason while the model is still loading and when it has no separate mesh parts (a single mesh or a merged model).

`bool canEditModelPart(const ModelComponent& model, Entity part, std::string* reason = nullptr) const` reports whether one part — or a plain entity grouping parts — may leave the place the file gave it. Returns `false` with the reason for joints, parts with joints below them, parts with their own translation, rotation, or scale keyframes, and rigid (unskinned) parts whose glTF ancestry contains an animated node. Skinned parts always pass, since their placement comes from the joints. The editor uses both methods to decide which parts can be rearranged in the Structure panel (p. 77).

`std::map<int, Entity> getModelNodeDefaultParents(Entity modelEntity, const ModelComponent& model) const` maps each glTF node index to the entity the file parents that node under: its parent node's entity on the full node-tree path, or the model entity itself on the child-mesh path. `bool hasCustomMeshParenting(Entity modelEntity, const ModelComponent& model) const` is `true` when any mesh part sits under a different parent than that.

Asynchronous model-load control

These methods inspect or stop model loads started by the asynchronous resource pipeline. All but `cancelAllAsyncModelLoads` are also bound to Lua:

Method	Scope
`bool hasPendingAsyncModelLoads() const`	Reports whether this system's scene has pending model work
`bool isAsyncModelLoadPending(Entity entity, const std::string& filename) const`	Reports whether the named model is pending for this system's scene
`void cancelAsyncModelLoad(Entity entity, const std::string& filename)`	Stops the named pending load from being applied
`void cancelAsyncModelLoads()`	Stops all pending model loads for this system's scene
`static void cancelAllAsyncModelLoads()`	Stops pending model loads across all scenes

Cancellation is synchronous: if a worker is already decoding a model, the cancellation method waits for that task to finish, removes its pending result, and marks its resource build as failed before returning. This guarantees that the worker cannot outlive scene, project, or resource-pool teardown.

RenderSystem

Inherits: SubSystem (p. 594)

Draws the scene (meshes, lights, UI, particles).

Methods

Name	Languages
`updateCameraSize`	C++ | Lua
`isAllLoaded`	C++ | Lua

`isAllLoaded()` is true once every mesh, UI element, and other drawable in the scene has loaded. SceneManager (p. 601) `loadingProgress` reports the same count, together with sounds, as a fraction.

SubSystem

Base class of the systems each Scene (p. 390) runs: ActionSystem (p. 590), AudioSystem (p. 591), MeshSystem (p. 592), PhysicsSystem (p. 567), RenderSystem (p. 594), and UISystem (p. 595).

Properties

Type	Name	Default	Languages
bool	paused	`false`	C++ | Lua

While `paused` is `true` the system skips its update, so what it drives stops changing in that scene; the scene is still drawn. A paused AudioSystem also pauses the scene's playing sounds. In C++ the property is `setPaused(bool)` / `isPaused()`.

Pausing Play in the editor sets it on the physics, action, and audio systems, and resuming clears it.

Lua

```lua
scene:getPhysicsSystem().paused = true
```

C++

```cpp
scene->getSystem<PhysicsSystem>()->setPaused(true);
```

UISystem

Inherits: SubSystem (p. 594)

UI layout, focus, and pointer routing.

Methods

Name	Languages
`getAnchorReferenceRect`	C++ | Lua
`setAnchorReferenceSize`	C++ | Lua
`clearAnchorReferenceSize`	C++ | Lua
`getAnchorReferenceWidth`	C++ | Lua
`getAnchorReferenceHeight`	C++ | Lua
`parentToLayoutSpace`	C++ | Lua
`layoutToParentSpace`	C++ | Lua
`isTextEditFocused`	C++ | Lua
`isTextEditTyping`	C++ | Lua
`hasPendingImages`	C++ | Lua
`resetButtonStates`	C++ | Lua
`setFocus`	C++ | Lua
`hideFocus`	C++ | Lua
`eventOnCharInput`	C++ | Lua
`eventOnKeyDown`	C++ | Lua
`eventOnPointerDown`	C++ | Lua
`eventOnPointerUp`	C++ | Lua
`eventOnPointerMove`	C++ | Lua
`eventOnNavigate`	C++ | Lua
`eventOnSubmitDown`	C++ | Lua
`eventOnSubmitUp`	C++ | Lua
`eventOnSubmitCancel`	C++ | Lua

An anchored element's offset is measured in the space of its nearest layout parent, and its position in its own parent's space. `parentToLayoutSpace(entity, point)` and `layoutToParentSpace(entity, point)` convert a point between the two.

`hasPendingImages()` is true while an image still waits for its texture to build its geometry. `resetButtonStates()` puts every pressed button back to its normal look, without calling `onRelease`, and keeps the keyboard and gamepad focus. The engine calls it when it removes a scene and when it adds one, so a button focused before the scene runs stays focused.

`isTextEditTyping()` is true while a focused text edit is also visible and enabled. While one is, in any scene, the engine leaves the arrows and the submit keys to it.

`setFocus(entity)` gives an entity the keyboard and gamepad focus and shows it, and `NULL_ENTITY` clears it. Without a focused button, keys and gamepads stay with gameplay. `hideFocus()` stops showing it until the next key or gamepad press, the same as moving the mouse, and the focused button doesn't take Enter, Space, or A meanwhile. The engine calls it for a scene it removes. The engine calls `eventOnNavigate` with a NavigationDirection (p. 364) for the arrow keys, Tab, the d-pad, and the left stick, and `eventOnSubmitDown` / `eventOnSubmitUp` for Enter, Space, and the A button. `eventOnSubmitCancel` lets go of the pressed button

without a click, for a controller that was unplugged or a scene that stopped taking UI events. Each returns `true` when the scene used the event. See Keyboard and gamepad navigation (p. 295).

Managers

BundleManager

C++ type: `BundleManager` (static)

Description

A registry for reusable **bundles** — prefab-like groups of entities that can be spawned into any Scene (p. 390) at runtime. A bundle is authored in the editor and saved as a `.bundle` file (see Bundles (p. 188)).

Loading a bundle by API

There is **no API that loads a `.bundle` file directly at runtime**. Instead, each `.bundle` file is converted into a **factory function** during export, and the exporter emits a `registerBundle` call that wires that factory to a numeric **ID** and a **name**:

```
// Generated by export — you do not write this by hand
BundleManager::registerBundle(1, "enemies/EnemyShip", create_bundle_enemies_EnemyShip);
```

The generated `bundle_<name>.h` also declares `build_bundle_<name>(Scene* scene, Entity root, std::vector<Entity>& members)`, the same factory appending the entities it creates to `members`; generated scene code uses it to reach instance members, whose ids only exist at runtime. Spawn through `createBundle` rather than calling either function yourself, so the instance is tracked for `destroyBundle`.

The bundle **name** is the `.bundle` file's path with the extension removed, using forward slashes (for example, `bundles/enemies/EnemyShip.bundle` registers as `enemies/EnemyShip`). Registration covers the bundles a scene instantiates plus the ones listed in **Project → Bundles**; a bundle that exists only to be spawned from a script must be listed there, or it is not built and `createBundle` does not find it (see Bundles in the build (p. 114)). At runtime you spawn an instance by that name (or ID) with `createBundle`, and remove it with `destroyBundle`:

Lua

```
local root = BundleManager.createBundle("enemies/EnemyShip", scene)
-- ... later ...
BundleManager.destroyBundle(scene, root)
```

C++

```
Entity root = BundleManager::createBundle("enemies/EnemyShip", &scene);
// ... later ...
BundleManager::destroyBundle(&scene, root);
```

This makes bundles ideal for entities created on demand — enemies, projectiles, item drops, UI cards, and any pooled hierarchy.

Methods

Type	Name	Langs
static void	registerBundle	C++
static Entity	createBundle	C++ | Lua
static bool	destroyBundle	C++ | Lua
static uint32_t	getBundleId	C++ | Lua
static std::string	getBundleName	C++ | Lua
static std::vector<std::string>	getBundleNames	C++ | Lua

Type	Name	Langs
static void	destroyAllInstances	C++
static void	clearAll	C++ | Lua
static void	setScriptCallbacks	C++
static bool	isStopping	C++

Constants

Name	Type	Langs	Description
`bundleCount`	int	Lua	Number of registered bundles (the property form of `getBundleCount()`).

Method details

registerBundle

- static void **registerBundle**(uint32_t id, const std::string& name, std::function<bool(Scene*, Entity)> factory, std::function<bool(Scene*, Entity)> destroyer = nullptr)

Registers a named bundle factory. The `factory` receives the target `Scene*` and a root `Entity`, and must return `true` on success. On failure (or if the factory throws), `createBundle` does not record an instance and rolls back entities created during the call. Void-returning callables are still accepted and treated as success.

The optional `destroyer` is called by `destroyBundle` instead of the default entity destruction; it receives `(Scene*, Entity root)` and returns `true` on success.

This call is **generated by the export step** for every bundle the project builds, so you normally never write it yourself. It is C++-only and not exposed to Lua.

C++

```
BundleManager::registerBundle(1, "enemies/EnemyShip",
    [](Scene* scene, Entity root) -> bool {
        // create child entities/components under root...
        return true;
    });
```

createBundle

- static Entity **createBundle**(const std::string& name, Scene* scene)
- static Entity **createBundle**(uint32_t id, Scene* scene)
- static Entity **createBundle**(const std::string& name, Scene* scene, const std::string& parentName)
- static Entity **createBundle**(uint32_t id, Scene* scene, const std::string& parentName)
- static Entity **createBundle**(const std::string& name, const EntityHandle& parent)
- static Entity **createBundle**(uint32_t id, const EntityHandle& parent)
- static Entity **createBundle**(const std::string& name, Scene* scene, Entity parent)
- static Entity **createBundle**(uint32_t id, Scene* scene, Entity parent)

Spawns an instance of a registered bundle into a scene and returns the root entity of the new hierarchy, or a null entity if the name/ID is not found or the factory fails.

Every overload creates the instance root, so the same bundle can be spawned any number of times. The third argument only says where the new root is parented:

- The two-argument overloads leave the instance at the top level of the scene.
- `(name/id, scene, parentName)` looks up `parentName` **in that scene** and parents the instance under it.

- `(name/id, EntityHandle)` parents it under that object, using the scene already stored on the handle (any Object (p. 455), Button (p. 506), and so on).
- `(name/id, scene, Entity)` parents it under an existing entity id. That id must belong to `scene`.

Each successful call tracks the instance internally so `destroyBundle` can later remove it, leaving the parent alone. Note the argument order: the bundle **name/ID comes first, then the scene**.

The instance's scripts, Lua and C++, start before `createBundle` returns, the same way a scene's scripts do. Only scripts that have not started yet run, so the rest of the scene is left alone; spawning into a scene whose scripts were already cleaned up starts all of them again. If a script destroys its own instance from `init()` or its constructor, `createBundle` returns a null entity.

> **Entity IDs are scene-local**
>
> Each scene allocates its own entity IDs. When parenting, prefer `createBundle(name, scene, "parentName")` so the name is resolved in that scene, or pass an object that already carries its scene. Passing a raw `Entity` from another scene (for example a main-scene id into a UI scene) targets the wrong entity.

Lua

```lua
-- New root
local root = BundleManager.createBundle("enemy_grunt", scene)

-- By ID
local r2 = BundleManager.createBundle(1, scene)

-- Parented under a named entity in that scene
BundleManager.createBundle("enemy_grunt", scene, "spawn")

-- Parented under an object that already carries its scene
BundleManager.createBundle("enemy_grunt", someObject)

-- Parented under an existing entity id (must belong to scene)
BundleManager.createBundle("enemy_grunt", scene, existingEntity)
```

C++

```cpp
Entity root = BundleManager::createBundle("enemy_grunt", &scene);
Entity r2   = BundleManager::createBundle(1, &scene);
BundleManager::createBundle("enemy_grunt", &scene, "spawn");
BundleManager::createBundle("enemy_grunt", someObject);
BundleManager::createBundle("enemy_grunt", &scene, existingEntity);
```

destroyBundle

- static bool **destroyBundle**(Scene* scene, Entity rootEntity)

Destroys a bundle instance by its root entity. Its scripts stop first, then every entity the spawn created is removed together with anything parented under the instance since (children first, then the root). A bundle spawned under the instance goes through its own `destroyBundle` before that. The entity the instance was parented to is not touched. If the bundle was registered with a custom destroyer, that is called instead of the default destruction. Returns `false` if `rootEntity` is not a tracked bundle root. Note the argument order: **scene first, then the root entity**.

Lua

```lua
BundleManager.destroyBundle(scene, root)
```

C++

```cpp
BundleManager::destroyBundle(&scene, root);
```

A C++ script destroying its own bundle

The script is deleted during the call, so return right after it without touching a member. A Lua script can finish the callback, but its entity is already gone.

```
void Bullet::onUpdate() {
    if (hit) {
        BundleManager::destroyBundle(getScene(), getEntity()); // the script is on the root
        return;
    }
}
```

getBundleId / getBundleName

- static uint32_t **getBundleId**(const std::string& name)
- static std::string **getBundleName**(uint32_t id)

Look up a bundle's numeric ID by name, or its name by ID. Returns `0` / `""` if not found.

getBundleNames

- static std::vector<std::string> **getBundleNames**()

Returns all registered bundle names, in registration order. Use the `bundleCount` property (Lua) or `getBundleCount()` (C++) for the count.

destroyAllInstances

- static void **destroyAllInstances**(Scene* scene)

Destroys every tracked bundle instance belonging to `scene`, calling the appropriate destroyer for each. An instance spawned under another goes with it. C++-only; useful when tearing a scene down manually.

clearAll

- static void **clearAll**()

Removes all registered bundles **and** all tracked instances. Used by the editor when resetting state.

setScriptCallbacks

- static void **setScriptCallbacks**(std::function<void(Scene*)> start, std::function<void(Scene*, Entity)> stop)

Sets how spawned instances start and stop their scripts: `start` runs every script of the scene that has not started yet, and `stop` the scripts of one entity. By default only Lua scripts run. The generated `main.cpp` passes the project's `initScripts` and `cleanupEntityScripts`, and the editor sets its own for Play, so you only call this when running the engine without the generated code. C++-only.

isStopping

- static bool **isStopping**(Scene* scene, Entity entity)

Returns `true` while `destroyBundle` stops the scripts of that entity. A `start` callback skips such entities, so a script that spawns a bundle from its destructor does not restart what is being destroyed. C++-only.

SceneManager

C++ type: `SceneManager` (static)

Description

A registry for named scene stacks that lets you switch between scenes at runtime. A *scene stack* consists of one main Scene (p. 390) plus zero or more layer scenes (see Engine::addSceneLayer (p. 368)).

The typical workflow is:

1. **Register** all scenes at startup with the factories that build them. In an exported project this is done for you — the editor registers every scene stack.
2. **Load** a scene by name or ID; the manager calls `Engine::removeAllScenes()` to clear the current main scene and all layers, then invokes the new scene's factory function.

Two operations change what is on screen:

- `loadScene` is a full **transition** — it tears down everything and builds a new stack. Use it to move between levels, menus, and game-over screens.
- `addChildScene` / `removeChildScene` **overlay** a scene on top of the running stack without a transition. Use them for pause menus, dialogs, and HUD toggles. See Switching scenes with SceneManager (p. 185) for the broader workflow.

Methods

Type	Name	Langs
static void	registerScene	C++ | Lua
static bool	loadScene	C++ | Lua
static bool	isLoadPending	C++ | Lua (`loadPending`)
static bool	setLoadingScene	C++ | Lua
static uint32_t	getLoadingSceneId	C++ | Lua (`loadingSceneId`)
static void	setLoadingDelay	C++ | Lua (`loadingDelay`)
static float	getLoadingDelay	C++ | Lua (`loadingDelay`)
static void	setLoadingTimeout	C++ | Lua (`loadingTimeout`)
static float	getLoadingTimeout	C++ | Lua (`loadingTimeout`)
static bool	isLoading	C++ | Lua (`loading`)
static float	getLoadingProgress	C++ | Lua (`loadingProgress`)
static bool	holdLoading	C++ | Lua
static void	releaseLoading	C++ | Lua
static bool	isCoveredByLoading	C++ | Lua
static bool	preloadScene	C++ | Lua
static float	getPreloadProgress	C++ | Lua
static void	cancelPreload	C++ | Lua
static bool	addChildScene	C++ | Lua
static bool	removeChildScene	C++ | Lua
static uint32_t	getSceneId	C++ | Lua
static std::string	getSceneName	C++ | Lua
static std::vector<std::string>	getSceneNames	C++ | Lua
static int	getSceneCount	C++ | Lua (`sceneCount`)
static uint32_t	getCurrentSceneId	C++ | Lua (`currentSceneId`)

Type	Name	Langs
static std::string	getCurrentSceneName	C++ | Lua (`currentSceneName`)
static void	clearAll	C++ | Lua
static void	setScenePtr	C++ | Lua
static Scene*	getScenePtr	C++ | Lua
static void	removeScenePtr	C++ | Lua

Method details

registerScene

- static void **registerScene**(uint32_t id, const std::string& name, std::function<void()> loadFactory)
- static void **registerScene**(uint32_t id, const std::string& name, std::function<void()> loadFactory, const std::vector<uint32_t>& sceneIds)
- static void **registerScene**(uint32_t id, const std::string& name, std::function<void()> loadFactory, std::function<void()> addFactory)
- static void **registerScene**(uint32_t id, const std::string& name, std::function<void()> loadFactory, std::function<void()> addFactory, const std::vector<uint32_t>& sceneIds, std::function<SceneResources()> resources = nullptr)

Registers a named scene stack with the functions that build it. A stack has two of them because coming up *as the world* and coming up *on top of the world* are different jobs.

`loadFactory` runs on `loadScene`. It owns the transition: it calls `Engine::setScene()` for the stack's main scene, `Engine::addSceneLayer()` for its layers, and it is free to destroy the stack it replaces.

`addFactory` runs on `addChildScene`. It only **creates** the stack's scenes and registers each one with `setScenePtr`. It must not call `Engine::setScene()` and must not destroy anything, because the stack it is being layered onto keeps running. It does not add layers either — `addChildScene` does that, in stack order. It must be safe to call repeatedly, because `addChildScene` calls it every time.

A stack registered without an add factory can still be loaded. It can only be added as a child once something else has created its scenes.

The optional `sceneIds` vector lists the scenes that come up **together** with this one — the stack's *active* scenes. It is what `addChildScene` puts on screen, the stack's own scene first so it sits below the layers it owns. A braced list works in C++ (`{1, 2}`), and the stack's own id is added when the list leaves it out.

The optional **`resources`** function lists the files the scene loads — its `textures`, `sounds` and `models` — for `preloadScene`. Exported games and the editor's Play pass it for every scene.

C++

```
static Scene* menuScene = nullptr;

void load_MainMenu() {
    if (!menuScene) {
        menuScene = new Scene();
        SceneManager::setScenePtr(1, menuScene);
        // populate scene...
    }
    Engine::setScene(menuScene);
}

void add_MainMenu() {
    if (!menuScene) {
        menuScene = new Scene();
        SceneManager::setScenePtr(1, menuScene);
        // populate scene...
    }
```

```
    // no setScene, no teardown, no addSceneLayer
}

SceneManager::registerScene(1, "MainMenu", load_MainMenu, add_MainMenu);
SceneManager::loadScene("MainMenu");
```

Lua

```
local menuScene = nil

local function load_MainMenu()
    if not menuScene then
        menuScene = Scene()
        SceneManager.setScenePtr(1, menuScene)
        -- populate scene...
    end
    Engine.setScene(menuScene)
end

local function add_MainMenu()
    if not menuScene then
        menuScene = Scene()
        SceneManager.setScenePtr(1, menuScene)
        -- populate scene...
    end
    -- no setScene, no teardown, no addSceneLayer
end

SceneManager.registerScene(1, "MainMenu", load_MainMenu, add_MainMenu)
SceneManager.loadScene("MainMenu")
```

loadScene

- static bool **loadScene**(const std::string& name)
- static bool **loadScene**(uint32_t id)

Loads a registered scene stack by name or ID, performing a full scene transition. Calls `Engine::removeAllScenes()` first — this clears the current main scene **and** every layer — then runs the registered factory to build the new stack. Returns `false` if the name or ID is not found.

Because it clears everything, do not call `loadScene` once per layer. Persistent layers (a HUD, shared lighting) should be **start-active child scenes** of the target scene so they come up in the same call.

Called while a scene is running — from a script callback, a physics contact or an input event — the transition is **deferred to the start of the next frame**, because the factory tears down the scenes and scripts that are still executing. The replaced scenes are deleted, which stops their sounds; a loading delay lets a click sound finish first. The call returns `true` when the scene exists; the last request wins (a replaced request logs a warning), `isLoadPending` reports the wait, and `getCurrentSceneId` only changes once the load is applied. Only the first load, with nothing on screen yet (the `init()` entry point), builds the stack immediately, unless a loading scene is set: the start scene then waits for the first frame too, so the loading scene covers it. The editor's Play also builds the stack again from the scene files, so a level loaded a second time (a retry) starts over as it does in an exported game.

With a loading scene set, the switch also waits for it to be on screen and for the loading delay.

Lua

```
SceneManager.loadScene("Level2")
SceneManager.loadScene(2)
```

C++

```
SceneManager::loadScene("Level2");
SceneManager::loadScene(2);
```

addChildScene / removeChildScene

- static bool **addChildScene**(uint32_t id)
- static bool **addChildScene**(const std::string& name)
- static bool **removeChildScene**(uint32_t id)
- static bool **removeChildScene**(const std::string& name)

Add or remove a scene stack as layers on top of the running scenes **without** a full transition — ideal for overlaying a UI scene (pause menu, dialog) on top of a game scene.

`addChildScene` runs the stack's **add factory**, then adds its active scenes at the **top** of the layer list, in order, so the overlay draws above everything below it and the stack's own scene stays below the layers it owns.

The scene does **not** have to be loaded first. If its scenes do not exist yet the factory creates them on the spot; if they exist already the factory finds nothing to do and the call is cheap. Either way the factory runs on every call, which is what lets it restore a stack whose scripts were torn down earlier.

Unlike `loadScene`, this takes effect **immediately**, including when called inside a frame, because nothing is destroyed. The scenes it adds update and draw from that same frame on.

It can also be called from a script's `init` while that script's own stack is still loading — a level that opens its HUD from code, say. The stack it adds is kept when the load finishes, rather than torn down with the scenes the transition replaced.

Returns `false` if the scene is unknown, if it has no add factory and its scenes were never created, or if the factory did not produce the whole stack.

`removeChildScene` removes those layers again, keeping the main scene and any other layers intact. It does **not** destroy the scenes, so a later `addChildScene` brings the same ones back with their state intact. Returns `false` if nothing was removed.

See Preloaded or built on demand (p. 187) for choosing between a scene that is built with its parent and one built the first time it is shown.

Lua

```lua
SceneManager.addChildScene("PauseMenu")     -- show overlay
SceneManager.removeChildScene("PauseMenu")  -- hide it again
```

C++

```cpp
SceneManager::addChildScene("PauseMenu");
SceneManager::removeChildScene("PauseMenu");
```

getSceneId / getSceneName

- static uint32_t **getSceneId**(const std::string& name)
- static std::string **getSceneName**(uint32_t id)

Look up a scene's numeric ID by name, or its name by ID. Returns `0` / `""` if not found.

getSceneNames / getSceneCount

- static std::vector<std::string> **getSceneNames**()
- static int **getSceneCount**()

Returns all registered scene names, or the total number of registered scenes.

In Lua the count is the read-only property `SceneManager.sceneCount`, not a call.

isLoadPending

- static bool **isLoadPending**()

`true` between a deferred `loadScene` call and the frame that replaces the old stack, loading delay included. Lua reads it as the `SceneManager.loadPending` property.

setLoadingScene / getLoadingSceneId

- static bool **setLoadingScene**(const std::string& name)
- static bool **setLoadingScene**(uint32_t id)
- static uint32_t **getLoadingSceneId**()

Sets a registered scene stack as the **loading screen** that `loadScene` shows over every transition, the start scene included. An empty name or `0` turns it off. Returns `false` if the scene is unknown. Exported games and the editor's Play set it from Project Settings → General → Loading (p. 56), so most projects never call it.

A transition then runs in three steps:

1. The loading scene is added **on top** of the running scenes, which keep running underneath it. While it is shown, the scenes below it get no UI input: their buttons cannot be pressed and their text fields get no keys. Pointer releases still reach them, so nothing stays pressed. Engine input events such as `Engine::onKeyDown` still fire.
2. Once it is drawn with its resources loaded and the loading delay has passed, the old stack is replaced. The game freezes while the new scenes are built, with the loading scene on screen.
3. When every mesh, UI element, sky and sound of the new stack is loaded, `Engine::onSceneLoaded` (p. 373) fires and the loading scene is removed, unless a hold keeps it up to fade out.

A load that makes no progress for the loading timeout stops waiting.

It is kept alive between transitions, so its scripts keep their state. Their `onUpdate` also runs while it is off screen, so check `isLoading` first. It is not shown for a transition to a stack that includes it.

Lua

```lua
SceneManager.setLoadingScene("Loading")
SceneManager.loadingDelay = 0.3
SceneManager.loadScene("Level2")   -- Loading appears first, then Level2 behind it
```

C++

```cpp
SceneManager::setLoadingScene("Loading");
SceneManager::setLoadingDelay(0.3f);
SceneManager::loadScene("Level2");
```

setLoadingDelay / getLoadingDelay

- static void **setLoadingDelay**(float seconds)
- static float **getLoadingDelay**()

Seconds the loading scene covers the old scenes before they are replaced. Default `0`. The old scenes keep running below it until then, which leaves time for a fade in (a script on the loading scene can raise its alpha from `0` to `1` over the delay) and for their last sounds, such as a button click, which stop when the scenes are deleted.

setLoadingTimeout / getLoadingTimeout

- static void **setLoadingTimeout**(float seconds)
- static float **getLoadingTimeout**()

Seconds a load waits without progress before it gives up. Default `5`. It applies to the loading scene's own resources before the switch, and to the new stack after it, which logs a warning naming what did not load. The clock restarts every time another resource finishes, so a slow load that keeps moving is never cut short.

isLoading / getLoadingProgress

- static bool **isLoading**()
- static float **getLoadingProgress**()

`isLoading` is `true` from a `loadScene` call until the new stack has loaded its resources and every hold is released, with or without a loading scene. It is already `true` while the new stack's scripts are constructed, and for the start scene too. To start gameplay once the level is ready, handle `Engine::onSceneLoaded` (p. 373) instead of polling it.

`getLoadingProgress` is the share of the new stack's meshes, UI elements, skies and sounds that finished loading: `0` until the old stack is replaced, `1` once it has loaded (also after a timeout, and while holds keep the load open). A sound whose file cannot be loaded counts as done. The value is updated once per frame and never goes back within a load, even when the new scenes create more to load. It only moves across several frames with async loading (p. 367); otherwise it jumps from `0` to `1`. For a smooth bar, ease the drawn width toward it.

Lua

```lua
function LoadingScreen:onUpdate()
    if not SceneManager.loading then return end
    self.bar.width = 360 * SceneManager.loadingProgress
end
```

C++

```cpp
void LoadingScreen::onUpdate() {
    if (!SceneManager::isLoading()) return;
    barFill->setWidth((unsigned int)(360 * SceneManager::getLoadingProgress()));
}
```

holdLoading / releaseLoading

- static bool **holdLoading**()
- static void **releaseLoading**()

`holdLoading` keeps the current load open after the new stack has loaded: the loading scene stays on screen and `isLoading` stays `true` until every hold is released. Returns `false`, and holds nothing, when no load is running. Each successful `holdLoading` needs one `releaseLoading`.

The usual use is a fade out: a script on the loading scene holds the load when `Engine::onSceneLoaded` (p. 373) fires, lowers its alpha, then releases it. The new scene's scripts have already started by then, so the level is ready behind the fade. A hold has no timeout, so release it in the script's destructor too.

Lua

```lua
function LoadingScreen:init()
    RegisterEngineEvent(self, "onUpdate")
    RegisterEngineEvent(self, "onSceneLoaded")
end

function LoadingScreen:onSceneLoaded()
    -- only when this loading scene is on screen
    if Engine.isSceneRunning(self.scene) and SceneManager.holdLoading() then
        self.fading = 0
    end
end

function LoadingScreen:onUpdate()
```

```
    if not self.fading then return end
    self.fading = self.fading + Engine.deltatime
    self.backdrop.alpha = math.max(0, 1 - self.fading / 0.3)
    if self.fading >= 0.3 then
        self.fading = nil
        SceneManager.releaseLoading()
    end
end
```

C++

```
void LoadingScreen::onSceneLoaded() {
    // only when this loading scene is on screen
    if (Engine::isSceneRunning(scene) && SceneManager::holdLoading()) {
        holding = true;
        fadeTimer = 0.0f;
    }
}

void LoadingScreen::onUpdate() {
    if (!holding) return;
    fadeTimer += Engine::getDeltatime();
    backdrop->setAlpha(std::max(0.0f, 1.0f - fadeTimer / 0.3f));
    if (fadeTimer >= 0.3f) {
        holding = false;
        SceneManager::releaseLoading();
    }
}
```

isCoveredByLoading

- static bool **isCoveredByLoading**(Scene* scene)

`true` for a running scene below the shown loading scene. Such a scene gets no UI input, but its scripts still receive engine input events, so a script can check this to ignore keys while its scene is covered.

Lua

```
function Menu:onKeyDown(key)
    if SceneManager.isCoveredByLoading(self.scene) then return end
    -- menu keys
end
```

C++

```
void Menu::onKeyDown(int key, bool repeat, int mods) {
    if (SceneManager::isCoveredByLoading(scene)) return;
    // menu keys
}
```

preloadScene / getPreloadProgress / cancelPreload

- static bool **preloadScene**(const std::string& name)
- static bool **preloadScene**(uint32_t id)
- static float **getPreloadProgress**(const std::string& name)
- static float **getPreloadProgress**(uint32_t id)
- static void **cancelPreload**(const std::string& name)
- static void **cancelPreload**(uint32_t id)

`preloadScene` loads the textures, sounds and models of a scene stack in the background while the current scene keeps running, so a later `loadScene` finds them ready and only has to build the scene. Call it early, for example when the player gets close to the exit door. It returns `false` for an unknown scene.

The files are held until that scene has loaded, or until `cancelPreload`, which frees the ones no running scene uses. Scenes of the stack that are already running, such as a shared HUD, are skipped. Calling `loadScene` before the preload finishes is fine: files still loading are not loaded twice. `getPreloadProgress` goes from `0` to `1` once the request starts at the end of the frame, and is `0` for a scene that was never preloaded.

Preloading needs async loading (p. 367) to stay smooth. Without it, one file loads per frame on the main thread and models are skipped. Preloading only loads files: building the scene's entities still happens in `loadScene`. In the editor's Play, a scene that is not open is read from its file first, which takes a moment.

Lua

```lua
function ExitDoor:onPlayerNear()
    SceneManager.preloadScene("Level2")
end

function ExitDoor:onEnter()
    SceneManager.loadScene("Level2")   -- its files are ready
end
```

C++

```cpp
void ExitDoor::onPlayerNear() {
    SceneManager::preloadScene("Level2");
}

void ExitDoor::onEnter() {
    SceneManager::loadScene("Level2");   // its files are ready
}
```

getCurrentSceneId / getCurrentSceneName

- static uint32_t **getCurrentSceneId**()
- static std::string **getCurrentSceneName**()

Returns the ID or name of the most recently loaded scene.

Lua

```lua
-- read-only properties, not calls
local id = SceneManager.currentSceneId
local name = SceneManager.currentSceneName
```

C++

```cpp
uint32_t id = SceneManager::getCurrentSceneId();
std::string name = SceneManager::getCurrentSceneName();
```

clearAll

- static void **clearAll**()

Unregisters all scenes. Used by the editor when resetting state. Does not destroy scene objects.

setScenePtr / getScenePtr / removeScenePtr

- static void **setScenePtr**(uint32_t id, Scene* scene)
- static Scene* **getScenePtr**(uint32_t id)
- static void **removeScenePtr**(uint32_t id)

Associates a live `Scene*` pointer with a registered scene ID. Used when one scene needs to reference entities in another scene (cross-scene entity resolution), and by an add factory to hand `addChildScene` the scenes it created. `getScenePtr` returns `nullptr` (`nil` in Lua) for an ID with no scene.

Lua

```lua
SceneManager.setScenePtr(2, hudScene)
local hud = SceneManager.getScenePtr(2)
SceneManager.removeScenePtr(2)
```

C++

```cpp
SceneManager::setScenePtr(2, &hudScene);
Scene* hud = SceneManager::getScenePtr(2);
SceneManager::removeScenePtr(2);
```

Services

AdMob

C++ type: `AdMob` (static) · **Header:** `AdMob.h`

Description

Google Mobile Ads on Android and iOS: banner, interstitial, rewarded, rewarded interstitial and app open ads, plus Google's consent form (User Messaging Platform). Exports include the SDK only when **Google AdMob** is enabled in the project's Android (p. 62) or iOS (p. 62) settings. Everywhere else the calls still compile and run, and report `errorNotAvailable` through their events.

Results arrive as events at the start of a frame, never as return values. Loading a format again replaces its ad. See Monetization (p. 338) for the setup and the full flow.

C++

```
REGISTER_EVENT(AdMob::onUserEarnedReward, onUserEarnedReward);
AdMob::loadRewardedAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx");

void Shop::onUserEarnedReward(AdMobFormat format, std::string type, int amount) {
    coins += amount;
}
```

Lua

```
RegisterEvent(self, AdMob.onUserEarnedReward, "onUserEarnedReward")
AdMob.loadRewardedAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx")

function Shop:onUserEarnedReward(format, rewardType, amount)
    self.coins = self.coins + amount
end
```

Methods

Type	Name	Langs
static void	initialize	C++ | Lua
static bool	isInitialized	C++ | Lua
static void	setMaxAdContentRating	C++ | Lua
static AdMobRating	getMaxAdContentRating	C++ | Lua
static void	setAgeRestriction	C++ | Lua
static AdMobAgeRestriction	getAgeRestriction	C++ | Lua
static void	setPersonalization	C++ | Lua
static AdMobPersonalization	getPersonalization	C++ | Lua
static void	setTestDeviceIds	C++ | Lua
static std::vector<std::string>	getTestDeviceIds	C++ | Lua
static void	requestConsent	C++ | Lua
static void	setConsentDebugGeography	C++ | Lua
static AdMobDebugGeography	getConsentDebugGeography	C++ | Lua
static AdMobConsentStatus	getConsentStatus	C++ | Lua
static bool	canRequestAds	C++ | Lua
static bool	isPrivacyOptionsRequired	C++ | Lua
static void	showPrivacyOptionsForm	C++ | Lua

Type	Name	Langs
static void	resetConsent	C++ | Lua
static void	loadBannerAd	C++ | Lua
static bool	isBannerAdLoaded	C++ | Lua
static void	showBannerAd	C++ | Lua
static void	hideBannerAd	C++ | Lua
static bool	isBannerAdVisible	C++ | Lua
static void	setBannerAdPosition	C++ | Lua
static AdMobBannerPosition	getBannerAdPosition	C++ | Lua
static void	removeBannerAd	C++ | Lua
static int	getBannerAdWidth	C++ | Lua
static int	getBannerAdHeight	C++ | Lua
static void	loadInterstitialAd	C++ | Lua
static bool	isInterstitialAdLoaded	C++ | Lua
static void	showInterstitialAd	C++ | Lua
static void	loadRewardedAd	C++ | Lua
static bool	isRewardedAdLoaded	C++ | Lua
static void	showRewardedAd	C++ | Lua
static void	loadRewardedInterstitialAd	C++ | Lua
static bool	isRewardedInterstitialAdLoaded	C++ | Lua
static void	showRewardedInterstitialAd	C++ | Lua
static void	loadAppOpenAd	C++ | Lua
static bool	isAppOpenAdLoaded	C++ | Lua
static void	showAppOpenAd	C++ | Lua
static void	setServerSideVerificationOptions	C++ | Lua
static void	setAppVolume	C++ | Lua
static void	setAppMuted	C++ | Lua
static void	openAdInspector	C++ | Lua

Callback events

Callback	Name	Languages
void()	onInitialized	C++ | Lua
void(int, std::string)	onConsentUpdated	C++ | Lua
void(AdMobFormat)	onAdLoaded	C++ | Lua
void(AdMobFormat, int, std::string)	onAdFailedToLoad	C++ | Lua
void(AdMobFormat)	onAdShown	C++ | Lua
void(AdMobFormat, int, std::string)	onAdFailedToShow	C++ | Lua
void(AdMobFormat)	onAdDismissed	C++ | Lua
void(AdMobFormat)	onAdClicked	C++ | Lua
void(AdMobFormat)	onAdImpression	C++ | Lua
void(AdMobFormat, std::string, int)	onUserEarnedReward	C++ | Lua
void(AdMobFormat, long long, std::string, AdMobPrecision)	onAdPaid	C++ | Lua
void(int, std::string)	onAdInspectorClosed	C++ | Lua

Method details

initialize / isInitialized

- static void **initialize**()
- static bool **isInitialized**()

Starts Google Mobile Ads with the request settings made so far. `onInitialized` fires when it is ready, and `isInitialized()` is true from then on. Call it once consent allows ad requests (`canRequestAds()`), and load the first ads in `onInitialized`.

Request settings

- static void **setMaxAdContentRating**(AdMobRating rating)
- static AdMobRating **getMaxAdContentRating**()
- static void **setAgeRestriction**(AdMobAgeRestriction restriction)
- static AdMobAgeRestriction **getAgeRestriction**()
- static void **setPersonalization**(AdMobPersonalization personalization)
- static AdMobPersonalization **getPersonalization**()

Settings for the ad requests made after them. Set them before `requestConsent()` and `initialize()`; settings made earlier reach the first requests too. All default to `UNSPECIFIED` or `DEFAULT`.

- **Max ad content rating** — the highest content rating of the ads served.
- **Age restriction** — `CHILD` for a game directed at children and `TEEN` for one directed at teens. `CHILD` also makes `requestConsent()` ask as for a user under the age of consent.
- **Personalization** — `DISABLED` asks for non-personalized ads only.

C++

```
AdMob::setAgeRestriction(AdMobAgeRestriction::CHILD);
AdMob::setMaxAdContentRating(AdMobRating::GENERAL);
AdMob::requestConsent();
```

Lua

```
AdMob.setAgeRestriction(AdMobAgeRestriction.CHILD)
AdMob.setMaxAdContentRating(AdMobRating.GENERAL)
AdMob.requestConsent()
```

setTestDeviceIds / getTestDeviceIds

- static void **setTestDeviceIds**(const std::vector<std::string>& ids)
- static std::vector<std::string> **getTestDeviceIds**()

Devices that get test ads, by the hashed ID Google Mobile Ads prints to the device log (Logcat or the Xcode console). The consent debug settings also apply only to these devices. In Lua, pass a table of strings.

Lua

```
AdMob.setTestDeviceIds({ "33BE2250B43518CCDA7DE426D04EE231" })
```

requestConsent

- static void **requestConsent**()

Updates the user's consent information and shows Google's consent form when the user still has to answer, as Google's policies require in the EEA, the UK, Switzerland and some US states. Call it at every launch, before

`initialize()`. `onConsentUpdated(errorCode, message)` reports the result, with `errorCode` 0 on success. Without AdMob in the build it reports `errorNotAvailable`.

setConsentDebugGeography / getConsentDebugGeography

- static void **setConsentDebugGeography**(AdMobDebugGeography geography)
- static AdMobDebugGeography **getConsentDebugGeography**()

Makes the next `requestConsent()` on a test device act as if the user were in that region, to see the consent form from anywhere. Defaults to `DISABLED`.

getConsentStatus / canRequestAds

- static AdMobConsentStatus **getConsentStatus**()
- static bool **canRequestAds**()

The consent state Google has stored. `canRequestAds()` is true once the consent allows ad requests, including consent from an earlier launch. Without AdMob in the build they return `UNKNOWN` and `false`.

isPrivacyOptionsRequired / showPrivacyOptionsForm

- static bool **isPrivacyOptionsRequired**()
- static void **showPrivacyOptionsForm**()

When `isPrivacyOptionsRequired()` is true, the game must offer a way back to the consent form, such as a **Privacy** button in its settings, which calls `showPrivacyOptionsForm()`. Closing the form reports through `onConsentUpdated`.

resetConsent

- static void **resetConsent**()

Forgets the stored consent, so the next `requestConsent()` asks again. Meant for testing.

loadBannerAd

- static void **loadBannerAd**(const std::string& adUnitId, AdMobBannerSize size = ADAPTIVE, AdMobBannerPosition position = BOTTOM)

Loads a banner and shows it over the game as soon as it loads, replacing any banner already there. `onAdLoaded(BANNER)` or `onAdFailedToLoad(BANNER, ...)` reports the result. An `ADAPTIVE` banner takes the screen width when it loads, so load it again after the screen rotates.

Banners refresh at the rate set on their ad unit. A refresh that fails reports `onAdFailedToLoad(BANNER, ...)` while the last ad keeps showing, and `isBannerAdLoaded()` stays true.

C++

```
AdMob::loadBannerAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx");
AdMob::loadBannerAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx", AdMobBannerSize::BANNER, AdMobBannerPosi
tion::TOP);
```

Lua

```
AdMob.loadBannerAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx")
AdMob.loadBannerAd("ca-app-pub-xxxxxxxxxxxxxxxx/xxxxxxxxxx", AdMobBannerSize.BANNER, AdMobBannerPositi
on.TOP)
```

Banner control

- static bool **isBannerAdLoaded**()
- static void **showBannerAd**()
- static void **hideBannerAd**()
- static bool **isBannerAdVisible**()
- static void **setBannerAdPosition**(AdMobBannerPosition position)
- static AdMobBannerPosition **getBannerAdPosition**()
- static void **removeBannerAd**()
- static int **getBannerAdWidth**()
- static int **getBannerAdHeight**()

`hideBannerAd()` and `showBannerAd()` toggle the banner without reloading it, and `isBannerAdVisible()` is true when it is loaded and not hidden. `setBannerAdPosition()` moves it. `removeBannerAd()` destroys it; results of its load still on their way are dropped.

`getBannerAdWidth()` and `getBannerAdHeight()` give the loaded banner's size in screen pixels, as `System::getScreenWidth` (p. 408) counts them, and are 0 until it loads. On Android the banner also stays clear of display cutouts and visible system bars.

Full-screen ads

- static void **loadInterstitialAd**(const std::string& adUnitId)
- static bool **isInterstitialAdLoaded**()
- static void **showInterstitialAd**()
- static void **loadRewardedAd**(const std::string& adUnitId)
- static bool **isRewardedAdLoaded**()
- static void **showRewardedAd**()
- static void **loadRewardedInterstitialAd**(const std::string& adUnitId)
- static bool **isRewardedInterstitialAdLoaded**()
- static void **showRewardedInterstitialAd**()
- static void **loadAppOpenAd**(const std::string& adUnitId)
- static bool **isAppOpenAdLoaded**()
- static void **showAppOpenAd**()

Each format holds one ad. `load...()` loads it, replacing a loaded one, and `onAdLoaded(format)` or `onAdFailedToLoad(format, errorCode, message)` reports the result; only the latest load of a format reports. `is...Loaded()` is true from `onAdLoaded` until the ad is shown.

`show...()` shows the ad once: `onAdShown`, `onAdImpression` and `onAdDismissed` follow, and the format needs loading again. Showing a format that is not loaded reports `onAdFailedToShow(format, errorNotLoaded, message)`. While the ad covers the game, the engine pauses (p. 338).

- **Rewarded** and **rewarded interstitial** ads report the reward through `onUserEarnedReward(format, type, amount)`. A player who closes the ad early gets no such event.
- **App open** ads expire four hours after loading, as Google stops serving them then: `isAppOpenAdLoaded()` turns false, and showing one reports "The ad expired, load it again".

setServerSideVerificationOptions

- static void **setServerSideVerificationOptions**(const std::string& userId, const std::string& customData = "")

Data passed to the server-side verification callback of rewarded ads, which lets your server confirm a reward before granting it. It applies to the rewarded and rewarded interstitial ads shown after the call.

setAppVolume / setAppMuted

- static void **setAppVolume**(float volume)
- static void **setAppMuted**(bool muted)

Volume of video ads relative to the game, from 0 to 1, and whether they play muted. Match them to the game's own volume settings.

openAdInspector

- static void **openAdInspector**()

Opens Google's Ad Inspector, which shows the requests and mediation results of each ad unit. It only opens on a test device. `onAdInspectorClosed(errorCode, message)` fires when it closes.

Events

Each event is a static `FunctionSubscribe` member; subscribe with `REGISTER_EVENT` in C++ or `RegisterEvent` in Lua (see Events (p. 221)). C++ handlers take the parameters by value, exactly as listed.

Event	Parameters	When
`onInitialized`	—	Google Mobile Ads is ready to load ads
`onConsentUpdated`	`int errorCode, std::string message`	`requestConsent()` or the privacy options form finished. `errorCode` is 0 on success
`onAdLoaded`	`AdMobFormat format`	An ad loaded and can be shown
`onAdFailedToLoad`	`AdMobFormat format, int errorCode, std::string message`	An ad failed to load, with Google's error code
`onAdShown`	`AdMobFormat format`	A full-screen ad opened, or a banner opened an overlay after a click
`onAdFailedToShow`	`AdMobFormat format, int errorCode, std::string message`	A full-screen ad could not be shown
`onAdDismissed`	`AdMobFormat format`	A full-screen ad, or a banner's overlay, closed
`onAdClicked`	`AdMobFormat format`	An ad was clicked
`onAdImpression`	`AdMobFormat format`	An ad recorded an impression
`onUserEarnedReward`	`AdMobFormat format, std::string type, int amount`	The player earned the reward of a rewarded ad. Type and amount come from the ad unit
`onAdPaid`	`AdMobFormat format, long long valueMicros, std::string currencyCode, AdMobPrecision precision`	An ad earned revenue, in millionths of the currency
`onAdInspectorClosed`	`int errorCode, std::string message`	The Ad Inspector closed

On Android, the events of a full-screen ad arrive when it closes, since the whole app pauses while it shows.

Constants

Error codes the engine reports in place of Google's:

Name	Value	Meaning
`errorNotAvailable`	`-1`	The platform or the build has no AdMob
`errorNotLoaded`	`-2`	The ad shown was not loaded, or has expired

Enumerations

AdMobFormat

- **BANNER**
- **INTERSTITIAL**
- **REWARDED**
- **REWARDED_INTERSTITIAL**
- **APP_OPEN**

AdMobBannerSize

Fixed sizes are in Google's units: dp on Android, points on iOS.

- **ADAPTIVE** — Anchored adaptive banner: the screen width, with a height Google picks.
- **BANNER** — 320×50.
- **LARGE_BANNER** — 320×100.
- **MEDIUM_RECTANGLE** — 300×250.
- **FULL_BANNER** — 468×60.
- **LEADERBOARD** — 728×90.

AdMobBannerPosition

`TOP`, `BOTTOM`, `TOP_LEFT`, `TOP_RIGHT`, `BOTTOM_LEFT`, `BOTTOM_RIGHT`, `CENTER`

AdMobRating

- **UNSPECIFIED** — No limit set.
- **GENERAL** — Content for general audiences.
- **PARENTAL_GUIDANCE** — Content for most audiences, with parental guidance.
- **TEEN** — Content for teen and older audiences.
- **MATURE_AUDIENCE** — Content only for mature audiences.

AdMobAgeRestriction

- **UNSPECIFIED** — No age treatment.
- **CHILD** — Directed at children; also asks for consent as for a user under the age of consent.
- **TEEN** — Directed at teens.

AdMobPersonalization

- **DEFAULT** — Google decides.
- **ENABLED** — Personalized ads allowed.
- **DISABLED** — Non-personalized ads only.

AdMobConsentStatus

- **UNKNOWN** — Not known yet, or no AdMob in the build.
- **REQUIRED** — The user has to answer the consent form.
- **NOT_REQUIRED** — No consent needed, as outside the regulated regions.
- **OBTAINED** — The user answered.

AdMobDebugGeography

`DISABLED`, `EEA`, `REGULATED_US_STATE`, `OTHER`

AdMobPrecision

How exact an `onAdPaid` value is.

- **UNKNOWN**
- **ESTIMATED** — Estimated from aggregated data.
- **PUBLISHER_PROVIDED** — From the CPM set in a mediation network.
- **PRECISE** — The value paid for this ad.

See also

- Monetization (p. 337) — setup, consent, and testing
- InAppPurchase (p. 618) and WebPortal (p. 626)

InAppPurchase

C++ type: `InAppPurchase` (static) · **Header:** `InAppPurchase.h`

Description

One-time products, consumable or not, and subscriptions, through Google Play Billing on Android and StoreKit 2 on iOS. Exports include it only when **Google Play Billing** is enabled in the project's Android settings (p. 62), or **App Store Purchases** in its iOS settings (p. 62). On every other platform the calls report `BILLING_UNAVAILABLE` through their events.

Results arrive as events at the start of a frame. A purchase must be acknowledged or consumed: Google Play refunds one left unsettled for three days, and the App Store delivers it again at every launch. See Monetization → In-app purchases (p. 342) for the setup, a complete store script, and the App Store differences (p. 346).

C++

```cpp
REGISTER_EVENT(InAppPurchase::onPurchaseUpdated, onPurchaseUpdated);
InAppPurchase::purchase("remove_ads");

void Store::onPurchaseUpdated(PurchaseDetails purchase) {
    if (purchase.state == PurchaseState::PURCHASED && !purchase.acknowledged) {
        InAppPurchase::acknowledgePurchase(purchase.purchaseToken);
    }
}
```

Lua

```lua
RegisterEvent(self, InAppPurchase.onPurchaseUpdated, "onPurchaseUpdated")
InAppPurchase.purchase("remove_ads")

function Store:onPurchaseUpdated(purchase)
    if purchase.state == PurchaseState.PURCHASED and not purchase.acknowledged then
        InAppPurchase.acknowledgePurchase(purchase.purchaseToken)
    end
end
```

Methods

Type	Name	Langs
static void	initialize	C++ | Lua
static bool	isReady	C++ | Lua
static void	queryProducts	C++ | Lua
static bool	hasProduct	C++ | Lua
static ProductDetails	getProduct	C++ | Lua
static std::vector<ProductDetails>	getProducts	C++ | Lua
static void	purchase	C++ | Lua
static void	changeSubscription	C++ | Lua
static void	acknowledgePurchase	C++ | Lua
static void	consumePurchase	C++ | Lua
static void	queryPurchases	C++ | Lua
static void	restorePurchases	C++ | Lua
static std::vector<PurchaseDetails>	getPurchases	C++ | Lua
static bool	isPurchased	C++ | Lua
static void	setObfuscatedAccountId	C++ | Lua

Type	Name	Langs
static void	setObfuscatedProfileId	C++ | Lua
static void	openSubscriptionManagement	C++ | Lua
static void	showInAppMessages	C++ | Lua

Callback events

Callback	Name	Languages
void(BillingResponse, std::string)	onInitialized	C++ | Lua
void()	onDisconnected	C++ | Lua
void(BillingResponse, std::string)	onProductsQueried	C++ | Lua
void(PurchaseDetails)	onPurchaseUpdated	C++ | Lua
void(std::string, BillingResponse, std::string)	onPurchaseFailed	C++ | Lua
void(ProductType, BillingResponse, std::string)	onPurchasesQueried	C++ | Lua
void(std::string, BillingResponse, std::string)	onPurchaseAcknowledged	C++ | Lua
void(std::string, BillingResponse, std::string)	onPurchaseConsumed	C++ | Lua

Method details

initialize / isReady

- static void **initialize**()
- static bool **isReady**()

Connects to the store. `onInitialized(response, message)` reports `OK` once it is ready, and every call before that fails with `SERVICE_DISCONNECTED`. Calling `initialize()` again while connected reports `OK` again.

On Android, `isReady()` is true while connected to Google Play. When the connection drops, `onDisconnected` fires and the next call connects again by itself. On iOS, `initialize()` reports `OK` at once and starts listening for App Store transactions, which delivers the ones left unsettled in earlier launches through `onPurchaseUpdated`.

queryProducts

- static void **queryProducts**(const std::vector<std::string>& productIds, ProductType type)

Fetches the details of these products, which must all be of `type`: their localized names, descriptions, prices and, for subscriptions, plans and offers. `onProductsQueried(response, message)` reports when they are stored, and products the store does not know are left out. On iOS so are products of the other type, and the message names both kinds. An empty list reports `DEVELOPER_ERROR`. In Lua, pass a table of strings.

C++

```
InAppPurchase::queryProducts({"coins_100", "remove_ads"}, ProductType::INAPP);
InAppPurchase::queryProducts({"premium"}, ProductType::SUBS);
```

Lua

```
InAppPurchase.queryProducts({ "coins_100", "remove_ads" }, ProductType.INAPP)
InAppPurchase.queryProducts({ "premium" }, ProductType.SUBS)
```

hasProduct / getProduct / getProducts

- static bool **hasProduct**(const std::string& productId)
- static ProductDetails **getProduct**(const std::string& productId)
- static std::vector<ProductDetails> **getProducts**()

The products queried so far. `getProduct()` returns an empty `ProductDetails` for a product that was not queried, so check `hasProduct()` first.

Lua

```lua
for _, product in ipairs(InAppPurchase.getProducts()) do
    Log.print(product.name .. " " .. product.price)
end
```

purchase

- static void **purchase**(const std::string& productId, const std::string& offerToken = "")

Opens the store's purchase screen for a queried product. The result is `onPurchaseUpdated` for a completed or pending purchase, or `onPurchaseFailed(productId, response, message)`, for example with `USER_CANCELED`, `ITEM_ALREADY_OWNED`, or `ITEM_UNAVAILABLE` for a product that was not queried.

On Google Play, `offerToken` picks a subscription plan from the product's `offers`. Left empty, a subscription buys its first base plan (or its first offer when it has no base plan), and a one-time product its default purchase option. The App Store ignores it: a product has one plan, and its introductory offer applies by itself.

On iOS, a device not allowed to make payments reports `BILLING_UNAVAILABLE`, and a purchase waiting for a parent's approval (Ask to Buy) is reported as `PENDING` with no `purchaseToken`; once approved, it arrives as a new purchase.

The purchases include the ids set with `setObfuscatedAccountId` and `setObfuscatedProfileId`.

changeSubscription

- static void **changeSubscription**(const std::string& productId, const std::string& offerToken, const std::string& oldPurchaseToken, SubscriptionReplacementMode mode = WITH_TIME_PRORATION)

Replaces an owned subscription with another plan or product, an upgrade or a downgrade. `oldPurchaseToken` must belong to a purchase in `getPurchases()`, so query `SUBS` purchases first; otherwise `onPurchaseFailed` reports `DEVELOPER_ERROR`. `mode` decides how the time left on the old plan is billed. `KEEP_EXISTING` keeps the old plan's payment schedule and ignores `offerToken`.

On iOS it buys `productId` like `purchase()`, and the App Store replaces the subscription of the same group; `offerToken` and `mode` are ignored.

acknowledgePurchase / consumePurchase

- static void **acknowledgePurchase**(const std::string& purchaseToken)
- static void **consumePurchase**(const std::string& purchaseToken)

Settle a `PURCHASED` purchase. Google Play refunds one left unsettled for three days, and the App Store delivers it again at every launch. On iOS, settling finishes the purchase's transactions.

- **Acknowledge** products bought once and subscriptions. `onPurchaseAcknowledged(purchaseToken, response, message)` reports it, and the stored purchase's `acknowledged` turns true.
- **Consume** consumables, like coins. Consuming also acknowledges, removes the purchase from `getPurchases()`, and lets the player buy the product again. `onPurchaseConsumed(purchaseToken, response, message)` reports it.

Grant and save a consumable *before* consuming it, once per `purchaseToken`: the same purchase can be reported twice, and a consumed one is never reported again. On iOS, consuming a product that is not a consumable in App Store Connect reports `DEVELOPER_ERROR`, and consuming one twice reports `ITEM_NOT_OWNED`. Never settle a `PENDING` purchase.

queryPurchases

- static void **queryPurchases**(ProductType type)

Asks the store for the player's purchases of `type`: active subscriptions, products bought once, and consumables not consumed yet. Each one is reported through `onPurchaseUpdated` with `restored` set, then `onPurchasesQueried(type, response, message)` fires. With `OK`, the result replaces the stored purchases of that type, so refunded and expired ones go away.

Call it for each type at startup and on `Engine::onResume` (p. 373): purchases also complete outside the app, through pending payments, promo codes or another device.

restorePurchases

- static void **restorePurchases**()

For a **Restore Purchases** button: queries the purchases of both types, as two `queryPurchases` calls do. On iOS it first syncs the player's App Store account, which may ask them to sign in; a failed sync reports its error through both `onPurchasesQueried` events. Apple's review guidelines ask apps to offer a way to restore purchases that can be restored.

getPurchases / isPurchased

- static std::vector<PurchaseDetails> **getPurchases**()
- static bool **isPurchased**(const std::string& productId)

The purchases known in this session, from purchase screens and `queryPurchases`, less the consumed ones. `isPurchased()` is true when one of them includes the product, is `PURCHASED`, and is not `suspended`. It does not check acknowledgment.

setObfuscatedAccountId / setObfuscatedProfileId

- static void **setObfuscatedAccountId**(const std::string& accountId)
- static void **setObfuscatedProfileId**(const std::string& profileId)

A hashed id of the player's account in your game, and of a profile within it, attached to the next purchases. Google Play uses them against fraud, and they come back in `PurchaseDetails` for your server. Never pass plain personal data.

The App Store takes the account id only as a UUID, which it keeps as the transaction's `appAccountToken`; any other value is left out with a warning. It has no profile id.

openSubscriptionManagement

- static void **openSubscriptionManagement**(const std::string& productId = "")

Opens the store's subscription management, on one subscription or on all of them, where the player can cancel or fix a payment. On Android it opens Google Play's subscription center. On iOS it shows Apple's subscription sheet over the game, on the product's subscription group from iOS 17, and queries the subscriptions again when it closes.

showInAppMessages

- static void **showInAppMessages**()

Lets Google Play show its messages about subscription payment problems, such as a declined card. When the player fixes the payment, subscription purchases are queried again, and `onPurchaseUpdated` reports the subscription without `suspended`. It does nothing on iOS, which shows these messages by itself.

Events

Each event is a static `FunctionSubscribe` member; subscribe with `REGISTER_EVENT` in C++ or `RegisterEvent` in Lua (see Events (p. 221)). C++ handlers take the parameters by value, exactly as listed.

Event	Parameters	When
`onInitialized`	`BillingResponse response, std::string message`	The store connection was set up, or failed
`onDisconnected`	—	Android: the store connection was lost; the next call reconnects
`onProductsQueried`	`BillingResponse response, std::string message`	Product details arrived; read them with `getProducts()`
`onPurchaseUpdated`	`PurchaseDetails purchase`	A purchase is new, changed, or reported by `queryPurchases`. Grant it when `PURCHASED`. On iOS, a refund or an ended subscription also arrives here, with state `UNSPECIFIED`
`onPurchaseFailed`	`std::string productId, BillingResponse response, std::string message`	A purchase screen failed or was canceled. `productId` is empty when unknown
`onPurchasesQueried`	`ProductType productType, BillingResponse response, std::string message`	`queryPurchases`, or one type of `restorePurchases`, finished after its `onPurchaseUpdated` events
`onPurchaseAcknowledged`	`std::string purchaseToken, BillingResponse response, std::string message`	A purchase was acknowledged
`onPurchaseConsumed`	`std::string purchaseToken, BillingResponse response, std::string message`	A purchase was consumed and can be bought again

Types

These structs are also Lua classes with the same fields. Lists are Lua tables.

ProductDetails

A product as the store describes it, from `queryProducts`.

Field	Type	Meaning
`productId`	std::string	The product's id in the Play Console or App Store Connect
`type`	ProductType	`INAPP` or `SUBS`
`title`	std::string	Title as Google Play shows it, which may include the app name. On iOS, the display name
`name`	std::string	The product's name, for a store screen. On iOS, the display name
`description`	std::string	Description
`price`	std::string	Formatted price with its currency symbol. For a subscription, the recurring price of the plan `purchase()` buys without an offer token
`priceMicros`	long long	The price in millionths of the currency
`currencyCode`	std::string	ISO 4217 currency code
`offers`	std::vector<ProductOffer>	Subscription plans and offers, or the purchase options of a one-time product. On iOS, one entry with no token, holding the pricing phases

ProductOffer

A subscription's base plan or offer, or a one-time product's purchase option.

Field	Type	Meaning
`offerToken`	std::string	What `purchase()` takes to buy this plan or offer
`offerId`	std::string	The offer's id; empty for a base plan
`basePlanId`	std::string	The base plan's id, or the purchase option's id of a one-time product
`tags`	std::vector<std::string>	Tags set in the Play Console
`pricingPhases`	std::vector<PricingPhase>	The phases a buyer goes through, like a free trial and then the regular price

PricingPhase

Field	Type	Meaning
`price`	std::string	Formatted price with its currency symbol
`priceMicros`	long long	The price in millionths of the currency
`currencyCode`	std::string	ISO 4217 currency code
`billingPeriod`	std::string	ISO 8601 period, like `P1M` for a month. Empty for one-time products
`billingCycleCount`	int	Cycles a `FINITE_RECURRING` phase lasts
`recurrenceMode`	RecurrenceMode	How the phase repeats

PurchaseDetails

Field	Type	Meaning
`orderId`	std::string	Google Play's order id. On iOS, the transaction id
`productId`	std::string	The first product of the purchase
`productIds`	std::vector<std::string>	Every product of the purchase
`productType`	ProductType	`INAPP` or `SUBS`
`purchaseToken`	std::string	Identifies the purchase; what acknowledging, consuming and server checks take. On iOS, the original transaction id, which renewals keep. Empty for a purchase waiting for Ask to Buy
`purchaseTime`	long long	When it was bought, in milliseconds since the epoch
`state`	PurchaseState	Grant only when `PURCHASED`
`quantity`	int	Units bought at once, for products that allow several
`acknowledged`	bool	Already acknowledged or consumed. On iOS, its transaction is finished
`autoRenewing`	bool	A subscription that renews; false once canceled
`suspended`	bool	A subscription on hold for a payment problem. Do not grant it
`restored`	bool	Reported by `queryPurchases`, not by a purchase screen
`packageName`	std::string	The app's package name, or bundle identifier on iOS
`obfuscatedAccountId`	std::string	The id set with `setObfuscatedAccountId`. On iOS, the `appAccountToken` UUID, in lower case
`obfuscatedProfileId`	std::string	The id set with `setObfuscatedProfileId`. Empty on iOS
`originalJson`	std::string	The purchase data Google Play signed, to verify on a server. On iOS, the transaction JSON
`signature`	std::string	Its signature. On iOS, the transaction's JWS, which a server verifies with Apple's App Store Server Library or API

Enumerations

ProductType

- **INAPP** — A one-time product, consumable or not. Also an iOS non-renewing subscription, which the store never expires: work out its end from `purchaseTime`.
- **SUBS** — An auto-renewing subscription.

PurchaseState

- **UNSPECIFIED** — Unknown state. On iOS, also a refunded purchase or an ended subscription.
- **PURCHASED** — Paid. Grant it and settle it.
- **PENDING** — Waiting for a payment or an approval made outside the app, like Ask to Buy on iOS. Do not grant it yet.

BillingResponse

Google Play Billing's response codes, with the same values. iOS maps StoreKit's errors to them.

Value	Code	Meaning
`SERVICE_TIMEOUT`	-3	Google Play took too long to answer
`FEATURE_NOT_SUPPORTED`	-2	The device's Google Play does not support the request
`SERVICE_DISCONNECTED`	-1	Not connected: call `initialize()` first, or try again
`OK`	0	Success
`USER_CANCELED`	1	The player closed the purchase screen
`SERVICE_UNAVAILABLE`	2	Google Play cannot be reached for now, often for lack of network
`BILLING_UNAVAILABLE`	3	No billing here: another platform, a build without it, or an unsupported Play account
`ITEM_UNAVAILABLE`	4	The product cannot be bought, or was not queried
`DEVELOPER_ERROR`	5	Invalid arguments
`FATAL_ERROR`	6	Google Play's `ERROR`: an internal failure
`ITEM_ALREADY_OWNED`	7	The player owns it already; query purchases to restore it
`ITEM_NOT_OWNED`	8	The player does not own it
`NETWORK_ERROR`	12	A network failure during the operation

RecurrenceMode

- **INFINITE_RECURRING** — Repeats until the subscription is canceled.
- **FINITE_RECURRING** — Repeats `billingCycleCount` times.
- **NON_RECURRING** — Charged once.

SubscriptionReplacementMode

How `changeSubscription` bills the time left on the old plan.

- **WITH_TIME_PRORATION** — The change applies now, and the time left is credited toward the new plan.
- **CHARGE_PRORATED_PRICE** — The change applies now and the billing date stays; the price difference for the time left is charged. Upgrades only.
- **WITHOUT_PRORATION** — The change applies now, and the new price is charged from the next billing date.
- **CHARGE_FULL_PRICE** — The change applies now, the new plan is charged in full, and the time left is credited.

- **DEFERRED** — The change applies when the old plan renews.
- **KEEP_EXISTING** — The old plan's payment schedule carries over. `offerToken` is ignored.

See also

- Monetization (p. 342) — setup, testing, and a complete store script
- AdMob (p. 610) and WebPortal (p. 626)

WebPortal

C++ type: `WebPortal` (static) · **Header:** `WebPortal.h`

Description

The SDK of a web game portal, behind one API: ads, gameplay and loading events, and cloud saves. A web export builds in the portal chosen with **Game Portal** in the project's Web settings (p. 60): CrazyGames, Poki, GameDistribution, Yandex Games or YouTube Playables. Without one, and on every other platform, the calls do nothing and ads and saves report errors.

Results arrive as events at the start of a frame. The engine pauses while an ad plays or the portal pauses the game, and resumes after. See Monetization → Web portals (p. 346) for the flow, what each portal supports, and how to test locally.

C++

```
REGISTER_EVENT(WebPortal::onAdFinished, onAdFinished);
WebPortal::initialize();
WebPortal::loadingStop();

void Game::onAdFinished(WebPortalAdType type) {
    if (type == WebPortalAdType::REWARDED) lives++;
}
```

Lua

```
RegisterEvent(self, WebPortal.onAdFinished, "onAdFinished")
WebPortal.initialize()
WebPortal.loadingStop()

function Game:onAdFinished(adType)
    if adType == WebPortalAdType.REWARDED then
        self.lives = self.lives + 1
    end
end
```

Methods

Type	Name	Langs
static WebPortalType	getPortal	C++ | Lua
static void	initialize	C++ | Lua
static WebPortalEnvironment	getEnvironment	C++ | Lua
static bool	isAvailable	C++ | Lua
static void	requestAd	C++ | Lua
static void	gameplayStart	C++ | Lua
static void	gameplayStop	C++ | Lua
static void	loadingStart	C++ | Lua
static void	loadingStop	C++ | Lua
static void	happytime	C++ | Lua
static void	loadData	C++ | Lua
static void	saveData	C++ | Lua

Callback events

Callback	Name	Languages
void(WebPortalEnvironment)	onInitialized	C++ | Lua

Callback	Name	Languages
void(WebPortalAdType)	onAdStarted	C++ | Lua
void(WebPortalAdType)	onAdFinished	C++ | Lua
void(WebPortalAdType, std::string, std::string)	onAdError	C++ | Lua
void(std::string)	onDataLoaded	C++ | Lua
void(std::string)	onDataLoadFailed	C++ | Lua
void(std::string)	onDataSaveFailed	C++ | Lua

Method details

getPortal

- static WebPortalType **getPortal**()

The portal this build includes, or `NONE`. It reflects the build, not where the page runs: see `getEnvironment()` for that.

initialize

- static void **initialize**()

Loads the portal's SDK and starts it. Call it once at startup, as some portals pause or mute the game from the start. `onInitialized(environment)` reports where the game runs.

Calls made after `initialize()` wait for the SDK to start. Calls made before it are dropped: ads report `unavailable` and saves fail.

getEnvironment / isAvailable

- static WebPortalEnvironment **getEnvironment**()
- static bool **isAvailable**()

Where the game runs, known once `onInitialized` fired. `isAvailable()` is true for `LOCAL` and `PORTAL`, when the SDK works.

requestAd

- static void **requestAd**(WebPortalAdType type)

Asks the portal for an ad: `MIDGAME` at a natural break, or `REWARDED` when the player chose to watch one for a reward. `onAdStarted` and `onAdFinished` report an ad that played, and `onAdError(type, code, message)` one that did not. Grant a reward only in `onAdFinished`.

Portals decide when a midgame ad really plays, so a request may show none and report the code `unfilled`. The engine pauses while the ad plays. See ad error codes (p. 346).

gameplayStart / gameplayStop

- static void **gameplayStart**()
- static void **gameplayStop**()

Tell the portal when the player is playing: start at play and on resume, stop at every break, such as menus, pauses, and the end of a level. Portals use it to time their ads. CrazyGames, Poki and Yandex Games use it; the others ignore it.

loadingStart / loadingStop

- static void **loadingStart**()
- static void **loadingStop**()

Bracket loading screens. Call `loadingStop()` once the game can be played: Yandex Games and YouTube Playables require it. CrazyGames and Poki also use `loadingStart()`.

happytime

- static void **happytime**()

Marks a happy moment, like a level completed or a record beaten. CrazyGames and Poki use it.

loadData / saveData

- static void **loadData**()
- static void **saveData**(const std::string& data)

The cloud save of YouTube Playables, kept in the player's account as one string. `loadData()` reports `onDataLoaded(data)`, with an empty string when nothing was saved yet, or `onDataLoadFailed(message)`. `saveData()` reports only failures, through `onDataSaveFailed(message)`. YouTube refuses saves made before a `loadData()`.

The other portals have no cloud save: `loadData()` and `saveData()` report a failure.

Events

Each event is a static `FunctionSubscribe` member; subscribe with `REGISTER_EVENT` in C++ or `RegisterEvent` in Lua (see Events (p. 221)). C++ handlers take the parameters by value, exactly as listed.

Event	Parameters	When
`onInitialized`	`WebPortalEnvironment environment`	The SDK started, or could not
`onAdStarted`	`WebPortalAdType type`	An ad started. The engine pauses until it ends. YouTube Playables gives no start signal, so it never fires there
`onAdFinished`	`WebPortalAdType type`	An ad finished; grant the reward of a rewarded ad
`onAdError`	`WebPortalAdType type, std::string code, std::string message`	No ad played, or it failed. See the codes below
`onDataLoaded`	`std::string data`	The cloud save loaded; saving works from now on
`onDataLoadFailed`	`std::string message`	The cloud save could not be loaded
`onDataSaveFailed`	`std::string message`	Saving to the cloud failed

`onAdError` codes:

Code	Meaning
`unfilled`	No ad played: the portal had none, chose not to show one now, or another ad is playing
`unavailable`	No portal SDK: a build without a portal, `initialize()` not called, or the portal refused to start here
`other`	The ad started but failed, or a rewarded ad was closed before its reward
A CrazyGames code	CrazyGames passes its own codes, like `adCooldown`

Enumerations

WebPortalType

- **NONE** — No portal in this build.

- **CRAZYGAMES** — CrazyGames, SDK v3.
- **POKI** — Poki, SDK v2.
- **GAMEDISTRIBUTION** — GameDistribution. Needs the **Portal Game ID** project setting.
- **YANDEX** — Yandex Games.
- **YOUTUBE** — YouTube Playables.

WebPortalAdType

- **MIDGAME** — An ad at a break in the game: CrazyGames' midgame ad, Poki's commercial break, an interstitial or fullscreen ad elsewhere.
- **REWARDED** — An ad the player chose to watch for a reward.

WebPortalEnvironment

- **UNINITIALIZED** — `initialize()` has not finished.
- **DISABLED** — No portal in this build, its SDK failed to load, or the portal does not run on this host. Calls do nothing.
- **LOCAL** — `localhost`, `127.0.0.1` or `[::1]`, where most portals show test ads.
- **PORTAL** — Any other host, normally the portal's.

See also

- Monetization → Web portals (p. 346) — setup, portal differences, and local testing
- AdMob (p. 610) and InAppPurchase (p. 618)
- Building for HTML5 (p. 760)

Scripting

ScriptBase

C++ type: `ScriptBase`

Description

The base class for all C++ gameplay scripts attached to entities. To create a component script in C++, inherit from `ScriptBase` and register engine events with `REGISTER_ENGINE_EVENT` (there are no virtual `update` overrides).

In Lua, scripts are plain Lua tables with lifecycle functions (`init`, `onUpdate`, etc.) — the `ScriptBase` class itself is not exposed directly to Lua. See Creating Scripts (p. 199) for the full guide.

Properties

Type	Name	Langs
Scene*	scene	C++
Entity	entity	C++

Methods

Type	Name	Langs
Scene*	getScene	C++
Entity	getEntity	C++

Property details

scene / entity

- Scene* **getScene**() const
- Entity **getEntity**() const

The scene this script belongs to, and the entity it is attached to. Use these to access Object (p. 455) transforms, add/remove components, or query the ECS.

C++

```
class PlayerController : public ScriptBase {
public:
    PlayerController(Scene* scene, Entity entity)
        : ScriptBase(scene, entity) {
        REGISTER_ENGINE_EVENT(onUpdate);
    }

    ~PlayerController() {
        UNREGISTER_ENGINE_EVENT(onUpdate);
    }

    void onUpdate() {
        Object obj(getScene(), getEntity());
        if (Input::isKeyPressed(D_KEY_W)) {
            obj.setPosition(obj.getPosition() + Vector3(0, 0, -5.0f * Engine::getDeltatime()));
        }
    }
};
```

See Creating Scripts (p. 199) for instructions on attaching scripts to entities in the editor.

Math

AABB

C++ type: `AABB`

Description

An **Axis-Aligned Bounding Box** defined by a minimum and maximum Vector3 (p. 660) corner. All edges are parallel to the world axes, making AABB overlap tests extremely fast. AABBs are used throughout Doriax for mesh bounds queries, frustum culling, and physics broad-phase.

For rotated bounding volumes use OBB (p. 639).

Properties

Type	Name	Langs
Vector3	minimum	C++ | Lua
Vector3	maximum	C++ | Lua

Static constants

Name	Description
`ZERO`	A valid AABB with all extents at zero

Methods

Type	Name	Langs
void	setExtents	C++ | Lua
Vector3	getCenter	C++ | Lua
Vector3	getSize	C++ | Lua
Vector3	getHalfSize	C++ | Lua
AABB&	merge	C++ | Lua
AABB&	transform	C++ | Lua
AABB	intersection	C++ | Lua
bool	intersects	C++ | Lua
bool	contains	C++ | Lua
float	distance	C++ | Lua
float	squaredDistance	C++ | Lua
float	volume	C++ | Lua
void	scale	C++ | Lua
OBB (p. 639)	getOBB	C++
Vector3	getCorner	C++ | Lua
const Vector3*	getCorners	C++ | Lua
void	setNull / setInfinite / setFinite	C++ | Lua
bool	isNull / isInfinite / isFinite	C++ | Lua

Property details

minimum / maximum

- *Setter*: void **setMinimum**(const Vector3& vec) / individual axis setters
- *Getter*: const Vector3& **getMinimum**() const
- *Setter*: void **setMaximum**(const Vector3& vec)
- *Getter*: const Vector3& **getMaximum**() const

The two corner points of the box in world space.

Method details

setExtents

- void **setExtents**(const Vector3& min, const Vector3& max)
- void **setExtents**(float mx, float my, float mz, float Mx, float My, float Mz)

Sets both minimum and maximum in one call.

getCenter / getSize / getHalfSize

- Vector3 **getCenter**() const
- Vector3 **getSize**() const
- Vector3 **getHalfSize**() const

`getCenter()` returns the midpoint. `getSize()` returns `max - min`. `getHalfSize()` returns half the size — convenient for physics and intersection tests.

merge

- AABB& **merge**(const AABB& rhs)
- AABB& **merge**(const Vector3& point)

Expands this AABB to also contain `rhs` or the given `point`.

transform

- AABB& **transform**(const Matrix4& matrix)

Transforms the AABB by `matrix`. Note: an AABB after transformation is re-fitted, which may be larger than the OBB would be.

intersection

- AABB **intersection**(const AABB& b2) const

Returns the AABB formed by the overlap of this box and `b2`. Returns a null AABB if there is no overlap.

intersects

- bool **intersects**(const AABB& b2) const
- bool **intersects**(const OBB (p. 639)& obb) const
- bool **intersects**(const Plane (p. 642)& p) const

- bool **intersects**(const Sphere (p. 654)& sp) const
- bool **intersects**(const Vector3 (p. 660)& v) const

Overlap tests against other volumes or a point.

C++

```
AABB playerBounds = player.getWorldAABB();
AABB triggerZone(Vector3(-2,-1,-2), Vector3(2,1,2));
if (playerBounds.intersects(triggerZone)) {
    // player entered trigger
}
```

Lua

```
local playerBounds = player:getWorldAABB()
local triggerZone = AABB(Vector3(-2,-1,-2), Vector3(2,1,2))
if playerBounds:intersects(triggerZone) then
    -- player entered trigger
end
```

contains

- bool **contains**(const Vector3& v) const
- bool **contains**(const AABB& other) const

Returns `true` if this AABB fully contains the given point or box.

distance / squaredDistance

- float **distance**(const Vector3& v) const
- float **squaredDistance**(const Vector3& v) const

Distance from a point to the nearest surface of this box. Returns `0` if the point is inside.

volume

- float **volume**() const

Returns the volume `(max.x-min.x) * (max.y-min.y) * (max.z-min.z)`.

scale

- void **scale**(const Vector3& s)

Scales the AABB from its centre by the given factor.

getOBB

- OBB (p. 639) **getOBB**() const

Converts this AABB to an OBB (p. 639) with axis-aligned orientation.

getCorner / getCorners

- Vector3 **getCorner**(AABB::CornerEnum corner) const
- const Vector3* **getCorners**() const

`getCorner()` returns one corner, named by `AABB.CornerEnum` (`FAR_LEFT_BOTTOM`, `NEAR_RIGHT_TOP`, and so on). `getCorners()` returns all eight, in the order of those enum values. Lua gets them as a table of copies, so

`corners[c + 1]` is `getCorner(c)`. Only a finite box has corners: in C++ `getCorners()` asserts on a null or infinite box, and Lua gets an empty table.

C++

```
AABB bounds = player.getWorldAABB();
const Vector3* corners = bounds.getCorners();
for (int i = 0; i < 8; i++) {
    Log::print("%s", corners[i].toString().c_str());
}
```

Lua

```
local bounds = player:getWorldAABB()
for i, corner in ipairs(bounds:getCorners()) do
    Log.print(tostring(corner))
end
```

null / infinite / finite

- void **setNull**() / bool **isNull**()
- void **setInfinite**() / bool **isInfinite**()
- void **setFinite**() / bool **isFinite**()

An AABB can be in one of three states: *null* (undefined, no extent), *finite* (normal bounding box), or *infinite* (the entire space). Intersection tests respect these states.

Angle

C++ type: `Angle` (static utility class)

Description

A static utility class providing angle conversion functions between degrees, radians, and Doriax's *default* unit, which is controlled by `Engine::useDegrees`.

Because the "default" unit changes depending on the engine configuration, these helpers make it easy to write code that works correctly regardless of how the engine is configured.

Methods

Type	Name	Langs
static float	radToDefault	C++ | Lua
static float	degToDefault	C++ | Lua
static float	defaultToRad	C++ | Lua
static float	defaultToDeg	C++ | Lua
static float	radToDeg	C++ | Lua
static float	degToRad	C++ | Lua

Method details

radToDefault / degToDefault

- static float **radToDefault**(float radians)
- static float **degToDefault**(float degrees)

Convert a value in radians or degrees to the engine's current default angle unit. If `Engine::useDegrees` is `true`, `radToDefault()` converts radians → degrees and `degToDefault()` is a no-op.

C++

```cpp
// Rotate 45° regardless of engine angle setting
float angle = Angle::degToDefault(45.0f);
obj.setRotation(Quaternion(0, angle, 0));
```

Lua

```lua
local angle = Angle.degToDefault(45)
obj:setRotation(Quaternion(0, angle, 0))
```

defaultToRad / defaultToDeg

- static float **defaultToRad**(float angle)
- static float **defaultToDeg**(float angle)

Convert from the engine's default unit to radians or degrees.

C++

```cpp
float deg = Angle::defaultToDeg(obj.getRotation().getYaw());
```

Lua

```lua
local deg = Angle.defaultToDeg(obj.rotation:getYaw())
```

radToDeg / degToRad

- static float **radToDeg**(float radians)
- static float **degToRad**(float degrees)

Direct conversion between radians and degrees, independent of the engine's `useDegrees` setting.

C++

```cpp
float rad = Angle::degToRad(180.0f);  // π
```

Lua

```lua
local rad = Angle.degToRad(180)  -- π
```

Matrix3

3×3 matrix.

Methods

Name	Languages
`rotateMatrix`	C++ | Lua
`rotateXMatrix`	C++ | Lua
`rotateYMatrix`	C++ | Lua
`rotateZMatrix`	C++ | Lua
`scaleMatrix`	C++ | Lua
`row`	C++ | Lua
`column`	C++ | Lua
`set`	C++ | Lua
`get`	C++ | Lua
`setRow`	C++ | Lua
`setColumn`	C++ | Lua
`isValid`	C++ | Lua
`identity`	C++ | Lua
`calcInverse`	C++ | Lua
`inverse`	C++ | Lua
`transpose`	C++ | Lua
`decomposeQDU`	C++ | Lua

Matrix4

4×4 transform matrix.

Methods

Name	Languages
translateMatrix	C++ | Lua
rotateMatrix	C++ | Lua
rotateXMatrix	C++ | Lua
rotateYMatrix	C++ | Lua
rotateZMatrix	C++ | Lua
scaleMatrix	C++ | Lua
lookAtMatrix	C++ | Lua
frustumMatrix	C++ | Lua
orthoMatrix	C++ | Lua
perspectiveMatrix	C++ | Lua
row	C++ | Lua
column	C++ | Lua
set	C++ | Lua
get	C++ | Lua
setRow	C++ | Lua
setColumn	C++ | Lua
isValid	C++ | Lua
identity	C++ | Lua
translateInPlace	C++ | Lua
linear	C++ | Lua
inverse	C++ | Lua
transpose	C++ | Lua
determinant	C++ | Lua
decomposeStandard	C++ | Lua
decomposeQDU	C++ | Lua
decompose	C++ | Lua

OBB

C++ type: `OBB`

Description

An **Oriented Bounding Box** — a box whose axes can be rotated arbitrarily in world space. OBBs provide tighter fit than AABB (p. 632)s for non-axis-aligned objects at the cost of more complex intersection tests.

Properties

Type	Name	Langs
Vector3	center	C++ | Lua
Vector3	halfExtents	C++ | Lua
Quaternion	orientation	C++ | Lua

Methods

Type	Name	Langs
void	setAxes	C++ | Lua
void	setOrientation	C++
Quaternion	getOrientation	C++
void	transform	C++ | Lua
AABB	toAABB	C++ | Lua
Vector3	getCorner	C++ | Lua
const Vector3*	getCorners	C++ | Lua
bool	intersects	C++ | Lua
bool	contains	C++ | Lua
float	distance	C++ | Lua
float	squaredDistance	C++ | Lua
Vector3	closestPoint	C++ | Lua
float	volume	C++ | Lua
void	enclose	C++ | Lua
std::string	toString	C++ | Lua

Property details

center / halfExtents

- *Setter/Getter*: **setCenter** / **getCenter**, **setHalfExtents** / **getHalfExtents**

The box centre and the half-extents along each local axis.

Method details

setAxes

- void **setAxes**(const Vector3& axisX, const Vector3& axisY, const Vector3& axisZ)
- void **setAxes**(const Quaternion& orientation)

Sets the three orthonormal local axes of the OBB. Can also be specified as a quaternion.

setOrientation / getOrientation

- void **setOrientation**(const Quaternion& orientation)
- Quaternion **getOrientation**() const

The rotation of the box as a quaternion. Lua reads and writes it through the `orientation` property.

transform

- void **transform**(const Matrix4& matrix)
- void **transform**(const Vector3& translate, const Quaternion& rotate, const Vector3& scale)

Applies a transformation to the OBB.

toAABB

- AABB (p. 632) **toAABB**() const

Returns the axis-aligned bounding box that fully encloses this OBB.

getCorner / getCorners

- Vector3 **getCorner**(OBB::CornerEnum corner) const
- const Vector3* **getCorners**() const

`getCorner()` returns one corner, named by `OBB.CornerEnum` (`FAR_LEFT_BOTTOM`, `NEAR_RIGHT_TOP`, and so on). `getCorners()` returns all eight, in the order of those enum values. Lua gets them as a table of copies, so `corners[c + 1]` is `getCorner(c)`.

intersects

- bool **intersects**(const OBB& other) const
- bool **intersects**(const AABB& aabb) const
- bool **intersects**(const Sphere& sphere) const
- bool **intersects**(const Plane& plane) const
- bool **intersects**(const Vector3& point) const

Intersection tests using the Separating Axis Theorem (SAT).

contains

- bool **contains**(const Vector3& point) const
- bool **contains**(const OBB& other) const

Returns `true` if this OBB fully contains the given point or box.

distance / squaredDistance

- float **distance**(const Vector3& point) const
- float **squaredDistance**(const Vector3& point) const

Distance from a point to the nearest surface of this OBB.

closestPoint

- Vector3 **closestPoint**(const Vector3& point) const

Returns the point on or inside the OBB closest to `point`.

volume

- float **volume**() const

Returns `8 * halfExtents.x * halfExtents.y * halfExtents.z`.

enclose

- void **enclose**(const OBB& other)
- void **enclose**(const Vector3& point)

Expands the OBB to also contain another OBB or a point.

toString

- std::string **toString**() const

Returns a human-readable description of the OBB.

Plane

C++ type: `Plane`

Description

An infinite plane in 3D space represented in the form `n·x + d = 0`, where `n` is the unit normal vector and `d` is the signed distance from the origin.

Properties

Type	Name	Default	Langs
Vector3	normal	`(0,0,0)`	C++ | Lua
float	d	`0.0`	C++ | Lua

Methods

Type	Name	Langs
Side	getSide	C++ | Lua
float	getDistance	C++ | Lua
void	redefine	C++ | Lua
Vector3	projectVector	C++ | Lua
Plane&	normalize	C++ | Lua
Plane	normalized	C++ | Lua

Enumerations

Side

Classifies the relative position of a point or volume:

- **NO_SIDE** — The object is on or spans both sides.
- **POSITIVE_SIDE** — The object is in front of the plane (same direction as `normal`).
- **NEGATIVE_SIDE** — The object is behind the plane.
- **BOTH_SIDE** — The object spans both sides.

Method details

getSide

- Side **getSide**(const Vector3& rkPoint) const
- Side **getSide**(const Vector3& centre, const Vector3& halfSize) const
- Side **getSide**(const AABB& rkBox) const
- Side **getSide**(const OBB& obb) const

Determines which side of the plane a point or volume is on. Useful for frustum culling and portal rendering.

getDistance

- float **getDistance**(const Vector3& rkPoint) const

Returns the signed distance from `rkPoint` to the plane. Positive values are on the `normal` side.

redefine

- void **redefine**(const Vector3& rkPoint0, const Vector3& rkPoint1, const Vector3& rkPoint2)
- void **redefine**(const Vector3& rkNormal, const Vector3& rkPoint)

Recomputes the plane from three non-collinear points or from a normal and a point.

projectVector

- Vector3 **projectVector**(const Vector3& v) const

Projects `v` onto the plane, returning the component of `v` tangent to the plane.

normalize / normalized

- Plane& **normalize**()
- Plane **normalized**() const

Normalises the plane equation so that `normal` has unit length.

Quaternion

C++ type: `Quaternion`

Description

Represents a rotation in 3D space using the quaternion formalism. Quaternions avoid gimbal lock, support smooth slerp interpolation, and are more numerically stable than Euler-angle representations for concatenated rotations.

In Doriax, all Object (p. 455) rotations are stored as quaternions. Euler angles, axis-angle, and matrix forms can be converted to/from quaternions using the provided constructors and instance methods.

> **Lua components are writable**
> Lua can read and assign `w`, `x`, `y`, and `z` directly. `obj.rotation` returns a quaternion value, so assign a mutated value back to persist it. Component edits can produce a non-unit quaternion; call `normalize()` before using it as a rotation.

Properties

Type	Name	Default	Langs
float	w	`1.0`	C++ | Lua
float	x	`0.0`	C++ | Lua
float	y	`0.0`	C++ | Lua
float	z	`0.0`	C++ | Lua

Static constants

Name	Value
`IDENTITY`	`(1, 0, 0, 0)` — no rotation

Methods

Type	Name	Langs
void	fromEulerAngles	C++ | Lua
void	fromAngleAxis	C++ | Lua
void	fromAngle	C++ | Lua
void	fromAxes	C++ | Lua
Matrix4	getRotationMatrix	C++ | Lua
Vector3	getEulerAngles	C++ | Lua
Vector3	xAxis / yAxis / zAxis	C++ | Lua
float	dot	C++ | Lua
float	norm	C++ | Lua
Quaternion	inverse	C++ | Lua
Quaternion	unitInverse	C++ | Lua
Quaternion&	normalize	C++ | Lua
Quaternion	normalized	C++ | Lua
float	normalizeL	C++ | Lua
float	getRoll / getPitch / getYaw	C++ | Lua
static Quaternion	slerp	C++ | Lua

Type	Name	Langs
static Quaternion	nlerp	C++ | Lua
static Quaternion	squad	C++ | Lua
static Quaternion	lookRotation	C++ | Lua
std::string	toString	C++ | Lua (`tostring`)

Enumerations

RotationOrder

Controls the order in which Euler-angle rotations are applied when constructing a quaternion:

- **XYZ** — Roll, then Pitch, then Yaw
- **XZY**, **YXZ**, **YZX**, **ZXY**, **ZYX** — Other standard combinations

The most common order in Doriax is **ZYX** — that is the default used by `Quaternion(xAngle, yAngle, zAngle)` and `Object::setRotation(x, y, z)`.

Method details

fromEuler

- void **fromEulerAngles**(float xAngle, float yAngle, float zAngle, const RotationOrder& order)

Constructs the quaternion from Euler angles. Angles are in degrees if `Engine::useDegrees` is `true`.

C++

```
Quaternion q;
q.fromEulerAngles(0.0f, 45.0f, 0.0f, RotationOrder::ZYX);
obj.setRotation(q);
```

Lua

```
local q = Quaternion()
q:fromEulerAngles(0, 45, 0, RotationOrder.ZYX)
obj.rotation = q
```

There is also a shorthand constructor (defaults to **ZYX** order):

C++

```
Quaternion q(0.0f, 45.0f, 0.0f);  // ZYX order, useDegrees aware
```

Lua

```
local q = Quaternion(0, 45, 0)
```

fromAngleAxis

- void **fromAngleAxis**(float angle, const Vector3& rkAxis)

Constructs the quaternion from an angle-axis representation. Useful for arbitrary-axis rotations.

C++

```
Quaternion q;
q.fromAngleAxis(90.0f, Vector3::UNIT_Y);  // 90° around Y
```

Lua

```lua
local q = Quaternion()
q:fromAngleAxis(90, Vector3.UNIT_Y)
```

fromAngle

• void **fromAngle**(float angle)

Constructs a 2D-style rotation around the Z axis. Convenience for 2D scenes.

fromAxes

• void **fromAxes**(const Vector3& xaxis, const Vector3& yaxis, const Vector3& zaxis)

Builds the quaternion from an orthonormal basis of three axes.

getRotationMatrix

• Matrix4 **getRotationMatrix**() const

Converts this quaternion to a `Matrix4` rotation matrix.

getEuler

• Vector3 **getEulerAngles**(const RotationOrder& order) const

Extracts Euler angles from this quaternion in the specified rotation order.

xAxis / yAxis / zAxis

• Vector3 **xAxis**() const
• Vector3 **yAxis**() const
• Vector3 **zAxis**() const

Returns the local X, Y, or Z axis expressed in world space — equivalent to rotating the corresponding unit vector by this quaternion.

dot

• float **dot**(const Quaternion& rkQ) const

Returns the dot product between two quaternions. Values near `1` indicate similar orientations; near `-1` indicate opposite orientations.

norm

• float **norm**() const

Returns the squared length of the quaternion. A normalised quaternion has `norm()` = `1`.

inverse / unitInverse

• Quaternion **inverse**() const
• Quaternion **unitInverse**() const

`inverse()` returns the inverse (safe for any non-zero quaternion). `unitInverse()` is faster but only correct for unit quaternions.

normalize / normalized

- Quaternion& **normalize**()
- Quaternion **normalized**() const

`normalize()` scales the quaternion in-place to unit length. `normalized()` returns a new unit-length copy.

normalizeL

- float **normalizeL**()

Normalises in-place and returns the original norm.

getRoll / getPitch / getYaw

- float **getRoll**() const
- float **getPitch**() const
- float **getYaw**() const

Extracts the roll (Z), pitch (X), or yaw (Y) angle component.

slerp

- static Quaternion **slerp**(float t, const Quaternion& q1, const Quaternion& q2)
- static Quaternion **slerp**(float t, const Quaternion& q1, const Quaternion& q2, bool shortestPath)

Spherical linear interpolation. `t=0` returns `q1`, `t=1` returns `q2`. When `shortestPath` is `true`, the interpolation always takes the shorter arc.

C++

```
Quaternion smoothRot = Quaternion::slerp(0.5f, startRot, endRot, true);
```

Lua

```
local smoothRot = Quaternion.slerp(0.5, startRot, endRot, true)
```

nlerp

- static Quaternion **nlerp**(float fT, const Quaternion& rkP, const Quaternion& rkQ)
- static Quaternion **nlerp**(float fT, const Quaternion& rkP, const Quaternion& rkQ, bool shortestPath)

Normalised linear interpolation. Cheaper than `slerp()` but not constant-speed; suitable for small angular differences or when speed uniformity is not critical.

squad

- static Quaternion **squad**(float fT, const Quaternion& rkP, const Quaternion& rkA, const Quaternion& rkB, const Quaternion& rkQ)

Spherical cubic (SQUAD) interpolation between two quaternions `rkP` and `rkQ` with inner control points `rkA` and `rkB`. Used for smooth keyframe animation.

lookRotation

- static Quaternion **lookRotation**(const Vector3& forward)
- static Quaternion **lookRotation**(const Vector3& forward, const Vector3& up)

Returns a rotation whose local **+Z** axis faces `forward`, keeping local +Y close to `up` (`Vector3::UNIT_Y` by default). A zero `forward` returns `IDENTITY`. Cameras look down **-Z**, so for a camera call disableTarget (p. 419) and pass `position - target`.

C++

```
obj.setRotation(Quaternion::lookRotation(target - obj.getPosition()));
```

Lua

```
obj.rotation = Quaternion.lookRotation(target - obj.position)
```

toString

- std::string **toString**() const

Returns a string like `"Quaternion(1.000000, 0.000000, 0.000000, 0.000000)"`.

Ray

C++ type: `Ray`

Description

A ray defined by an origin point and a direction/length vector. Used for raycasting — detecting which objects lie along the ray. Doriax supports raycasts against geometric volumes (AABB (p. 632), OBB (p. 639), Sphere (p. 654), Plane (p. 642)), against the physics simulation (Body2D (p. 541) / Body3D (p. 551)), and against the bounds of every mesh in a scene, which needs no physics at all.

The direction vector doubles as the *length* of the ray — objects beyond `origin + direction` are not reported as hits.

Properties

Type	Name	Langs
Vector3	origin	C++ | Lua
Vector3	direction	C++ | Lua

Methods

Type	Name	Langs
Vector3	getPoint	C++ | Lua
RayReturn (p. 716)	intersects	C++ | Lua

Property details

origin / direction

- *Setter*: void **setOrigin**(Vector3 point) / void **setDirection**(Vector3 direction)
- *Getter*: Vector3 **getOrigin**() const / Vector3 **getDirection**() const

`origin` is the start of the ray. `direction` is both the normalised direction and the maximum reach (its magnitude = ray length). Use `Camera::screenToRay()` to generate a ray from a screen pixel.

Method details

getPoint

- Vector3 **getPoint**(float distance) const

Returns `origin + direction * distance`. Hits report `distance` as a fraction of the ray, from 0 at `origin` to 1 at `origin + direction`, so `getPoint(hit.distance)` is the hit point.

intersects

Multiple overloads for different collision targets:

- RayReturn (p. 716) **intersects**(const Plane& plane) const
- RayReturn (p. 716) **intersects**(const AABB& box) const
- RayReturn (p. 716) **intersects**(const OBB& obb) const
- RayReturn (p. 716) **intersects**(const Sphere& sphere) const

- RayReturn (p. 716) **intersects**(const Body2D& body) const
- RayReturn (p. 716) **intersects**(const Body2D& body, size_t shape) const
- RayReturn (p. 716) **intersects**(const Body3D& body) const
- RayReturn (p. 716) **intersects**(const Body3D& body, size_t shape) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest, bool onlyStatic) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest, uint16_t categoryBits, uint16_t maskBits) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest, bool onlyStatic, uint16_t categoryBits, uint16_t maskBits) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest, Entity ignoreEntity) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest, const std::vector<Entity>& ignoreEntities) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest, bool onlyStatic, uint16_t categoryBits, uint16_t maskBits, Entity ignoreEntity) const
- RayReturn (p. 716) **intersects**(Scene* scene, RayFilter raytest, bool onlyStatic, uint16_t categoryBits, uint16_t maskBits, const std::vector<Entity>& ignoreEntities) const
- RayReturn (p. 716) **intersects**(Scene* scene, uint8_t broadPhaseLayer3D) const
- RayReturn (p. 716) **intersects**(Scene* scene, uint8_t broadPhaseLayer3D, uint16_t categoryBits, uint16_t maskBits) const
- RayReturn (p. 716) **intersects**(Scene* scene, uint8_t broadPhaseLayer3D, Entity ignoreEntity) const
- RayReturn (p. 716) **intersects**(Scene* scene, uint8_t broadPhaseLayer3D, const std::vector<Entity>& ignoreEntities) const
- RayReturn (p. 716) **intersects**(Scene* scene, uint8_t broadPhaseLayer3D, uint16_t categoryBits, uint16_t maskBits, Entity ignoreEntity) const
- RayReturn (p. 716) **intersects**(Scene* scene, uint8_t broadPhaseLayer3D, uint16_t categoryBits, uint16_t maskBits, const std::vector<Entity>& ignoreEntities) const

Returns a RayReturn (p. 716) struct. Test the result with `if (result)` or `result.hit`.

The `Body2D` overload returns the closest hit among the body's shapes. Both `Body2D` overloads return no hit for a body that is not in the physics world yet, and the shape overload also for an index past the body's shapes. A ray that starts inside a 2D shape doesn't hit that shape, in these overloads and in `BODY_2D` scene queries, so a ray cast from inside a shooter reaches what is in front of it.

Scene queries return the closest hit. RayFilter only selects what is tested — 2D bodies, 3D bodies or mesh bounds — it does not skip a specific entity. Checking `hit.body == entity` after the fact also fails, because that body is already the closest hit and the ray never continues past it. Pass the entity (or a list of entities) to `intersects` so those bodies are ignored and the next hit is returned. This works with every filter.

The `uint8_t broadPhaseLayer3D` overloads test 3D bodies on one broad-phase layer only. Layers go from 0 to 5 (`MAX_BROADPHASELAYER_3D` is 6); a higher one returns no hit.

The `Body2D` overloads exist only when the build has 2D physics, and the `Body3D` and `broadPhaseLayer3D` ones only with 3D physics — see Project Settings → Physics backends (p. 59). A `BODY_2D` or `BODY_3D` query against a disabled backend returns no hit.

C++

```cpp
// Screen-to-world raycast, skipping this script's own body
Ray ray = camera.screenToRay(mouseX, mouseY);
RayReturn hit = ray.intersects(scene, RayFilter::BODY_3D, entity);
if (hit) {
    Log::print("Hit body at distance: " + std::to_string(hit.distance));
}

// Ignore several bodies
RayReturn hit2 = ray.intersects(scene, RayFilter::BODY_3D, std::vector<Entity>{entity, other});
```

Lua

```lua
-- Screen-to-world raycast, skipping this script's own body
local ray = camera:screenToRay(mouseX, mouseY)
local hit = ray:intersects(scene, RayFilter.BODY_3D, self.entity)
if hit.hit then
    Log.print("Hit at " .. tostring(hit.distance))
end

-- Ignore several bodies
local hit2 = ray:intersects(scene, RayFilter.BODY_3D, {self.entity, other.entity})
```

Enumerations

RayFilter

Selects what a scene raycast tests. Use a separate `ignoreEntity` / `ignoreEntities` argument to skip specific entities.

- **BODY_2D** — Test against Body2D (p. 541) physics bodies.
- **BODY_3D** — Test against Body3D (p. 551) physics bodies.
- **BOUNDS** — Test against the world-space AABB (p. 632) of every mesh — models, shapes, sprites, tilemaps and terrain. Needs no physics backend. The hit is the box, not the surface, so it is coarser than a collider cast; `body` holds the mesh entity and `normal` the face of the box that was hit (zero when the ray starts inside a box, which hits at distance 0). `onlyStatic` and the category/mask bits are ignored.

Rect

C++ type: `Rect`

Description

An axis-aligned rectangle defined by an origin `(x, y)` and size `(width, height)`. Used throughout the engine for screen regions, texture atlas frames, viewport definitions, and UI layouts.

> **Lua components are writable**
> Lua can read and assign `x`, `y`, `width`, and `height` directly. Engine properties such as `sprite.textureRect` return rect values, so assign a mutated value back to persist it: `local r = sprite.textureRect; r.width = 64; sprite.textureRect = r.`

Properties

Type	Name	Default	Langs
float	x	`0.0`	C++ | Lua
float	y	`0.0`	C++ | Lua
float	width	`0.0`	C++ | Lua
float	height	`0.0`	C++ | Lua

Methods

Type	Name	Langs
void	setRect	C++ | Lua
Vector4	getVector	C++ | Lua
Rect&	fitOnRect	C++ | Lua
bool	contains	C++ | Lua
bool	isNormalized	C++ | Lua
bool	isZero	C++ | Lua
std::string	toString	C++ | Lua (`tostring`)

Property details

x / y / width / height

- *Getter/Setter*: float **getX**() / void **setX**(float x)
- *Getter/Setter*: float **getY**() / void **setY**(float y)
- *Getter/Setter*: float **getWidth**() / void **setWidth**(float width)
- *Getter/Setter*: float **getHeight**() / void **setHeight**(float height)

The four components of the rectangle.

C++

```
Rect r(10.0f, 20.0f, 100.0f, 50.0f);  // x=10, y=20, w=100, h=50
```

Lua

```
local r = Rect(10, 20, 100, 50)
```

Method details

setRect

- void **setRect**(float x, float y, float width, float height)
- void **setRect**(Rect rect)

Sets all four components at once.

getVector

- Vector4 **getVector**()

Returns the rect as a `Vector4(x, y, width, height)`.

fitOnRect

- Rect& **fitOnRect**(Rect rect)

Scales and positions this rect to fit inside `rect` while preserving the aspect ratio. Modifies in-place and returns `*this`.

contains

- bool **contains**(Vector2 point)

Returns `true` if `point` lies inside this rectangle (inclusive).

C++

```
Rect bounds(0, 0, 800, 600);
if (bounds.contains(Vector2(400, 300))) {
    // point is inside
}
```

Lua

```
local bounds = Rect(0, 0, 800, 600)
if bounds:contains(Vector2(400, 300)) then
    -- point is inside
end
```

isNormalized

- bool **isNormalized**() const

Returns `true` if all components are in the `[0, 1]` range, as expected for normalised UV or NDC coordinates.

isZero

- bool **isZero**() const

Returns `true` if all four components are zero.

toString

- std::string **toString**() const

Returns a human-readable string like `"Rect(10.000000, 20.000000, 100.000000, 50.000000)"`.

Sphere

C++ type: `Sphere`

Description

A bounding sphere defined by a `center` point and a `radius`. Used for fast intersection tests and as a coarse culling primitive.

Properties

Type	Name	Default	Langs
Vector3	center	`(0,0,0)`	C++ | Lua
float	radius	`0.0`	C++ | Lua

Methods

Type	Name	Langs
bool	contains	C++ | Lua
bool	intersects	C++ | Lua
void	merge	C++ | Lua
float	surfaceArea	C++ | Lua
float	volume	C++ | Lua
std::string	toString	C++ | Lua

Method details

contains

- bool **contains**(const Vector3& point) const

Returns `true` if `point` is inside (or on) the sphere.

---

intersects

- bool **intersects**(const Sphere& other) const
- bool **intersects**(const AABB& aabb) const
- bool **intersects**(const OBB& obb) const
- bool **intersects**(const Plane& plane) const
- bool **intersects**(const Vector3& v) const

Overlap test against another sphere, AABB, OBB, plane, or a point.

---

merge

- void **merge**(const Sphere& other)

Expands this sphere to also enclose `other`.

---

surfaceArea / volume

- float **surfaceArea**() const — returns `4 * π * r²`
- float **volume**() const — returns `(4/3) * π * r³`

toString

- std::string **toString**() const

Returns a human-readable string like `"Sphere(center=(0,0,0), radius=1)"`.

Vector2

C++ type: `Vector2`

Description

A 2D vector with `float` components `x` and `y`. Used extensively for 2D positions, sizes, UV coordinates, and directions throughout the engine.

All arithmetic operators (`+`, `-`, `*`, `/`) are available for both component-wise vector operations and scalar multiplication/division, and the number can also come first in a multiplication (`2 * v`). Comparison operators (`<`, `>`, `==`) are also supported.

> **Lua components are writable**
> Lua can read and assign `x` and `y` directly. Engine properties such as `layout.positionOffset` return vector values, so assign a mutated value back to persist it: `local p = layout.positionOffset; p.x = p.x + 1; layout.positionOffset = p`.

Properties

Type	Name	Default	Langs
float	x	`0.0`	C++ | Lua
float	y	`0.0`	C++ | Lua

Static constants

Name	Value
`ZERO`	`(0, 0)`
`UNIT_X`	`(1, 0)`
`UNIT_Y`	`(0, 1)`
`NEGATIVE_UNIT_X`	`(-1, 0)`
`NEGATIVE_UNIT_Y`	`(0, -1)`
`UNIT_SCALE`	`(1, 1)`

Methods

Type	Name	Langs
float	length	C++ | Lua
float	squaredLength	C++ | Lua
float	distance	C++ | Lua
float	squaredDistance	C++ | Lua
float	dotProduct	C++ | Lua
float	crossProduct	C++ | Lua
Vector2&	normalize	C++ | Lua
Vector2	normalized	C++ | Lua
float	normalizeL	C++ | Lua
Vector2	midPoint	C++ | Lua
Vector2	moveTowards	C++ | Lua
Vector2	lerp	C++ | Lua
Vector2	perpendicular	C++ | Lua

Type	Name	Langs
Vector2	reflect	C++ | Lua
void	makeFloor	C++ | Lua
void	makeCeil	C++ | Lua
bool	isValid	C++ | Lua
std::string	toString	C++ | Lua (`tostring`)

Method details

length / squaredLength

- float **length**() const
- float **squaredLength**() const

`length()` returns the Euclidean magnitude `√(x² + y²)`. `squaredLength()` skips the square root for cheaper comparisons.

distance / squaredDistance

- float **distance**(const Vector2& rhs) const
- float **squaredDistance**(const Vector2& rhs) const

Distance between this vector and `rhs`. `squaredDistance()` is cheaper and useful when you only need to compare distances.

dotProduct

- float **dotProduct**(const Vector2& vec) const

Returns the dot product (`x*vec.x + y*vec.y`). Useful for projections and angle calculations.

crossProduct

- float **crossProduct**(const Vector2& rkVector) const

Returns the 2D cross product (scalar), which equals the signed area of the parallelogram formed by the two vectors. Positive means `rkVector` is to the left of this vector.

normalize / normalized

- Vector2& **normalize**()
- Vector2 **normalized**() const

`normalize()` scales the vector in-place to unit length and returns a reference to `*this`. `normalized()` returns a new normalised copy without modifying the original.

C++

```
Vector2 dir(3.0f, 4.0f);
Vector2 unit = dir.normalized();  // (0.6, 0.8)
```

Lua

```
local dir = Vector2(3.0, 4.0)
local unit = dir:normalized()
```

normalizeL

• float **normalizeL**()

Normalises in-place and returns the original length. Combines normalisation with a length query in a single pass.

midPoint

• Vector2 **midPoint**(const Vector2& vec) const

Returns the midpoint between this vector and `vec`.

moveTowards

• Vector2 **moveTowards**(const Vector2& target, float maxDistanceDelta) const

Returns `*this` moved toward `target` by at most `maxDistanceDelta`, never past it. See Vector3 moveTowards (p. 662). On a velocity it gives a steady acceleration toward a wanted speed:

C++

```
Body2D body = Object(getScene(), getEntity()).getBody2D();
Vector2 wanted(runSpeed, body.getLinearVelocity().y);
body.setLinearVelocity(body.getLinearVelocity().moveTowards(wanted, acceleration * Engine::getDeltatim
e()));
```

Lua

```
local body = Object(self.scene, self.entity):getBody2D()
local wanted = Vector2(self.runSpeed, body.linearVelocity.y)
body.linearVelocity = body.linearVelocity:moveTowards(wanted, self.acceleration * Engine.deltatime)
```

lerp

• Vector2 **lerp**(const Vector2& target, float t) const

Linear interpolation from `*this` toward `target` by factor `t`. `t = 0` returns `*this`, `t = 1` returns `target`, and values outside `[0, 1]` extrapolate. See Vector3 lerp (p. 662) for following a target at any frame rate.

perpendicular

• Vector2 **perpendicular**() const

Returns a vector perpendicular to this one (rotated 90° counter-clockwise).

reflect

• Vector2 **reflect**(const Vector2& normal) const

Reflects this vector about `normal`. `normal` should be a unit vector.

makeFloor / makeCeil

• void **makeFloor**(const Vector2& cmp)

• void **makeCeil**(const Vector2& cmp)

Component-wise minimum (`makeFloor`) or maximum (`makeCeil`) with `cmp`, applied in-place.

isValid

- bool **isValid**() const

Returns `false` if any component is NaN or infinite.

toString

- std::string **toString**() const

Returns a human-readable string representation like `"Vector2(1.000000, 2.000000)"`.

Vector3

C++ type: `Vector3`

Description

A 3D vector with `float` components `x`, `y`, and `z`. The primary type for 3D positions, directions, normals, colours (RGB), and velocity in Doriax.

`+` and `-` work on two vectors. `*` and `/` take a number or another vector, which multiplies or divides per component, and the number can also come first in a multiplication (`2 * v`). Comparison operators (`==`, `<`) are also available.

> **Lua components are writable**
> Lua can read and assign `x`, `y`, and `z` directly. Engine properties such as `obj.position` return vector values, so assign a mutated value back to persist it: `local p = obj.position; p.x = p.x + 1; obj.position = p`. Whole-value arithmetic such as `obj.position = obj.position + Vector3(1, 0, 0)` also works.

Properties

Type	Name	Default	Langs
float	x	`0.0`	C++ | Lua
float	y	`0.0`	C++ | Lua
float	z	`0.0`	C++ | Lua

Static constants

Name	Value
`ZERO`	`(0, 0, 0)`
`UNIT_X`	`(1, 0, 0)`
`UNIT_Y`	`(0, 1, 0)`
`UNIT_Z`	`(0, 0, 1)`
`UNIT_SCALE`	`(1, 1, 1)`

Methods

Type	Name	Langs
float	length	C++ | Lua
float	squaredLength	C++ | Lua
float	distance	C++ | Lua
float	squaredDistance	C++ | Lua
float	dotProduct	C++ | Lua
float	absDotProduct	C++ | Lua
Vector3	crossProduct	C++ | Lua
Vector3&	normalize	C++ | Lua
Vector3	normalized	C++ | Lua
float	normalizeL	C++ | Lua
Vector3	midPoint	C++ | Lua
Vector3	moveTowards	C++ | Lua
Vector3	lerp	C++ | Lua

Type	Name	Langs
Vector3	perpendicular	C++ | Lua
Vector3	reflect	C++ | Lua
void	makeFloor	C++ | Lua
void	makeCeil	C++ | Lua
bool	isValid	C++ | Lua
std::string	toString	C++ | Lua (`tostring`)

Method details

length / squaredLength

- float **length**() const
- float **squaredLength**() const

`length()` returns the Euclidean magnitude `√(x² + y² + z²)`. `squaredLength()` omits the square root for cheaper magnitude comparisons.

distance / squaredDistance

- float **distance**(const Vector3& rhs) const
- float **squaredDistance**(const Vector3& rhs) const

Distance between this point and `rhs`. Use `squaredDistance()` when only relative comparisons are needed.

dotProduct

- float **dotProduct**(const Vector3& v) const

Scalar dot product `(x*v.x + y*v.y + z*v.z)`. Returns the cosine of the angle between two unit vectors.

absDotProduct

- float **absDotProduct**(const Vector3& v) const

Returns the absolute value of the dot product. Useful for axis-aligned overlap tests.

crossProduct

- Vector3 **crossProduct**(const Vector3& v) const

Returns the cross product, a vector perpendicular to both `*this` and `v`. The result follows the right-hand rule.

C++

```
Vector3 right = forward.crossProduct(Vector3::UNIT_Y);
```

Lua

```
local right = forward:crossProduct(Vector3.UNIT_Y)
```

normalize / normalized

- Vector3& **normalize**()
- Vector3 **normalized**() const

`normalize()` scales the vector in-place to unit length. `normalized()` returns a new unit-length copy.

normalizeL

• float **normalizeL**()

Normalises in-place and returns the original magnitude in a single operation.

midPoint

• Vector3 **midPoint**(const Vector3& v) const

Returns the midpoint between this vector and `v`.

moveTowards

• Vector3 **moveTowards**(const Vector3& target, float maxDistanceDelta) const

Returns a point moved from `*this` toward `target` by at most `maxDistanceDelta`. It **never overshoots**: once the remaining distance is within `maxDistanceDelta`, `target` is returned exactly. Pass `speed * deltatime` as `maxDistanceDelta` for frame-rate independent movement. This is the preferred way to move toward a target, since the naïve `direction * speed * deltatime` overshoots when the step is larger than the remaining distance (for example on the first, large deltatime (p. 366) frame). A negative `maxDistanceDelta` moves away from `target`. Equivalent to Unity's `Vector3.MoveTowards` and Godot's `Vector3.move_toward`.

C++

```cpp
void MyScript::onUpdate() {
    Object obj(getScene(), getEntity());
    float step = speed * Engine::getDeltatime();
    obj.setPosition(obj.getPosition().moveTowards(targetPosition, step));
}
```

Lua

```lua
function MyScript:onUpdate()
    local obj = Object(self.scene, self.entity)
    local step = self.speed * Engine.deltatime
    obj.position = obj.position:moveTowards(self.targetPosition, step)
end
```

lerp

• Vector3 **lerp**(const Vector3& target, float t) const

Linear interpolation from `*this` toward `target` by factor `t`. `t = 0` returns `*this`, `t = 1` returns `target`, and values outside `[0, 1]` extrapolate. Unlike moveTowards, the step is proportional to the remaining distance, so the motion slows down as it gets close.

To follow a target smoothly every frame, derive `t` from the frame time as below. A fixed `t`, or `speed * deltatime`, makes the result depend on the frame rate.

C++

```cpp
// A higher sharpness follows the target more tightly
float t = 1.0f - std::exp(-sharpness * Engine::getDeltatime());
obj.setPosition(obj.getPosition().lerp(targetPosition, t));
```

Lua

```lua
-- A higher sharpness follows the target more tightly
local t = 1 - math.exp(-self.sharpness * Engine.deltatime)
obj.position = obj.position:lerp(self.targetPosition, t)
```

perpendicular

- Vector3 **perpendicular**() const

Returns an arbitrary vector perpendicular to this one. Useful when you need a tangent without a reference direction.

reflect

- Vector3 **reflect**(const Vector3& normal) const

Reflects this vector about `normal` (a unit vector). Equivalent to `v - 2 * dot(v, n) * n`.

makeFloor / makeCeil

- void **makeFloor**(const Vector3& v)
- void **makeCeil**(const Vector3& v)

Component-wise minimum (`makeFloor`) or maximum (`makeCeil`), applied in-place. Useful for AABB calculations.

isValid

- bool **isValid**() const

Returns `false` if any component is NaN or infinite.

toString

- std::string **toString**() const

Returns a human-readable string like `"Vector3(1.000000, 2.000000, 3.000000)"`.

Vector4

C++ type: `Vector4`

Description

A 4D vector with `float` components `x`, `y`, `z`, and `w`. Used for RGBA colours, homogeneous coordinates, and quaternion-adjacent math inside the engine.

`+` and `-` work on two vectors. `*` and `/` take a number or another vector, which multiplies or divides per component, and the number can also come first in a multiplication (`2 * v`).

> **Lua components are writable**
> Lua can read and assign `x`, `y`, `z`, and `w` directly. When a vector comes from an engine property, mutate a local value and assign it back to persist the change.

Properties

Type	Name	Default	Langs
float	x	`0.0`	C++ | Lua
float	y	`0.0`	C++ | Lua
float	z	`0.0`	C++ | Lua
float	w	`0.0`	C++ | Lua

Static constants

Name	Value
`ZERO`	`(0, 0, 0, 0)`
`UNIT_X`	`(1, 0, 0, 0)`
`UNIT_Y`	`(0, 1, 0, 0)`
`UNIT_Z`	`(0, 0, 1, 0)`
`UNIT_W`	`(0, 0, 0, 1)`
`UNIT_SCALE`	`(1, 1, 1, 1)`

Methods

Type	Name	Langs
float	dotProduct	C++ | Lua
void	divideByW	C++ | Lua
Vector4	moveTowards	C++ | Lua
Vector4	lerp	C++ | Lua
bool	isNaN	C++ | Lua
bool	isValid	C++ | Lua
std::string	toString	C++ | Lua (`tostring`)

Method details

dotProduct

- float **dotProduct**(const Vector4& vec) const

Returns the 4D dot product `(x*vec.x + y*vec.y + z*vec.z + w*vec.w)`.

divideByW

• void **divideByW**()

Divides `x`, `y`, and `z` by `w` in place, the perspective divide that turns homogeneous clip-space coordinates into NDC. `w` keeps its value.

moveTowards

• Vector4 **moveTowards**(const Vector4& target, float maxDistanceDelta) const

Returns `*this` moved toward `target` by at most `maxDistanceDelta`, never past it. See Vector3 moveTowards (p. 662). On a colour it gives a fade at a steady rate:

C++

```
// Fully transparent after half a second
Sprite sprite(getScene(), getEntity());
Vector4 c = sprite.getColor();
sprite.setColor(c.moveTowards(Vector4(c.x, c.y, c.z, 0.0f), 2.0f * Engine::getDeltatime()));
```

Lua

```
-- Fully transparent after half a second
local sprite = Sprite(self.scene, self.entity)
local c = sprite.color
sprite.color = c:moveTowards(Vector4(c.x, c.y, c.z, 0), 2 * Engine.deltatime)
```

lerp

• Vector4 **lerp**(const Vector4& target, float t) const

Linear interpolation from `*this` toward `target` by factor `t`. `t = 0` returns `*this`, `t = 1` returns `target`, and values outside `[0, 1]` extrapolate. Blends two colours, for example `red:lerp(blue, 0.5)`.

isNaN

• bool **isNaN**() const

Returns `true` if any component is NaN.

isValid

• bool **isValid**() const

Returns `false` if any component is NaN or infinite.

toString

• std::string **toString**() const

Returns a human-readable string like `"Vector4(1.0, 0.0, 0.0, 1.0)"`.

Components

ActionComponent

`ActionComponent` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`state`	C++ | Lua (read-only)
`timecount`	C++ | Lua (read-only)
`speed`	C++ | Lua
`startOffset`	C++ | Lua
`randomStart`	C++ | Lua
`startTrigger`	C++ | Lua
`stopTrigger`	C++ | Lua
`pauseTrigger`	C++ | Lua
`ownedTarget`	C++ | Lua
`target`	C++ | Lua
`onStart`	C++ | Lua
`onPause`	C++ | Lua
`onStop`	C++ | Lua
`onStep`	C++ | Lua

AlphaActionComponent

The start and end alpha of an AlphaAction (p. 572). In Lua, get it with `action:getAlphaActionComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`endAlpha`	C++ | Lua
`startAlpha`	C++ | Lua

AnimationComponent

The data behind an Animation (p. 573). In Lua, get it with `animation:getAnimationComponent()`.

Properties

Name	Languages
`actions`	C++ | Lua
`ownedActions`	C++ | Lua
`loop`	C++ | Lua
`duration`	C++ | Lua
`defaultFadeTime`	C++ | Lua

`AnimationComponent` does not store a separate name. Animations are entities, so their display and lookup name comes from the inherited `Animation::name` (p. 574) entity property.

`actions` is the list of ActionFrame (p. 708) entries making up the timeline. A frame `duration` of `0` means **auto** — the frame follows the action's own duration.

`defaultFadeTime` is the crossfade duration (seconds) used by Model::playAnimation() (p. 452) when no explicit fade time is passed. Edit it in the **Properties** window as **Fade time**, or on an Animation (p. 573) object via `defaultFadeTime`.

Body2DComponent

The data a Body2D (p. 541) is built from. In Lua, get it with `body:getBody2DComponent()`.

Properties

> **Writing a component field**
>
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`numShapes`	C++ | Lua (read-only)
`type`	C++ | Lua
`needReloadBody`	C++ | Lua
`needUpdateShapes`	C++ | Lua

Set `needReloadBody` to rebuild the body from these fields. To change shapes, or a live body without a rebuild, use the Body2D (p. 541) methods.

Body3DComponent

The data a Body3D (p. 551) is built from. In Lua, get it with `body:getBody3DComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`numShapes`	C++ | Lua (read-only)
`overrideMassProperties`	C++ | Lua
`solidBoxSize`	C++ | Lua
`solidBoxDensity`	C++ | Lua
`type`	C++ | Lua
`motionQuality`	C++ | Lua
`gravityFactor`	C++ | Lua
`buoyancy`	C++ | Lua
`waterDrag`	C++ | Lua
`waterAngularDrag`	C++ | Lua
`allowedDOFs`	C++ | Lua
`sensor`	C++ | Lua
`needReloadBody`	C++ | Lua
`needUpdateShapes`	C++ | Lua

Set `needReloadBody` to rebuild the body from these fields. To change shapes, or a live body without a rebuild, use the Body3D (p. 551) methods. `allowedDOFs` is a bit mask: X, Y, Z translation, then X, Y, Z rotation.

BoneComponent

The bind pose of a Bone (p. 413), filled in when its model loads. In Lua, get it with `bone:getBoneComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`model`	C++ | Lua (read-only)
`index`	C++ | Lua (read-only)
`bindPosition`	C++ | Lua (read-only)
`bindRotation`	C++ | Lua (read-only)
`bindScale`	C++ | Lua (read-only)
`offsetMatrix`	C++ | Lua (read-only)

ButtonComponent

`ButtonComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`label`	C++ | Lua (read-only)
`textureNormal`	C++ | Lua
`textureHovered`	C++ | Lua
`texturePressed`	C++ | Lua
`textureDisabled`	C++ | Lua
`colorNormal`	C++ | Lua
`colorHovered`	C++ | Lua
`colorPressed`	C++ | Lua
`colorDisabled`	C++ | Lua
`scaleHovered`	C++ | Lua
`scalePressed`	C++ | Lua
`transitionTime`	C++ | Lua
`onPress`	C++ | Lua
`onRelease`	C++ | Lua
`pressed`	C++ | Lua (read-only)
`hovered`	C++ | Lua (read-only)
`disabled`	C++ | Lua
`focusable`	C++ | Lua
`needUpdateButton`	C++ | Lua

CameraComponent

The data behind a Camera (p. 414). In Lua, get it with `camera:getCameraComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`type`	C++ | Lua
`projectionMatrix`	C++ | Lua (read-only)
`viewMatrix`	C++ | Lua (read-only)
`viewProjectionMatrix`	C++ | Lua (read-only)
`target`	C++ | Lua
`worldTarget`	C++ | Lua (read-only)
`up`	C++ | Lua
`direction`	C++ | Lua (read-only)
`right`	C++ | Lua (read-only)
`worldUp`	C++ | Lua (read-only)
`worldDirection`	C++ | Lua (read-only)
`worldRight`	C++ | Lua (read-only)
`leftClip`	C++ | Lua
`rightClip`	C++ | Lua
`bottomClip`	C++ | Lua
`topClip`	C++ | Lua
`yfov`	C++ | Lua
`aspect`	C++ | Lua
`nearClip`	C++ | Lua
`farClip`	C++ | Lua
`renderToTexture`	C++ | Lua
`framebuffer`	C++ | Lua (read-only)
`framebufferWidth`	C++ | Lua
`framebufferHeight`	C++ | Lua
`framebufferFilter`	C++ | Lua
`transparentSort`	C++ | Lua
`depthTest`	C++ | Lua
`useTarget`	C++ | Lua
`autoResize`	C++ | Lua
`hasCustomViewMatrix`	C++ | Lua
`customViewMatrix`	C++ | Lua
`invertCulling`	C++ | Lua
`needUpdate`	C++ | Lua
`shaderParameters`	C++

Set `needUpdate` after changing the projection, clip, or target fields. `framebuffer` is the render target used when `renderToTexture` is on. `shaderParameters` holds the camera's saved view shader parameters (p. 421) by `Collection.name`.

ColorActionComponent

The start and end colors of a ColorAction (p. 577). In Lua, get it with `action:getColorActionComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`endColor`	C++ | Lua
`startColor`	C++ | Lua
`useSRGB`	C++ | Lua

`useSRGB` tells whether the two colors are sRGB.

FogComponent

The data behind Fog (p. 422). In Lua, get it with `fog:getFogComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`type`	C++ | Lua
`color`	C++ | Lua
`density`	C++ | Lua
`linearStart`	C++ | Lua
`linearEnd`	C++ | Lua

`color` is linear here, while the Fog (p. 422) setter takes sRGB.

ImageComponent

The nine-patch settings of an Image (p. 513). In Lua, get it with `image:getImageComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`patchMarginLeft`	C++ | Lua
`patchMarginRight`	C++ | Lua
`patchMarginTop`	C++ | Lua
`patchMarginBottom`	C++ | Lua
`textureScaleFactor`	C++ | Lua
`needUpdatePatches`	C++ | Lua

Set `needUpdatePatches` after changing the margins or `textureScaleFactor`.

InstancedMeshComponent

The instances of a Mesh (p. 436) drawn in a single call. In Lua, `mesh:getInstancedMeshComponent()` returns it, or `nil` until the mesh is instanced with `createInstancedMesh()` or `addInstance()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`instances`	C++ | Lua
`maxInstances`	C++ | Lua (read-only)
`numVisible`	C++ | Lua (read-only)
`cullInstances`	C++ | Lua
`cullDistance`	C++ | Lua
`distanceFade`	C++ | Lua
`fadeStart`	C++ | Lua
`fadeEnd`	C++ | Lua
`instancedBillboard`	C++ | Lua
`instancedCylindricalBillboard`	C++ | Lua
`needUpdateBuffer`	C++ | Lua
`needUpdateInstances`	C++ | Lua

`instances` is a list of InstanceData (p. 713). Set `needUpdateInstances` after assigning it. Only the first `maxInstances` are drawn; raise it with `Mesh.maxInstances` (p. 436). In Lua, lists are copies: assign the changed table back.

Joint2DComponent

The data a Joint2D (p. 563) is built from. In Lua, get it with `joint:getJoint2DComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`type`	C++ | Lua
`bodyA`	C++ | Lua
`bodyB`	C++ | Lua
`anchorA`	C++ | Lua
`anchorB`	C++ | Lua
`axis`	C++ | Lua
`target`	C++ | Lua
`autoAnchors`	C++ | Lua
`rope`	C++ | Lua
`needUpdateJoint`	C++ | Lua

The Joint2D (p. 563) setters fill these fields and build the joint. A joint set during a physics step is built by the next update. Removing or rebuilding a body takes its joints down until the body is back. Set `needUpdateJoint` to rebuild it from these fields.

Joint3DComponent

The data a Joint3D (p. 564) is built from. In Lua, get it with `joint:getJoint3DComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`type`	C++ | Lua
`bodyA`	C++ | Lua
`bodyB`	C++ | Lua
`anchorA`	C++ | Lua
`anchorB`	C++ | Lua
`anchor`	C++ | Lua
`axis`	C++ | Lua
`normal`	C++ | Lua
`twistAxis`	C++ | Lua
`planeAxis`	C++ | Lua
`axisX`	C++ | Lua
`axisY`	C++ | Lua
`limitsMin`	C++ | Lua
`limitsMax`	C++ | Lua
`normalHalfConeAngle`	C++ | Lua
`planeHalfConeAngle`	C++ | Lua
`twistMinAngle`	C++ | Lua
`twistMaxAngle`	C++ | Lua
`fixedPointA`	C++ | Lua
`fixedPointB`	C++ | Lua
`hingeA`	C++ | Lua
`hingeB`	C++ | Lua
`hinge`	C++ | Lua
`slider`	C++ | Lua
`numTeethGearA`	C++ | Lua
`numTeethGearB`	C++ | Lua
`numTeethRack`	C++ | Lua
`numTeethGear`	C++ | Lua
`rackLength`	C++ | Lua
`motorState`	C++ | Lua
`motorTargetVelocity`	C++ | Lua
`motorTargetPosition`	C++ | Lua
`motorFrequency`	C++ | Lua
`motorDamping`	C++ | Lua
`motorMaxForce`	C++ | Lua

Name	Languages
`pathPoints`	C++ | Lua
`pathPosition`	C++ | Lua
`isLooping`	C++ | Lua
`autoAnchors`	C++ | Lua
`needUpdateJoint`	C++ | Lua
`needUpdateMotor`	C++ | Lua

The Joint3D (p. 564) setters fill these fields and build the joint. A joint whose bodies aren't ready, or a gear or rack whose hinge or slider isn't, is built by a later update. Removing or rebuilding a body takes its joints down, along with any gear or rack using them, until the body is back. While the scene runs, a joint rebuilt this way keeps the frames it had, so it comes back in the same rest pose; a path, gear, or rack joint, or one whose body's center of mass moved, is rebuilt from these fields instead. Set `needUpdateJoint` to rebuild it from these fields. `pathPoints` comes to Lua as a table copy. The `motor…` fields are described in Joint3D — Motors (p. 564); after writing them, set `needUpdateMotor` to apply them to the built joint. `needUpdateJoint` would rebuild it instead, making its current pose the hinge's angle `0`.

KeyframeTracksComponent

The key times and easings shared by a KeyframeTracks (p. 578) action and the track types built on it. In Lua, get it with `tracks:getKeyframeTracksComponent()`.

Properties

> **Writing a component field**
>
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`times`	C++ | Lua
`easings`	C++ | Lua
`loop`	C++ | Lua
`relative`	C++ | Lua
`index`	C++ | Lua (read-only)
`interpolation`	C++ | Lua (read-only)

`index` and `interpolation` are playback state. In Lua, lists are copies: assign the changed table back.

Light2DComponent

The data behind a Light2D (p. 430). In Lua, get it with `light:getLight2DComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`color`	C++ | Lua
`intensity`	C++ | Lua
`range`	C++ | Lua
`falloff`	C++ | Lua
`height`	C++ | Lua
`shadows`	C++ | Lua
`shadowBias`	C++ | Lua
`shadowSoftness`	C++ | Lua
`mapResolution`	C++ | Lua

`color` is linear here, while the Light2D (p. 430) setter takes sRGB.

LightComponent

The data behind a Light (p. 424). In Lua, get it with `light:getLightComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`type`	C++ | Lua
`direction`	C++ | Lua
`worldDirection`	C++ | Lua (read-only)
`color`	C++ | Lua
`range`	C++ | Lua
`intensity`	C++ | Lua
`innerConeCos`	C++ | Lua
`outerConeCos`	C++ | Lua
`spotMask`	C++ | Lua (read-only)
`shadows`	C++ | Lua
`automaticShadowCamera`	C++ | Lua
`shadowBias`	C++ | Lua
`shadowNormalBias`	C++ | Lua
`mapResolution`	C++ | Lua
`shadowCameraNearFar`	C++ | Lua
`numShadowCascades`	C++ | Lua
`shadowCascadeBlend`	C++ | Lua
`needUpdateShadowCamera`	C++ | Lua
`needUpdateShadowMap`	C++ | Lua

`color` is linear here, while the Light (p. 424) setter takes sRGB. `innerConeCos` and `outerConeCos` hold the cosine of half the spot cone angles. Change `type`, `shadows`, and `mapResolution` through the Light (p. 424) setters, which also rebuild the meshes' shaders.

LinesComponent

The data behind Lines (p. 433). In Lua, get it with `lines:getLinesComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`loaded`	C++ | Lua (read-only)
`lines`	C++ | Lua
`maxLines`	C++ | Lua (read-only)
`customShader`	C++ | Lua
`needUpdateBuffer`	C++ | Lua
`needReload`	C++ | Lua

`lines` is a list of LineData (p. 713). Set `needUpdateBuffer` after assigning it. Only the first `maxLines` are drawn; raise it with `Lines.maxLines` (p. 433). Set `needReload` after changing `customShader`. In Lua, lists are copies: assign the changed table back.

MeshComponent

The data behind a Mesh (p. 436) and the objects built on it, such as Sprite (p. 480), Model (p. 450), and Terrain (p. 486). In Lua, get it with `mesh:getMeshComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`loaded`	C++ | Lua (read-only)
`vertexCount`	C++ | Lua (read-only)
`numSubmeshes`	C++ | Lua (read-only)
`customShader`	C++ | Lua
`customDepthShader`	C++ | Lua
`vertexOffsetShader`	C++ | Lua
`vertexOffsetMax`	C++ | Lua
`coverageShader`	C++ | Lua
`coverageShadows`	C++ | Lua
`aabb`	C++ | Lua (read-only)
`worldAABB`	C++ | Lua (read-only)
`receiveLights`	C++ | Lua
`receiveIBL`	C++ | Lua
`castShadows`	C++ | Lua
`receiveShadows`	C++ | Lua
`shadowsBillboard`	C++ | Lua
`renderInReflectionProbes`	C++ | Lua
`transparent`	C++ | Lua
`autoTransparency`	C++ | Lua
`lodEnabled`	C++ | Lua
`lodBias`	C++ | Lua
`cullingMode`	C++ | Lua
`windingOrder`	C++ | Lua
`needUpdateAABB`	C++ | Lua
`needUpdateBuffer`	C++ | Lua
`needReload`	C++ | Lua

Submeshes, materials, vertex buffers, and morph weights are C++ only; from Lua, use the Mesh (p. 436) methods for them.

MeshPolygonComponent

The data behind a MeshPolygon (p. 713). In Lua, get it with `polygon:getMeshPolygonComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`width`	C++ | Lua (read-only)
`height`	C++ | Lua (read-only)
`automaticFlipY`	C++ | Lua
`flipY`	C++ | Lua
`points`	C++ | Lua
`needUpdatePolygon`	C++ | Lua

`points` is a list of PolygonPoint (p. 714). Set `needUpdatePolygon` after assigning it. `width` and `height` are measured from the points. In Lua, lists are copies: assign the changed table back.

MirrorComponent

The data behind a Mirror (p. 448). In Lua, get it with `mirror:getMirrorComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`normal`	C++ | Lua

`normal` is the local normal of the reflecting surface. The default, +Z, matches a wall mesh.

ModelComponent

The data behind a Model (p. 450). In Lua, get it with `model:getModelComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`skeleton`	C++ | Lua (read-only)
`bonesNameMapping`	C++ | Lua (read-only)
`morphNameMapping`	C++ | Lua (read-only)
`animations`	C++ | Lua (read-only)
`filename`	C++ | Lua
`mergeStaticMeshes`	C++ | Lua
`needUpdateModel`	C++ | Lua

`bonesNameMapping` maps bone names to entities and `morphNameMapping` maps morph target names to indices. Both come to Lua as tables. `animations` lists the model's Animation (p. 573) entities.

MorphTracksComponent

The key values of a MorphTracks (p. 579) action, one list of weights per key. In Lua, get it with `tracks:getMorphTracksComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`values`	C++ | Lua
`inTangents`	C++ | Lua
`outTangents`	C++ | Lua

In Lua, lists are copies: assign the changed table back.

Occluder2DComponent

The data behind an Occluder2D (p. 462). In Lua, get it with `occluder:getOccluder2DComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`shape`	C++ | Lua
`points`	C++ | Lua
`closed`	C++ | Lua
`enabled`	C++ | Lua

`points` is used when `shape` is `Occluder2DShape.POLYGON`. In Lua, lists are copies: assign the changed table back.

PanelComponent

`PanelComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`headerimage`	C++ | Lua (read-only)
`headercontainer`	C++ | Lua (read-only)
`headertext`	C++ | Lua (read-only)
`titleAnchorPreset`	C++ | Lua
`minWidth`	C++ | Lua
`minHeight`	C++ | Lua
`resizeMargin`	C++ | Lua
`canMove`	C++ | Lua
`canResize`	C++ | Lua
`canBringToFront`	C++ | Lua
`headerPointerDown`	C++ | Lua (read-only)
`onMove`	C++ | Lua
`onResize`	C++ | Lua
`needUpdatePanel`	C++ | Lua

ParticlesComponent

The data behind Particles (p. 724). In Lua, get it with `particles:getParticlesComponent()`. The old misspelled `getParticlesomponent()` still works.

Properties

Name	Languages
`newParticlesCount`	C++ | Lua (read-only)
`lastUsedParticle`	C++ | Lua (read-only)
`emitter`	C++ | Lua
`loop`	C++ | Lua
`localSpace`	C++ | Lua
`rate`	C++ | Lua
`maxPerUpdate`	C++ | Lua
`bursts`	C++ | Lua
`currentBurst`	C++ | Lua (read-only)
`lifeInitializer`	C++ | Lua
`positionInitializer`	C++ | Lua
`positionModifier`	C++ | Lua
`velocityInitializer`	C++ | Lua
`velocityModifier`	C++ | Lua
`accelerationInitializer`	C++ | Lua
`accelerationModifier`	C++ | Lua
`colorInitializer`	C++ | Lua
`colorModifier`	C++ | Lua
`colorGradient`	C++ | Lua
`alphaInitializer`	C++ | Lua
`alphaModifier`	C++ | Lua
`sizeInitializer`	C++ | Lua
`sizeModifier`	C++ | Lua
`spriteInitializer`	C++ | Lua
`spriteModifier`	C++ | Lua
`rotationInitializer`	C++ | Lua
`rotationModifier`	C++ | Lua
`scaleInitializer`	C++ | Lua
`scaleModifier`	C++ | Lua

PointsComponent

The data behind Points (p. 464). In Lua, get it with `points:getPointsComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`loaded`	C++ | Lua (read-only)
`points`	C++ | Lua
`numFramesRect`	C++ | Lua (read-only)
`maxPoints`	C++ | Lua (read-only)
`numVisible`	C++ | Lua (read-only)
`aabb`	C++ | Lua (read-only)
`customShader`	C++ | Lua
`texture`	C++ | Lua
`transparent`	C++ | Lua
`autoTransparency`	C++ | Lua
`needUpdate`	C++ | Lua
`needUpdateBuffer`	C++ | Lua
`needUpdateTexture`	C++ | Lua
`needReload`	C++ | Lua

`points` is a list of PointData (p. 714). Set `needUpdate` after assigning it. Only the first `maxPoints` are drawn; raise it with `Points.maxPoints` (p. 464). Set `needReload` after changing `customShader`. In Lua, lists are copies: assign the changed table back.

PolygonComponent

The data behind a UI Polygon (p. 521). In Lua, get it with `polygon:getPolygonComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`points`	C++ | Lua
`needUpdatePolygon`	C++ | Lua

`points` is a list of PolygonPoint (p. 714). Set `needUpdatePolygon` after assigning it. In Lua, lists are copies: assign the changed table back.

PositionActionComponent

The start and end positions of a PositionAction (p. 580). In Lua, get it with `action:getPositionActionComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`endPosition`	C++ | Lua
`startPosition`	C++ | Lua

ProgressbarComponent

`ProgressbarComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`fill`	C++ | Lua (read-only)
`type`	C++ | Lua
`value`	C++ | Lua
`fillMarginLeft`	C++ | Lua
`fillMarginRight`	C++ | Lua
`fillMarginTop`	C++ | Lua
`fillMarginBottom`	C++ | Lua
`needUpdateProgressbar`	C++ | Lua

ReflectionProbeComponent

The data behind a ReflectionProbe (p. 468). In Lua, get it with `probe:getReflectionProbeComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`mode`	C++ | Lua
`updateMode`	C++ | Lua
`boxOffset`	C++ | Lua
`boxSize`	C++ | Lua
`blendDistance`	C++ | Lua
`intensity`	C++ | Lua
`priority`	C++ | Lua
`resolution`	C++ | Lua
`nearClip`	C++ | Lua
`farClip`	C++ | Lua
`updateInterval`	C++ | Lua
`includeSky`	C++ | Lua
`texture`	C++ | Lua (read-only)
`needUpdate`	C++ | Lua

Set `needUpdate` to capture again. `texture` is the captured cubemap.

RotateTracksComponent

The key values of a RotateTracks (p. 581) action. In Lua, get it with `tracks:getRotateTracksComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`values`	C++ | Lua
`inTangents`	C++ | Lua
`outTangents`	C++ | Lua

In Lua, lists are copies: assign the changed table back.

RotationActionComponent

The rotations of a RotationAction (p. 582). In Lua, get it with `action:getRotationActionComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`endRotation`	C++ | Lua
`startRotation`	C++ | Lua
`shortestPath`	C++ | Lua
`spin`	C++ | Lua
`axis`	C++ | Lua
`angle`	C++ | Lua

ScaleActionComponent

The start and end scales of a ScaleAction (p. 584). In Lua, get it with `action:getScaleActionComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`endScale`	C++ | Lua
`startScale`	C++ | Lua

ScaleTracksComponent

The key values of a ScaleTracks (p. 585) action. In Lua, get it with `tracks:getScaleTracksComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`values`	C++ | Lua
`inTangents`	C++ | Lua
`outTangents`	C++ | Lua

In Lua, lists are copies: assign the changed table back.

ScrollbarComponent

`ScrollbarComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`bar`	C++ | Lua (read-only)
`type`	C++ | Lua
`onChange`	C++ | Lua
`barSize`	C++ | Lua
`step`	C++ | Lua
`barMarginLeft`	C++ | Lua
`barMarginRight`	C++ | Lua
`barMarginTop`	C++ | Lua
`barMarginBottom`	C++ | Lua
`barPointerDown`	C++ | Lua (read-only)
`barPointerPos`	C++ | Lua (read-only)
`needUpdateScrollbar`	C++ | Lua

SkyComponent

The data behind a SkyBox (p. 472). In Lua, get it with `sky:getSkyComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`loaded`	C++ | Lua (read-only)
`customShader`	C++ | Lua
`texture`	C++ | Lua
`color`	C++ | Lua
`rotation`	C++ | Lua
`visible`	C++ | Lua
`needUpdateEnvironment`	C++ | Lua
`needUpdateTexture`	C++ | Lua
`needUpdateSky`	C++ | Lua
`needReload`	C++ | Lua

Set `needUpdateTexture` after replacing `texture`, `needUpdateSky` after changing `rotation`, and `needReload` after changing `customShader`.

SoundComponent

`SoundComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`handle`	C++ | Lua (read-only)
`state`	C++ | Lua (read-only)
`filename`	C++ | Lua
`loaded`	C++ | Lua (read-only)
`loadFailed`	C++ | Lua (read-only)
`enableClocked`	C++ | Lua
`lastPosition`	C++ | Lua (read-only)
`startTrigger`	C++ | Lua
`stopTrigger`	C++ | Lua
`pauseTrigger`	C++ | Lua
`onStart`	C++ | Lua
`onPause`	C++ | Lua
`onStop`	C++ | Lua
`volume`	C++ | Lua
`pan`	C++ | Lua
`speed`	C++ | Lua
`looping`	C++ | Lua
`loopingPoint`	C++ | Lua
`protectVoice`	C++ | Lua
`inaudibleBehaviorMustTick`	C++ | Lua
`inaudibleBehaviorKill`	C++ | Lua
`minDistance`	C++ | Lua
`maxDistance`	C++ | Lua
`attenuationModel`	C++ | Lua
`attenuationRolloffFactor`	C++ | Lua
`dopplerFactor`	C++ | Lua
`length`	C++ | Lua (read-only)
`playingTime`	C++ | Lua (read-only)
`needUpdate`	C++ | Lua

`needUpdate` applies `volume`, `speed`, `pan`, `looping`, `loopingPoint`, `protectVoice`, the two `inaudibleBehavior` fields and the 3D attenuation to the playing voice. It is not a load: a new `filename` needs Sound.loadSound (p. 478).

SpriteAnimationComponent

`SpriteAnimationComponent` holds the frame sequence and playback state for a SpriteAnimation (p. 586) action. Configure it through `SpriteAnimation::setAnimation` (p. 586) or in the editor **Properties** window (**Sprite Animation**). In Lua, get it with `animation:getSpriteAnimationComponent()`.

Properties

Name	Languages
`loop`	C++ | Lua
`frames`	C++
`framesSize`	C++ | Lua (read-only)
`framesTime`	C++
`framesTimeSize`	C++ | Lua (read-only)
`frameIndex`	C++ | Lua (read-only)
`frameTimeIndex`	C++ | Lua (read-only)
`spriteFrameCount`	C++ | Lua (read-only)

From Lua, change the frames with `SpriteAnimation:setAnimation` (p. 586).

`SpriteAnimationComponent` does not store a separate name. Sprite animations are entities, so their display and lookup name comes from the inherited `SpriteAnimation` / entity name (p. 586) (`EntityHandle::name`). Rename the entity in the Structure panel (or call `setName`) when you need a searchable label — for example with `Scene::findEntity` (p. 379).

`frames` is a list of sprite atlas **frame indices**; `framesTime` is the matching list of per-frame durations in **milliseconds**. `loop` restarts the sequence when it finishes. Without it, playback stops on the last frame. `frameIndex`, `frameTimeIndex`, and `spriteFrameCount` are runtime playback state (shown read-only in the editor).

SpriteComponent

The data behind a Sprite (p. 480). In Lua, get it with `sprite:getSpriteComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`width`	C++ | Lua
`height`	C++ | Lua
`automaticFlipY`	C++ | Lua
`flipY`	C++ | Lua
`textureScaleFactor`	C++ | Lua
`numFramesRect`	C++ | Lua (read-only)
`pivotPreset`	C++ | Lua
`needUpdateSprite`	C++ | Lua
`needUpdateBatch`	C++ | Lua

Set `needUpdateSprite` after changing the size, flip, or pivot. A size of 0 uses the texture size. Frames are C++ only; from Lua, use the Sprite (p. 480) frame methods. After changing the sprite's `MeshComponent` directly, set `needUpdateBatch`.

TerrainComponent

The data behind a Terrain (p. 486). In Lua, get it with `terrain:getTerrainComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`heightMap`	C++ | Lua
`blendMaps`	C++ | Lua
`surfaceLayers`	C++ | Lua
`foliageLayers`	C++ | Lua
`foliageDistanceScale`	C++ | Lua
`heightMapLoaded`	C++ | Lua (read-only)
`offset`	C++ | Lua
`terrainSize`	C++ | Lua
`maxHeight`	C++ | Lua
`textureBaseTiles`	C++ | Lua
`textureDetailTiles`	C++ | Lua
`autoSetRanges`	C++ | Lua (read-only)
`ranges`	C++ | Lua (read-only)
`resolution`	C++ | Lua (read-only)
`rootGridSize`	C++ | Lua (read-only)
`levels`	C++ | Lua (read-only)
`needUpdateTerrain`	C++ | Lua
`needUpdateTexture`	C++ | Lua
`needUpdateFoliage`	C++ | Lua

Change `resolution`, `rootGridSize`, and `levels` with the Terrain (p. 486) setters, which rebuild the terrain. Set `needUpdateTexture` after changing the maps or surface layers, and `needUpdateFoliage` after changing `foliageLayers`. In Lua, lists are copies: assign the changed table back.

TextComponent

The data behind a Text (p. 528). In Lua, get it with `text:getTextComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`loaded`	C++ | Lua (read-only)
`font`	C++ | Lua
`text`	C++ | Lua
`fontSize`	C++ | Lua
`multiline`	C++ | Lua
`maxTextSize`	C++ | Lua
`charPositions`	C++ | Lua (read-only)
`fixedWidth`	C++ | Lua
`fixedHeight`	C++ | Lua
`pivotBaseline`	C++ | Lua
`pivotCentered`	C++ | Lua
`needReloadAtlas`	C++ | Lua
`needUpdateText`	C++ | Lua

Set `needUpdateText` after changing the text or layout fields, and also `needReloadAtlas` after changing `font` or `fontSize`. `font` comes to Lua as a table of four font paths, and takes back only a table of four.

TextEditComponent

`TextEditComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`text`	C++ | Lua (read-only)
`selection`	C++ | Lua (read-only)
`cursor`	C++ | Lua (read-only)
`cursorBlink`	C++ | Lua
`cursorWidth`	C++ | Lua
`cursorColor`	C++ | Lua
`selectionColor`	C++ | Lua
`placeholderColor`	C++ | Lua
`placeholder`	C++ | Lua
`passwordChar`	C++ | Lua
`cursorIndex`	C++ | Lua
`selectionAnchor`	C++ | Lua
`scrollOffset`	C++ | Lua
`disabled`	C++ | Lua
`password`	C++ | Lua
`onChange`	C++ | Lua
`onSubmit`	C++ | Lua
`needUpdateTextEdit`	C++ | Lua

TilemapComponent

The data behind a Tilemap (p. 492). In Lua, get it with `tilemap:getTilemapComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`width`	C++ | Lua
`height`	C++ | Lua
`automaticFlipY`	C++ | Lua
`flipY`	C++ | Lua
`textureScaleFactor`	C++ | Lua
`reserveTiles`	C++ | Lua
`numTiles`	C++ | Lua
`needUpdateTilemap`	C++ | Lua
`tilesRect`	C++
`tiles`	C++

TimedActionComponent

`TimedActionComponent` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`time`	C++ | Lua (read-only)
`value`	C++ | Lua (read-only)
`duration`	C++ | Lua
`loop`	C++ | Lua
`pingPong`	C++ | Lua
`function`	C++ | Lua

Transform

Position, rotation, and scale of an entity, plus the world values and matrices the scene derives from them. Every Object (p. 455) has one. In Lua, get it with `object:getTransform()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`position`	C++ | Lua
`rotation`	C++ | Lua
`scale`	C++ | Lua
`worldPosition`	C++ | Lua (read-only)
`worldRotation`	C++ | Lua (read-only)
`worldScale`	C++ | Lua (read-only)
`localMatrix`	C++ | Lua (read-only)
`modelMatrix`	C++ | Lua (read-only)
`visible`	C++ | Lua
`visibleInHierarchy`	C++ | Lua (read-only)
`parent`	C++ | Lua (read-only)
`distanceToCamera`	C++ | Lua (read-only)
`billboardRotation`	C++ | Lua
`billboard`	C++ | Lua
`fakeBillboard`	C++ | Lua
`cylindricalBillboard`	C++ | Lua
`needUpdate`	C++ | Lua

Set `needUpdate` after writing `position`, `rotation`, `scale`, or `billboardRotation`. A hidden entity hides its children too, while each child keeps its own `visible` flag. `visibleInHierarchy` combines the flags of the entity and its ancestors and is refreshed by the render system on each update. Reparent with `addChild` (p. 459).

TranslateTracksComponent

The key values of a TranslateTracks (p. 590) action. In Lua, get it with `tracks:getTranslateTracksComponent()`.

Properties

> **Writing a component field**
>
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`values`	C++ | Lua
`inTangents`	C++ | Lua
`outTangents`	C++ | Lua

In Lua, lists are copies: assign the changed table back.

UIComponent

`UIComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`loaded`	C++ | Lua (read-only)
`minBufferCount`	C++ | Lua (read-only)
`minIndicesCount`	C++ | Lua (read-only)
`primitiveType`	C++ | Lua
`vertexCount`	C++ | Lua (read-only)
`texture`	C++ | Lua
`color`	C++ | Lua
`onGetFocus`	C++ | Lua
`onLostFocus`	C++ | Lua
`onPointerEnter`	C++ | Lua
`onPointerLeave`	C++ | Lua
`onPointerMove`	C++ | Lua
`onPointerDown`	C++ | Lua
`onPointerUp`	C++ | Lua
`onClick`	C++ | Lua
`onDoubleClick`	C++ | Lua
`onDragStart`	C++ | Lua
`onDrag`	C++ | Lua
`onDragEnd`	C++ | Lua
`automaticFlipY`	C++ | Lua
`flipY`	C++ | Lua
`pointerMoved`	C++ | Lua (read-only)
`focused`	C++ | Lua (read-only)
`needReload`	C++ | Lua
`needUpdateBuffer`	C++ | Lua
`needUpdateTexture`	C++ | Lua
`buffer`	C++
`indices`	C++
`render`	C++
`shader`	C++
`shaderProperties`	C++
`slotVSParams`	C++
`slotFSParams`	C++

Property details

color

The tint multiplied with the widget's texture, as **linear** RGBA — not sRGB.

The object wrappers convert for you: `Image::setColor` (p. 514), `Text::setColor` (p. 530) and the `Button` state colours all take an sRGB value and store the linear one, and their getters convert back. Writing `ui.color` directly, from C++ or Lua, sets the linear value with no conversion.

The same goes for a colour written into a `.scene` or `.bundle` file by hand or by a tool: it reaches this field unconverted, so it has to be authored linear. A panel written as `0.07` renders like sRGB `0.29`, not like the dark value it looks like in the file. The editor's colour picker already shows and edits the sRGB form of the stored value, so colours set in the Properties window need nothing.

```
-- These two produce the same pixels
image:setColor(Vector4(0.29, 0.29, 0.29, 1.0))   -- sRGB, converted for you
image:getUIComponent().color = Vector4(0.07, 0.07, 0.07, 1.0)  -- linear, as stored
```

UIContainerComponent

The data behind a Container (p. 510). In Lua, get it with `container:getUIContainerComponent()`.

Properties

> **Writing a component field**
>
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`type`	C++ | Lua
`useAllWrapSpace`	C++ | Lua
`wrapCellWidth`	C++ | Lua
`wrapCellHeight`	C++ | Lua
`numBoxes`	C++ | Lua (read-only)

UILayoutComponent

`UILayoutComponent` API exposed to Lua and C++ gameplay code.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`width`	C++ | Lua
`height`	C++ | Lua
`anchorPointLeft`	C++ | Lua
`anchorPointTop`	C++ | Lua
`anchorPointRight`	C++ | Lua
`anchorPointBottom`	C++ | Lua
`anchorOffsetLeft`	C++ | Lua
`anchorOffsetTop`	C++ | Lua
`anchorOffsetRight`	C++ | Lua
`anchorOffsetBottom`	C++ | Lua
`positionOffset`	C++ | Lua
`pivot`	C++ | Lua
`anchorPreset`	C++ | Lua
`usingAnchors`	C++ | Lua
`panel`	C++ | Lua (read-only)
`containerBoxIndex`	C++ | Lua (read-only)
`scissor`	C++ | Lua
`ignoreScissor`	C++ | Lua
`ignoreEvents`	C++ | Lua
`needUpdateSizes`	C++ | Lua

WaterComponent

The data behind Water (p. 496). In Lua, get it with `water:getWaterComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`loaded`	C++ | Lua (read-only)
`aabb`	C++ | Lua (read-only)
`worldAABB`	C++ | Lua (read-only)
`customShader`	C++ | Lua
`customUnderwaterShader`	C++ | Lua
`size`	C++ | Lua
`subdivisions`	C++ | Lua
`shallowColor`	C++ | Lua
`deepColor`	C++ | Lua
`depthFade`	C++ | Lua
`waveHeight`	C++ | Lua
`waveLength`	C++ | Lua
`waveSpeed`	C++ | Lua
`waveDirection`	C++ | Lua
`waveSteepness`	C++ | Lua
`normalTexture`	C++ | Lua
`normalScale`	C++ | Lua
`normalStrength`	C++ | Lua
`rippleSpeed`	C++ | Lua
`reflectivity`	C++ | Lua
`specularIntensity`	C++ | Lua
`roughness`	C++ | Lua
`foamColor`	C++ | Lua
`shoreFoam`	C++ | Lua
`crestFoam`	C++ | Lua
`receiveShadows`	C++ | Lua
`depthEffects`	C++ | Lua
`planarReflection`	C++ | Lua
`reflectionDistortion`	C++ | Lua
`refraction`	C++ | Lua
`refractionDistortion`	C++ | Lua
`caustics`	C++ | Lua
`causticsScale`	C++ | Lua
`underwater`	C++ | Lua
`buoyancy`	C++ | Lua

Name	Languages
`buoyancyDepth`	C++ | Lua
`wakeArea`	C++ | Lua
`wakeFade`	C++ | Lua
`needUpdateTexture`	C++ | Lua
`needReload`	C++ | Lua

The colors are linear here, while the Water (p. 496) setters take sRGB. Set `needReload` after changing `size`, `subdivisions`, or a custom shader, and `needUpdateTexture` after replacing `normalTexture`.

WaterExclusionComponent

The data behind a WaterExclusion (p. 503). In Lua, get it with `exclusion:getWaterExclusionComponent()`.

Properties

> **Writing a component field**
> Some fields take effect only once the matching `needUpdate…` flag is set too. See Writing components directly (p. 191).

Name	Languages
`shape`	C++ | Lua
`center`	C++ | Lua
`size`	C++ | Lua
`wakeFoam`	C++ | Lua
`wakeWaves`	C++ | Lua
`aabb`	C++ | Lua (read-only)
`worldAABB`	C++ | Lua (read-only)

`center` and `size` place a `BOX` or `SPHERE` shape. A `HULL` is built from the entity's meshes. `wakeFoam` and `wakeWaves` are the wake (p. 504) it leaves while it moves.

I/O

Data

Inherits: FileData
C++ type: `Data`

Description

An in-memory byte buffer that implements the `FileData` streaming interface. It can be loaded from a file path, from a raw byte pointer, or from another File (p. 705) object. Once loaded, it supports sequential read/write operations and random seek access.

In native exports, a filename lookup checks the optional `resources.pak` before falling back to the filesystem. This makes `Data` suitable for read-only packaged resources; direct File (p. 705) handles can only open loose files.

`Data` is used by the engine for audio sample loading, asset bundling, and anywhere that in-memory file-like access is needed.

Methods

Type	Name	Langs
unsigned int	open	C++ | Lua
int	eof	C++ | Lua
unsigned int	read	C++ | Lua
unsigned int	write	C++ | Lua
unsigned int	length	C++ | Lua
void	seek	C++ | Lua
unsigned int	pos	C++ | Lua

Method details

open

- unsigned int **open**(unsigned char* aData, unsigned int aDataLength, bool aCopy = false, bool aTakeOwnership = true)
- unsigned int **open**(const char* aFilename)
- unsigned int **open**(File* aFile)

Loads data from a raw byte pointer, a file path, or a File (p. 705) object. The filename overload transparently reads packed `asset://`, `lua://`, and asset-relative paths in a native export. When `aCopy` is `true`, the data is duplicated into an internal buffer; when `aTakeOwnership` is `true`, the `Data` object takes ownership of the provided pointer. The `File` overload reads from the file's current position, and returns `FileErrors::INVALID_PARAMETER` when the file is not open. Lua has the filename overload only.

The `Data(path)` constructor, which `FileData.newFile(path)` also uses, logs an error when the file can't be opened and leaves the data empty. `open` doesn't log, so check what it returns to tell a missing file from an empty one.

eof

- int **eof**()

Returns non-zero (`true` in Lua) if the read cursor is at or past the end of the buffer.

read / write

- unsigned int **read**(unsigned char* aDst, unsigned int aBytes)
- unsigned int **write**(unsigned char* aSrc, unsigned int aBytes)

Read or write `aBytes` bytes at the current cursor position. Returns the number of bytes actually transferred.

length

- unsigned int **length**()

Returns the total size of the buffer in bytes.

seek

- void **seek**(int aOffset)

Moves the read/write cursor to `aOffset` bytes from the beginning. A negative offset counts back from the end, and the cursor always stays between the start and the end of the data.

pos

- unsigned int **pos**()

Returns the current cursor position.

File

Inherits: FileData (p. 706)
C++ type: `File`

Disk file access. `File` requires a real filesystem entry and cannot open assets or Lua files stored in an exported `resources.pak`; use Data (p. 703) or an engine resource loader for packed read-only resources.

Methods

Name	Languages
`open`	C++ | Lua
`flush`	C++ | Lua
`close`	C++ | Lua

Failed opens

`open` returns a `FileErrors` value, and the `File(path, write)` constructor logs an error when the file can't be opened. A `File` without an open file acts like an empty one: reads and writes return `0` or an empty string, and `eof` is true. To tell a missing file from an empty one, check what `open` returns:

Lua

```lua
local save = File()
if save:open("data://save.dat") ~= FileErrors.FILEDATA_OK then
    -- no save yet
end
```

C++

```cpp
File save;
if (save.open("data://save.dat") != FileErrors::FILEDATA_OK) {
    // no save yet
}
```

`close` is safe to call more than once, and the file closes itself when the object is destroyed. In C++ a `File` owns its handle, so it can't be copied.

FileData

Abstract streaming interface shared by File (p. 705) and Data (p. 703).

`newFile` constructs the concrete type. Pass `useHandle = true` to request a disk `File`; omit it or pass `false` for an in-memory `Data` buffer. In a native export with resource packing enabled, a packed path still returns `Data` even when a handle was requested, because a packed entry has no filesystem file to open.

In Lua, the object is released when it is collected, which also closes a file, and it comes back as a `FileData`: the methods of this page work on it, the ones only `File` or `Data` have don't. In C++ you own the returned pointer and must `delete` it.

`seek` and `eof` work the same on both types: a negative offset counts back from the end, and `eof` is true right after the last byte. Only a `File` can move past the end, to write after a gap. On a pipe or FIFO, `eof` never waits for data, so it only turns true after a read has hit the end.

Methods

Name	Languages
`newFile`	C++ | Lua
`getBaseDir`	C++ | Lua
`getFilePathExtension`	C++ | Lua
`getSystemPath`	C++ | Lua
`read8`	C++ | Lua
`read16`	C++ | Lua
`read32`	C++ | Lua
`eof`	C++ | Lua
`length`	C++ | Lua
`seek`	C++ | Lua
`pos`	C++ | Lua
`readString`	C++ | Lua
`writeString`	C++ | Lua

UserSettings

C++ type: `UserSettings` (static)

Description

A cross-platform key-value store for persistent user preferences. Values survive app restarts and are stored in a platform-appropriate location (e.g. `SharedPreferences` on Android, `NSUserDefaults` on iOS, a settings file on desktop platforms).

Supported value types: `bool`, `int`, `long`, `float`, `double`, `std::string`, and `Data`.

Methods

Type	Name	Langs
static bool	getBoolForKey	C++ | Lua
static int	getIntegerForKey	C++ | Lua
static long	getLongForKey	C++ | Lua
static float	getFloatForKey	C++ | Lua
static double	getDoubleForKey	C++ | Lua
static std::string	getStringForKey	C++ | Lua
static Data (p. 703)	getDataForKey	C++ | Lua
static void	setBoolForKey	C++ | Lua
static void	setIntegerForKey	C++ | Lua
static void	setLongForKey	C++ | Lua
static void	setFloatForKey	C++ | Lua
static void	setDoubleForKey	C++ | Lua
static void	setStringForKey	C++ | Lua
static void	setDataForKey	C++ | Lua
static void	removeKey	C++ | Lua

Method details

get methods

- static T **get<Type>ForKey**(const char* key)
- static T **get<Type>ForKey**(const char* key, T defaultValue)

Retrieve a stored value by key. If the key does not exist and no `defaultValue` is given, the zero/empty value for the type is returned.

C++

```
int volume = UserSettings::getIntegerForKey("volume", 80);
bool musicEnabled = UserSettings::getBoolForKey("music", true);
std::string playerName = UserSettings::getStringForKey("name", "Player1");
```

Lua

```
local volume = UserSettings.getIntegerForKey("volume", 80)
local musicEnabled = UserSettings.getBoolForKey("music", true)
local playerName = UserSettings.getStringForKey("name", "Player1")
```

set methods

- static void **set<Type>ForKey**(const char* key, T value)

Store a value. Creates the key if it does not exist, or overwrites the previous value.

C++

```
UserSettings::setIntegerForKey("highScore", 9999);
UserSettings::setBoolForKey("tutorialDone", true);
```

Lua

```
UserSettings.setIntegerForKey("highScore", 9999)
UserSettings.setBoolForKey("tutorialDone", true)
```

removeKey

- static void **removeKey**(const char* key)

Deletes a key and its associated value from persistent storage.

Other

ActionFrame

`ActionFrame` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`startTime`	C++ | Lua
`duration`	C++ | Lua
`action`	C++ | Lua

`startTime` is the offset in seconds from the animation start at which `action` runs.

`duration` is how long the frame lasts, in seconds. A value of `0` (the default) means **auto**: the frame follows the action's own duration — a `SpriteAnimation` lasts as long as its frame sequence, a `TimedAction` uses its duration, a nested `Animation` its own total length.

Base64

C++ type: `Base64`

Description

Static helpers for encoding and decoding binary data as base64 strings. Useful for embedding small payloads in JSON/YAML or for network/save serialization.

`encode` and `decode` are **C++ only** — they are not exposed to Lua yet.

Methods

Type	Name	Langs
static std::string	encode	C++
static std::vector<unsigned char>	decode	C++

Color

C++ type: `Color` (static utility class)

Description

A static utility class providing colour space conversion between **linear** (physically-based) and **sRGB** (perceptually uniform, display-encoded) colour spaces.

Doriax's rendering pipeline works in linear colour space. When you specify a colour constant (e.g. a material's `baseColorFactor`) it should typically be in linear space. If you are reading a colour from a design tool or HTML hex code, it is likely in sRGB space and needs conversion before use.

Methods

Type	Name	Langs
static Vector3	linearTosRGB	C++ | Lua
static Vector4	linearTosRGB	C++ | Lua
static Vector3	sRGBToLinear	C++ | Lua
static Vector4	sRGBToLinear	C++ | Lua

Method details

linearTosRGB

- static Vector3 **linearTosRGB**(float r, float g, float b)
- static Vector3 **linearTosRGB**(Vector3 color)
- static Vector4 **linearTosRGB**(float r, float g, float b, float a)
- static Vector4 **linearTosRGB**(Vector4 color)

Converts a linear colour to sRGB display-encoded space. Use this when you need to display a physically-accurate linear value to the user as a readable colour (e.g. in a UI colour picker).

C++

```
// Convert the material base colour to sRGB for display
Vector3 displayColor = Color::linearTosRGB(material.baseColorFactor);
```

Lua

```
local displayColor = Color.linearTosRGB(material.baseColorFactor)
```

sRGBToLinear

- static Vector3 **sRGBToLinear**(float r, float g, float b)
- static Vector3 **sRGBToLinear**(Vector3 srgbIn)
- static Vector4 **sRGBToLinear**(float r, float g, float b, float a)
- static Vector4 **sRGBToLinear**(Vector4 srgbIn)

Converts an sRGB colour to linear space. Use this before assigning a colour from a design tool or a hex code to a Material (p. 734) or light property.

C++

```
// Web colour #FF8040 converted to linear for PBR rendering
Vector3 linearOrange = Color::sRGBToLinear(1.0f, 0.502f, 0.251f);
mat.baseColorFactor = Vector4(linearOrange, 1.0f);
```

Lua

```lua
local linearOrange = Color.sRGBToLinear(1.0, 0.502, 0.251)
-- Prefer assigning in C++ or via mesh:setColor; Material fields are read-only in Lua.
mesh:setColor(Vector4(linearOrange, 1.0))
```

EntityHandle

C++ type: `EntityHandle`

Description

The base class for all high-level Doriax objects that wrap an ECS entity. `EntityHandle` stores a reference to its owning `Scene` and its `Entity` identifier, giving C++ and Lua code a convenient handle to a scene entity without having to interact with the raw ECS API.

Most classes in the Doriax API — Object (p. 455), Action (p. 569), Sound (p. 475), Skybox (p. 472), etc. — inherit from `EntityHandle`.

Properties

Type	Name	Default	Langs
std::string	name	`""`	C++ | Lua
bool	entityOwned	`true`	C++ | Lua

Methods

Type	Name	Langs
Scene*	getScene	C++
Entity	getEntity	C++

Property details

name

- *Setter*: void **setName**(const std::string& name)
- *Getter*: std::string **getName**() const

Human-readable tag for this entity. Used by the editor to display entities in the scene hierarchy, and can be used in game code for debugging or lookup.

entityOwned

- *Setter*: void **setEntityOwned**(bool entityOwned)
- *Getter*: bool **isEntityOwned**() const

Whether this handle *owns* its entity. When `true`, the entity is destroyed (and any sub-resources it set up are cleared) when the handle is destroyed. Set it to `false` to detach the handle from the entity's lifetime.

The value is decided by which constructor you use:

Constructor	Owns the entity?
`Type(scene)`	`true` — creates a **new** entity
`Type(scene, entity)`	`false` — **wraps** an existing entity

Ownership and lifetime

`EntityHandle` follows single-owner semantics so an entity is never destroyed twice:

- **Copying** produces a non-owning observer. `Type b = a;` leaves `a` as the sole owner (if it was one) and `b` referencing the same entity without owning it.
- **Moving** transfers ownership. `Type b = std::move(a);` makes `b` the owner and leaves `a` as a non-owning observer.
- **Wrapping** an existing entity (`Type(scene, entity)`) never owns it, so references you pass around — including handles resolved from script properties or `Scene:findEntity` (p. 379) — are safe to copy and discard.
- **Destroying the entity yourself** is fine: an owning handle whose entity no longer exists skips it when the handle is destroyed.
- **Outliving the scene** releases the ownership. When a scene is deleted, for example when `SceneManager.loadScene` (p. 603) loads its stack again, its owning handles stop owning, so a Lua handle collected later leaves the new scene alone.

> **Keep owning handles alive**
> An owning handle destroys its entity when it is itself destroyed — in Lua that happens when the handle is garbage-collected. If you create an entity at runtime with `Type(scene)` and want it to persist, store the handle (e.g. in `self`), or call `setEntityOwned(false)` to hand the entity's lifetime to the scene.

Method details

getScene

- Scene* **getScene**() const

Returns a pointer to the owning Scene (p. 390).

getEntity

- Entity **getEntity**() const

Returns the raw ECS entity identifier. Use this when interfacing with the low-level EntityManager (p. 378) or EntityRegistry (p. 379) APIs.

FramebufferRender

`FramebufferRender` API exposed to Lua and C++ gameplay code.

Methods

Name	Languages
`createFramebuffer`	C++ | Lua
`destroyFramebuffer`	C++ | Lua
`isCreated`	C++ | Lua
`getColorTexture`	C++ | Lua

InstanceData

`InstanceData` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`position`	C++ | Lua
`rotation`	C++ | Lua
`scale`	C++ | Lua
`color`	C++ | Lua
`textureRect`	C++ | Lua
`visible`	C++ | Lua

LineData

`LineData` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`pointA`	C++ | Lua
`pointB`	C++ | Lua
`colorA`	C++ | Lua
`colorB`	C++ | Lua

MeshPolygon

Inherits: Mesh (p. 436)
C++ type: `MeshPolygon`

`MeshPolygon` API exposed to Lua and C++ gameplay code.

Properties

> **`width` and `height` are derived**
> They come from the vertex list, so neither language has a setter. `addVertex` marks the polygon dirty and `MeshSystem` recomputes them on the next update, so they lag a frame behind the vertices.

Name	Languages
`width`	C++ | Lua (read-only)
`height`	C++ | Lua (read-only)
`flipY`	C++ | Lua

Methods

Name	Languages
`addVertex`	C++ | Lua
`getMeshPolygonComponent` (MeshPolygonComponent (p. 680))	Lua

OverallBuildProgress

`OverallBuildProgress` API exposed to Lua and C++ gameplay code.

Properties

> **Read-only from Lua**
> A progress snapshot: the build threads fill it in, scripts read it.

Name	Languages
`totalProgress`	C++ | Lua (read-only)
`totalBuilds`	C++ | Lua (read-only)
`completedBuilds`	C++ | Lua (read-only)
`currentBuildName`	C++ | Lua (read-only)
`currentBuildType`	C++ | Lua (read-only)
`hasActiveBuilds`	C++ | Lua (read-only)

PointData

`PointData` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`position`	C++ | Lua
`color`	C++ | Lua
`size`	C++ | Lua
`rotation`	C++ | Lua
`textureRect`	C++ | Lua
`visible`	C++ | Lua

PolygonPoint

A vertex of a Polygon (p. 521) or a MeshPolygon (p. 713). `PolygonPoint()` creates one in Lua.

Properties

Name	Languages
`position`	C++ | Lua
`color`	C++ | Lua

`color` is a linear vertex color, multiplied by the color of the polygon. `addVertex` leaves it white, so set per-vertex colors through the component's `points` list.

PostProcessPass

C++ type: `PostProcessPass` (struct)

Description

One entry of a scene's post-process chain: the forked fullscreen shader to run, whether it runs, and the values of the parameters it declares. Passes are plain values — build them, then hand the whole chain to Scene.postProcessPasses (p. 401). To change a value on a chain that is already running, use Scene.setPostProcessUniform (p. 405), which does not rebuild the chain.

See Custom Shaders — Post-process passes (p. 149) for how to write a pass.

Constructors

- `PostProcessPass()` — the built-in passthrough, enabled, with no values.

Properties

Type	Name	Default	Languages
string	shader	`""`	C++ | Lua
bool	enabled	`true`	C++ | Lua
vector<pair<string, Vector4>>	uniforms	`{}`	C++

Methods

Returns	Name	Languages
void	setUniform	C++ | Lua
Vector4	getUniform	C++ | Lua
bool	removeUniform	C++ | Lua

Property details

shader

Project-relative base path of the forked shader (`"shaders/sharpen"` resolves to `shaders/sharpen.vert` and `shaders/sharpen.frag`, or `"a.vert|b.frag"` names the files separately). Empty runs the built-in passthrough, which is also what a fork that fails to compile falls back to.

enabled

Whether the pass runs. A disabled pass keeps its shader and values, and is still exported, so it can be turned back on at runtime with Scene.setPostProcessPassEnabled (p. 406).

Method details

setUniform

- `void setUniform(const std::string& name, const Vector4& value)`
- `void setUniform(const std::string& name, const Vector3& value)`
- `void setUniform(const std::string& name, const Vector2& value)`
- `void setUniform(const std::string& name, float value)`

- `Vector4 getUniform(const std::string& name) const`
- `bool removeUniform(const std::string& name)`

Values of the parameters the shader declares with `#pragma param`, keyed by name and stored in `uniforms`. A name the shader does not declare is stored but unused, and a parameter with no value reads its declared default. A narrower value leaves the remaining components at zero. These edit the pass value only; the scene picks the change up when the chain is set again (a value-only change keeps the compiled chain).

C++

```
PostProcessPass vignette;
vignette.shader = "shaders/vignette";
vignette.setUniform("strength", 0.6f);
vignette.setUniform("color", Vector3(0.0f, 0.0f, 0.1f));

scene.setPostProcessPasses({vignette});
```

Lua

```
local vignette = PostProcessPass()
vignette.shader = "shaders/vignette"
vignette:setUniform("strength", 0.6)
vignette:setUniform("color", Vector3(0, 0, 0.1))

scene.postProcessPasses = {vignette}
```

RayReturn

Result of a ray cast (hit point, normal, distance).

Properties

> **Read-only from Lua**
> A raycast result: the systems fill it in, scripts read it.

Name	Languages
`hit`	C++ | Lua (read-only)
`distance`	C++ | Lua (read-only)
`point`	C++ | Lua (read-only)
`normal`	C++ | Lua (read-only)
`body`	C++ | Lua (read-only)
`shapeIndex`	C++ | Lua (read-only)

`distance` is where the hit lies along the ray, as a fraction from 0 at its origin to 1 at `origin + direction`. `point` is the hit position in world space.

ResourceBuildInfo

`ResourceBuildInfo` API exposed to Lua and C++ gameplay code.

Properties

> **Read-only from Lua**
> A progress snapshot: the build threads fill it in, scripts read it.

Name	Languages
`type`	C++ | Lua (read-only)
`name`	C++ | Lua (read-only)
`progress`	C++ | Lua (read-only)
`isActive`	C++ | Lua (read-only)
`startTime`	C++ | Lua (read-only)

ResourceProgress

Async resource build progress.

Methods

Name	Languages
`startBuild`	C++ | Lua
`updateProgress`	C++ | Lua
`completeBuild`	C++ | Lua
`failBuild`	C++ | Lua
`hasActiveBuilds`	C++ | Lua
`getOverallProgress`	C++ | Lua
`getActiveBuildCount`	C++ | Lua
`getAllActiveBuilds`	C++ | Lua
`getCurrentBuild`	C++ | Lua
`getResourceTypeName`	C++ | Lua

TerrainFoliageLayer

One foliage layer of a Terrain (p. 486): which model it scatters, where, and how densely. `TerrainFoliageLayer()` creates one in Lua.

Properties

Name	Languages
`meshPath`	C++ | Lua
`densityMap`	C++ | Lua
`density`	C++ | Lua
`minScale`	C++ | Lua
`maxScale`	C++ | Lua
`rotationJitter`	C++ | Lua
`alignToNormal`	C++ | Lua
`minSlope`	C++ | Lua
`maxSlope`	C++ | Lua
`minHeight`	C++ | Lua
`maxHeight`	C++ | Lua
`drawDistance`	C++ | Lua
`minDrawDistance`	C++ | Lua
`seed`	C++ | Lua
`castShadows`	C++ | Lua
`windSway`	C++ | Lua

Slopes are in degrees. `minHeight` and `maxHeight` are fractions of the terrain's `maxHeight`. `windSway` bends the instances toward the scene wind (p. 285): how far their tips move in 10 m/s, with the base still (`0`, the default, keeps them still).

TextureLoadResult

`TextureLoadResult` API exposed to Lua and C++ gameplay code.

Properties

> **Read-only from Lua**
> A load result: the loader fills it in, scripts read it.

Name	Languages
`id`	C++ | Lua (read-only)
`state`	C++ | Lua (read-only)
`errorMessage`	C++ | Lua (read-only)
`data`	C++ | Lua (read-only)

TextureRender

`TextureRender` API exposed to Lua and C++ gameplay code.

ThreadPoolManager

Worker threads for async loading.

Methods

Name	Languages
`getInstance`	C++ | Lua
`initialize`	C++ | Lua
`shutdown`	C++ | Lua
`getQueueSize`	C++ | Lua

TileData

`TileData` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`name`	C++ | Lua
`rectId`	C++ | Lua
`position`	C++ | Lua
`width`	C++ | Lua
`height`	C++ | Lua

TileRectData

`TileRectData` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`name`	C++ | Lua
`submeshId`	C++ | Lua
`rect`	C++ | Lua
`collision`	C++ | Lua

`collision` is `true` by default. Set it to `false` to leave this rect's tiles out of a tilemap shape (p. 547).

Particles

ParticleAccelerationInitializer

`ParticleAccelerationInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minAcceleration`	C++ | Lua
`maxAcceleration`	C++ | Lua

ParticleAccelerationModifier

`ParticleAccelerationModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromAcceleration`	C++ | Lua
`toAcceleration`	C++ | Lua
`function`	C++ | Lua

ParticleAlphaInitializer

`ParticleAlphaInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minAlpha`	C++ | Lua
`maxAlpha`	C++ | Lua

ParticleAlphaModifier

`ParticleAlphaModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromAlpha`	C++ | Lua
`toAlpha`	C++ | Lua
`function`	C++ | Lua

ParticleBurst

`ParticleBurst` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`time`	C++ | Lua
`minCount`	C++ | Lua
`maxCount`	C++ | Lua

ParticleColorGradient

`ParticleColorGradient` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`stops`	C++ | Lua
`useSRGB`	C++ | Lua

ParticleColorGradientStop

`ParticleColorGradientStop` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`time`	C++ | Lua
`color`	C++ | Lua

ParticleColorInitializer

`ParticleColorInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minColor`	C++ | Lua
`maxColor`	C++ | Lua
`useSRGB`	C++ | Lua

ParticleColorModifier

`ParticleColorModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromColor`	C++ | Lua
`toColor`	C++ | Lua
`function`	C++ | Lua
`useSRGB`	C++ | Lua

ParticleLifeInitializer

`ParticleLifeInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minLife`	C++ | Lua
`maxLife`	C++ | Lua

ParticlePositionInitializer

`ParticlePositionInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`shape`	C++ | Lua
`minPosition`	C++ | Lua
`maxPosition`	C++ | Lua
`radius`	C++ | Lua
`innerRadius`	C++ | Lua
`coneAngle`	C++ | Lua
`coneHeight`	C++ | Lua

ParticlePositionModifier

`ParticlePositionModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromPosition`	C++ | Lua
`toPosition`	C++ | Lua
`function`	C++ | Lua

ParticleRotationInitializer

`ParticleRotationInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minRotation`	C++ | Lua
`maxRotation`	C++ | Lua
`shortestPath`	C++ | Lua

ParticleRotationModifier

`ParticleRotationModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromRotation`	C++ | Lua
`toRotation`	C++ | Lua
`function`	C++ | Lua
`shortestPath`	C++ | Lua

Particles

Inherits: Action (p. 569)
C++ type: `Particles`

Description

A GPU-accelerated particle emitter driven by the Action (p. 569) system. `Particles` handles birth, lifetime, movement, colour, and death for large numbers of small quads (sprites) to create effects such as fire, smoke, sparks, rain, and explosions.

The emitter has two layers of per-particle customisation:

- **Initializers** — applied once at birth to set the particle's starting value.
- **Modifiers** — applied over the particle's lifetime to interpolate a value from one state to another, with optional easing.

Constructors

Available in C++ and Lua:

- `Particles(Scene* scene)` — creates a new entity.
- `Particles(Scene* scene, Entity entity)` — wraps an existing entity without taking ownership.

In Lua, use `Particles(scene, entity)` to access an existing action. The entity must already have the components required by this type; wrapping does not add them. See EntityHandle ownership (p. 712).

Properties

Type	Name	Default	Langs
unsigned int	maxParticles	`100`	C++ | Lua
int	rate	`10`	C++ | Lua
int	maxPerUpdate	`10`	C++ | Lua
bool	emitter	`true`	C++ | Lua
bool	loop	`true`	C++ | Lua
bool	localSpace	`false`	C++ | Lua
ParticleEmitterShape	emitterShape	`Box`	C++ | Lua

Methods

Type	Name	Langs
void	reset	C++ | Lua
void	setLifeInitializer	C++ | Lua
void	setPositionInitializer	C++ | Lua
void	setSpherePositionInitializer	C++ | Lua
void	setHemispherePositionInitializer	C++ | Lua
void	setCirclePositionInitializer	C++ | Lua
void	setConePositionInitializer	C++ | Lua
void	setPositionModifier	C++ | Lua
void	addBurst	C++ | Lua
void	setBursts	C++ | Lua
void	clearBursts	C++ | Lua
void	setVelocityInitializer	C++ | Lua

Type	Name	Langs
void	setVelocityModifier	C++ | Lua
void	setAccelerationInitializer	C++ | Lua
void	setAccelerationModifier	C++ | Lua
void	setColorInitializer	C++ | Lua
void	setColorModifier	C++ | Lua
void	addColorGradientStop	C++ | Lua
void	setColorGradient	C++ | Lua
void	clearColorGradient	C++ | Lua
void	setColorGradientUseSRGB	C++ | Lua
void	setAlphaInitializer	C++ | Lua
void	setAlphaModifier	C++ | Lua
void	setSizeInitializer	C++ | Lua
void	setSizeModifier	C++ | Lua
void	setSpriteIntializer	C++ | Lua
void	setSpriteModifier	C++ | Lua
void	setRotationInitializer	C++ | Lua
void	setRotationModifier	C++ | Lua
void	setScaleInitializer	C++ | Lua
void	setScaleModifier	C++ | Lua
ParticlesComponent (p. 683)	`getParticlesComponent`	Lua

Enumerations

ParticleEmitterShape

The volume from which new particles are spawned.

- **Box** — Particles spawn within a box region defined by setPositionInitializer.
- **Sphere** — Particles spawn within a sphere. Configure with setSpherePositionInitializer.
- **Hemisphere** — Particles spawn within the upper hemisphere. Configure with setHemispherePositionInitializer.
- **Circle** — Particles spawn on a flat circle. Configure with setCirclePositionInitializer.
- **Cone** — Particles spawn inside a cone pointing upward. Configure with setConePositionInitializer.

Property details

maxParticles

- *Setter*: void **setMaxParticles**(unsigned int maxParticles)
- *Getter*: unsigned int **getMaxParticles**() const

The maximum number of particles alive at any time. Increase this for denser effects; be mindful of GPU/CPU budget.

rate

- *Setter*: void **setRate**(int rate)
- *Getter*: int **getRate**() const

Number of new particles spawned per second while the emitter is active.

maxPerUpdate

- *Setter*: void **setMaxPerUpdate**(int maxPerUpdate)
- *Getter*: int **getMaxPerUpdate**() const

Caps the number of new particles spawned in a single update tick. Prevents a very long delta time (e.g. after a frame spike) from spawning hundreds of particles in one frame.

emitter

- *Setter*: void **setEmitter**(bool emitter)
- *Getter*: bool **isEmitter**() const

When `true`, the particle system actively spawns new particles at the configured rate. Set to `false` to stop new emissions while allowing existing particles to finish their lifetime.

loop

- *Setter*: void **setLoop**(bool loop)
- *Getter*: bool **isLoop**() const

When `true`, the particle system restarts automatically after all particles have died. When `false`, the system stops after the first complete emission cycle.

localSpace

- *Setter*: void **setLocalSpace**(bool localSpace)
- *Getter*: bool **isLocalSpace**() const

When `true`, particle positions are in local object space — they move with the emitter entity. When `false` (default), positions are in world space — particles continue along their trajectories even after the emitter moves.

emitterShape

- *Setter*: void **setPositionInitializerShape**(ParticleEmitterShape shape)
- *Getter*: ParticleEmitterShape **getPositionInitializerShape**() const

The spawn volume for new particles. See ParticleEmitterShape.

Method details

reset

- void **reset**()

Immediately destroys all live particles and resets the emitter to its initial state.

setLifeInitializer

- void **setLifeInitializer**(float life)
- void **setLifeInitializer**(float minLife, float maxLife)

Set how long (in seconds) each particle lives. The two-argument form randomises lifetime between `minLife` and `maxLife`.

setPositionInitializer

- void **setPositionInitializer**(Vector3 position)
- void **setPositionInitializer**(Vector3 minPosition, Vector3 maxPosition)

Sets the spawn position (or random range) relative to the emitter. For Box shape, `minPosition` and `maxPosition` define the box corners.

C++

```
Particles fire(&scene);
fire.setLifeInitializer(1.5f, 2.5f);
fire.setPositionInitializer(Vector3(-0.5f, 0, -0.5f), Vector3(0.5f, 0, 0.5f));
fire.setVelocityInitializer(Vector3(-0.2f, 1.0f, -0.2f), Vector3(0.2f, 2.0f, 0.2f));
fire.start();
```

Lua

```
local fire = Particles(scene)
fire:setLifeInitializer(1.5, 2.5)
fire:setPositionInitializer(Vector3(-0.5, 0, -0.5), Vector3(0.5, 0, 0.5))
fire:setVelocityInitializer(Vector3(-0.2, 1.0, -0.2), Vector3(0.2, 2.0, 0.2))
fire:start()
```

setSpherePositionInitializer

- void **setSpherePositionInitializer**(float radius)
- void **setSpherePositionInitializer**(float radius, float innerRadius)

Spawns particles within a sphere of the given radius. `innerRadius` creates a hollow sphere (donut shape) — particles only spawn in the shell between `innerRadius` and `radius`.

setHemispherePositionInitializer

- void **setHemispherePositionInitializer**(float radius)
- void **setHemispherePositionInitializer**(float radius, float innerRadius)

Spawns particles within the upper hemisphere (Y ≥ 0). Useful for upward explosions or fountain effects.

setCirclePositionInitializer

- void **setCirclePositionInitializer**(float radius)
- void **setCirclePositionInitializer**(float radius, float innerRadius)

Spawns particles on a flat circle in the XZ plane. `innerRadius` creates a ring.

setConePositionInitializer

- void **setConePositionInitializer**(float angle, float height)

Spawns particles inside a cone. `angle` is the half-angle of the cone (in degrees or radians depending on `Engine::useDegrees`), and `height` is its vertical extent.

setPositionModifier

- void **setPositionModifier**(float fromTime, float toTime, Vector3 fromPosition, Vector3 toPosition)
- void **setPositionModifier**(float fromTime, float toTime, Vector3 fromPosition, Vector3 toPosition, EaseType functionType)

Adds a position offset over the particle's normalised lifetime (0 = birth, 1 = death). The `fromTime` and `toTime` values are in the `[0, 1]` range. An optional EaseType (p. 588) controls interpolation.

addBurst / setBursts / clearBursts

- void **addBurst**(float time, int count)
- void **addBurst**(float time, int minCount, int maxCount)
- void **setBursts**(std::vector<ParticleBurst> bursts)
- void **clearBursts**()

Bursts spawn a fixed number of particles at a specific time (in seconds) during the emitter's cycle, independent of rate. Useful for explosions.

C++

```cpp
Particles explosion(&scene);
explosion.setEmitter(false);
explosion.addBurst(0.0f, 50, 80);  // spawn 50–80 particles at t=0
explosion.setLoop(false);
explosion.start();
```

Lua

```lua
local explosion = Particles(scene)
explosion.emitter = false
explosion:addBurst(0.0, 50, 80)
explosion.loop = false
explosion:start()
```

setVelocityInitializer

- void **setVelocityInitializer**(Vector3 velocity)
- void **setVelocityInitializer**(Vector3 minVelocity, Vector3 maxVelocity)

Sets the initial velocity (units per second) at birth. The range form picks a random velocity for each particle between `minVelocity` and `maxVelocity`.

setVelocityModifier

- void **setVelocityModifier**(float fromTime, float toTime, Vector3 fromVelocity, Vector3 toVelocity)
- void **setVelocityModifier**(float fromTime, float toTime, Vector3 fromVelocity, Vector3 toVelocity, EaseType functionType)

Interpolates the particle's velocity offset over its normalised lifetime. Times are in the `[0, 1]` range.

setAccelerationInitializer

- void **setAccelerationInitializer**(Vector3 acceleration)
- void **setAccelerationInitializer**(Vector3 minAcceleration, Vector3 maxAcceleration)

Sets a constant per-particle acceleration (like gravity) applied from birth. Use `Vector3(0, -9.81f, 0)` to simulate gravity.

setAccelerationModifier

- void **setAccelerationModifier**(float fromTime, float toTime, Vector3 fromAcceleration, Vector3 toAcceleration)
- void **setAccelerationModifier**(float fromTime, float toTime, Vector3 fromAcceleration, Vector3 toAcceleration, EaseType functionType)

Animates the per-particle acceleration offset over its normalised lifetime.

setColorInitializer

- void **setColorInitializer**(Vector3 color)
- void **setColorInitializer**(Vector3 minColor, Vector3 maxColor)

Sets the particle's initial linear-space RGB colour. The range form randomises colour per particle.

setColorModifier

- void **setColorModifier**(float fromTime, float toTime, Vector3 fromColor, Vector3 toColor)
- void **setColorModifier**(float fromTime, float toTime, Vector3 fromColor, Vector3 toColor, EaseType functionType)

Lerps colour over the particle's normalised lifetime. Times are in `[0, 1]`.

addColorGradientStop / setColorGradient / clearColorGradient / setColorGradientUseSRGB

- void **addColorGradientStop**(float time, Vector3 color)
- void **setColorGradient**(std::vector<ParticleColorGradientStop> stops)
- void **clearColorGradient**()
- void **setColorGradientUseSRGB**(bool useSRGB)

Defines a multi-stop colour gradient applied over the particle's lifetime instead of a simple two-point modifier. `addColorGradientStop()` adds one stop at `time` (0–1). `setColorGradientUseSRGB()` enables perceptually-linear interpolation through sRGB colour space.

C++

```cpp
fire.addColorGradientStop(0.0f, Vector3(1.0f, 0.6f, 0.0f));  // orange
fire.addColorGradientStop(0.5f, Vector3(1.0f, 0.1f, 0.0f));  // red
fire.addColorGradientStop(1.0f, Vector3(0.2f, 0.2f, 0.2f));  // smoke grey
```

Lua

```lua
fire:addColorGradientStop(0.0, Vector3(1.0, 0.6, 0.0))
fire:addColorGradientStop(0.5, Vector3(1.0, 0.1, 0.0))
fire:addColorGradientStop(1.0, Vector3(0.2, 0.2, 0.2))
```

setAlphaInitializer

- void **setAlphaInitializer**(float alpha)
- void **setAlphaInitializer**(float minAlpha, float maxAlpha)

Sets the initial opacity for each particle. `1.0` = fully opaque, `0.0` = invisible.

setAlphaModifier

- void **setAlphaModifier**(float fromTime, float toTime, float fromAlpha, float toAlpha)
- void **setAlphaModifier**(float fromTime, float toTime, float fromAlpha, float toAlpha, EaseType functionType)

Animates opacity over the particle's normalised lifetime. For a fade-out effect, use `fromTime=0, toTime=1, fromAlpha=1, toAlpha=0`.

setSizeInitializer

- void **setSizeInitializer**(float size)
- void **setSizeInitializer**(float minSize, float maxSize)

Sets the initial uniform size of each particle in world units.

setSizeModifier

- void **setSizeModifier**(float fromTime, float toTime, float fromSize, float toSize)
- void **setSizeModifier**(float fromTime, float toTime, float fromSize, float toSize, EaseType functionType)

Animates particle size over its normalised lifetime.

setSpriteInitializer

- void **setSpriteIntializer**(std::vector<int> frames)
- void **setSpriteIntializer**(int minFrame, int maxFrame)

Sets which sprite atlas frame each particle shows at birth. The vector form provides an explicit list to pick from randomly; the two-integer form picks a random frame in the range.

Requires the emitter mesh to have an atlas configured (see Sprite (p. 480) frame definitions).

setSpriteModifier

- void **setSpriteModifier**(float fromTime, float toTime, std::vector<int> frames)
- void **setSpriteModifier**(float fromTime, float toTime, std::vector<int> frames, EaseType functionType)

Plays `frames` in order between `fromTime` and `toTime` of each particle's life, in seconds, ending on the last frame. With the default linear easing each frame gets an equal share of that time.

setRotationInitializer

- void **setRotationInitializer**(Quaternion rotation)
- void **setRotationInitializer**(float rotation)
- void **setRotationInitializer**(Quaternion minRotation, Quaternion maxRotation)
- void **setRotationInitializer**(float minRotation, float maxRotation)

Sets the initial rotation of each particle. The scalar overload takes an angle (degrees or radians per `Engine::useDegrees`) around the Z axis for 2D-style billboard rotation.

setRotationModifier

- void **setRotationModifier**(float fromTime, float toTime, float fromRotation, float toRotation)
- void **setRotationModifier**(float fromTime, float toTime, Quaternion fromRotation, Quaternion toRotation)
- void **setRotationModifier**(float fromTime, float toTime, float fromRotation, float toRotation, EaseType functionType)

Animates rotation over the particle's normalised lifetime.

setScaleInitializer

- void **setScaleInitializer**(float scale)
- void **setScaleInitializer**(Vector3 scale)
- void **setScaleInitializer**(float minScale, float maxScale)
- void **setScaleInitializer**(Vector3 minScale, Vector3 maxScale)

Sets the initial scale at birth. Use the `Vector3` overload for non-uniform scale.

setScaleModifier

- void **setScaleModifier**(float fromTime, float toTime, float fromScale, float toScale)
- void **setScaleModifier**(float fromTime, float toTime, Vector3 fromScale, Vector3 toScale)
- void **setScaleModifier**(float fromTime, float toTime, Vector3 fromScale, Vector3 toScale, EaseType functionType)

Animates the particle's scale over its normalised lifetime. Useful for particles that shrink as they die.

ParticleScaleInitializer

`ParticleScaleInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minScale`	C++ | Lua
`maxScale`	C++ | Lua

ParticleScaleModifier

`ParticleScaleModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromScale`	C++ | Lua
`toScale`	C++ | Lua
`function`	C++ | Lua

ParticleSizeInitializer

`ParticleSizeInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minSize`	C++ | Lua
`maxSize`	C++ | Lua

ParticleSizeModifier

`ParticleSizeModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromSize`	C++ | Lua
`toSize`	C++ | Lua
`function`	C++ | Lua

ParticleSpriteInitializer

`ParticleSpriteInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`frames`	C++ | Lua

ParticleSpriteModifier

`ParticleSpriteModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`frames`	C++ | Lua
`function`	C++ | Lua

ParticleVelocityInitializer

`ParticleVelocityInitializer` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`minVelocity`	C++ | Lua
`maxVelocity`	C++ | Lua

ParticleVelocityModifier

`ParticleVelocityModifier` API exposed to Lua and C++ gameplay code.

Properties

Name	Languages
`fromTime`	C++ | Lua
`toTime`	C++ | Lua
`fromVelocity`	C++ | Lua
`toVelocity`	C++ | Lua
`function`	C++ | Lua

Textures

Framebuffer

Off-screen render target.

Properties

Name	Languages
`minFilter`	C++ | Lua
`magFilter`	C++ | Lua

Methods

Name	Languages
`create`	C++ | Lua
`destroy`	C++ | Lua
`isCreated`	C++ | Lua
`getRender`	C++ | Lua
`getVersion`	C++ | Lua
`create`	C++ | Lua
`width`	C++ | Lua
`height`	C++ | Lua

Material

C++ type: `Material` (struct)

Description

A PBR (Physically Based Rendering) material definition that drives how a mesh surface looks under lighting. `Material` is a plain data struct — it holds colour factors, metallic/roughness values, and up to five texture slots that correspond to the standard glTF 2.0 PBR workflow.

Materials are attached to Mesh (p. 436) objects (or individual sub-meshes) via `Mesh::setMaterial()`. When loading a Model (p. 450), each sub-mesh automatically receives the material defined in the source file.

> **Lua binding is read-only**
> Lua can read material fields from `mesh:getMaterial()`, but there is no `Material()` constructor and fields cannot be assigned from Lua. Create or edit materials in C++, or use editor `.material` files and mesh colour helpers like `mesh:setColor(...)`.

Material files (`.material`)

In the editor, materials can be saved as standalone `.material` YAML files and **linked** from multiple mesh submeshes. All linked meshes read from the same file, so one edit updates every user of that material.

Task	Editor workflow
Create a file	Drag the material preview from Properties into the Resources Browser
Apply to a mesh	Drag the `.material` file onto a mesh in the Scene view or onto the Material row in Properties
Share across meshes	Link each submesh to the same file
Break the link	Click the unlink button next to the material name in Properties

Linked materials reload when the file changes on disk. See Resources Browser — Material files (p. 108).

Properties

Type	Name	Default	Langs
Vector4	baseColorFactor	`(1,1,1,1)`	C++ | Lua (read-only)
float	metallicFactor	`1.0`	C++ | Lua (read-only)
float	roughnessFactor	`1.0`	C++ | Lua (read-only)
MaterialAlphaMode	alphaMode	`AUTO`	C++ | Lua (read-only)
float	alphaCutoff	`0.5`	C++ | Lua (read-only)
Vector3	emissiveFactor	`(0,0,0)`	C++ | Lua (read-only)
float	foliageTransmission	`0`	C++ | Lua (read-only)
Texture (p. 738)	baseColorTexture	empty	C++ | Lua (read-only)
Texture (p. 738)	emissiveTexture	empty	C++ | Lua (read-only)
Texture (p. 738)	metallicRoughnessTexture	empty	C++ | Lua (read-only)
Texture (p. 738)	occlusionTexture	empty	C++ | Lua (read-only)
Texture (p. 738)	normalTexture	empty	C++ | Lua (read-only)
std::string	name	`""`	C++

Property details

baseColorFactor

The base linear-space RGBA colour multiplied with `baseColorTexture`. When no texture is assigned, this is the solid colour of the surface. How the combined alpha is interpreted depends on alphaMode.

C++

```
Material mat;
mat.baseColorFactor = Vector4(1.0f, 0.0f, 0.0f, 1.0f);  // solid red
mesh.setMaterial(mat);
```

Lua

```
-- Materials cannot be constructed or mutated from Lua yet.
-- Prefer editor .material files, or tint with mesh:setColor(...).
mesh:setColor(Vector4(1.0, 0.0, 0.0, 1.0))
```

alphaMode / alphaCutoff

`alphaMode` controls how the combined alpha from `baseColorFactor.a` and the base-colour texture is rendered:

Mode	Behaviour
`MaterialAlphaMode::AUTO`	Legacy Doriax behaviour. `autoTransparency` selects the transparent pass when the texture or base-colour factor contains transparency. The submesh's **Texture Shadow** option remains a manual cutout control.
`MaterialAlphaMode::ALPHA_OPAQUE`	Ignores material alpha and renders every surviving fragment fully opaque.
`MaterialAlphaMode::MASK`	Discards fragments whose combined alpha is below `alphaCutoff`; surviving fragments are fully opaque.
`MaterialAlphaMode::BLEND`	Preserves alpha for conventional transparency and selects the transparent pass when `autoTransparency` is enabled. Depth testing stays on; depth writes are disabled in the colour pass.

`alphaCutoff` is used only by `MASK` and defaults to `0.5`. The same combined-alpha test is used by the lit pass, shadow/depth pass, and SSR G-buffer, so the visible surface, its shadow, and screen-space effects keep the same silhouette.

```
Material leaves;
leaves.baseColorTexture = Texture("textures/leaves.png");
leaves.alphaMode = MaterialAlphaMode::MASK;
leaves.alphaCutoff = 0.35f;
mesh.setMaterial(leaves);
```

Alpha-mode materials must be authored in C++ or as editor `.material` files; Lua cannot construct or mutate `Material` instances.

GLTF/GLB loading maps the source material's `OPAQUE`, `MASK`, or `BLEND` mode directly onto `MaterialAlphaMode::ALPHA_OPAQUE`, `MASK`, and `BLEND`. Editor-created materials default to `AUTO` for compatibility with older projects.

metallicFactor / roughnessFactor

Controls the PBR metallic-roughness workflow:

- **metallicFactor** (`0.0`–`1.0`) — `1.0` is a fully metallic surface (like polished steel), `0.0` is dielectric (like plastic or wood). Multiplied with the blue channel of `metallicRoughnessTexture`.
- **roughnessFactor** (`0.0`–`1.0`) — `0.0` is a mirror-like glossy surface, `1.0` is fully diffuse. Multiplied with the green channel of `metallicRoughnessTexture`.

emissiveFactor

An additive linear-space RGB glow colour. The surface emits light of this colour independent of external lighting. Multiplied with `emissiveTexture` when present. Keep all channels at `0` (the default) to disable emission.

foliageTransmission

Light let through a thin surface when a light is behind it, `0`–`1` (values are clamped). Each light adds its intensity times the base colour, this factor, how directly the light shines through, and the same shadow term as direct lighting, so a leaf glows only where the sun actually reaches its back. `0` (the default) disables it; keep it there on opaque geometry. Unlit materials ignore it.

Imported glTF materials read it from the material extra `doriax_foliage_transmission`; in the editor it is the **Leaf Transmission** row of the Material and is kept as a submesh override.

C++

```
Material leaves = mesh.getMaterial(0);
leaves.foliageTransmission = 0.3f;
mesh.setMaterial(0, leaves);
```

baseColorTexture

A Texture (p. 738) that modulates the `baseColorFactor`. The texture's RGBA channels are multiplied component-wise with `baseColorFactor`. Supports 2D textures loaded from file or created from pixel data.

C++

```
Material mat;
mat.baseColorTexture = Texture("textures/ground_albedo.png");
mesh.setMaterial(mat);
```

Lua

```
-- Read an existing material; assignment is not supported from Lua.
local mat = mesh:getMaterial()
Log.print("Base color factor: " .. tostring(mat.baseColorFactor))
```

emissiveTexture

A Texture (p. 738) whose RGB channels are multiplied with `emissiveFactor` to produce surface emission. Useful for glowing screens, neon signs, or lava surfaces.

metallicRoughnessTexture

A Texture (p. 738) encoding metallic information in the **blue channel** and roughness in the **green channel** (following the glTF 2.0 specification). The red channel is unused.

occlusionTexture

A Texture (p. 738) used for baked ambient-occlusion. The red channel is sampled to darken areas that receive less indirect light (crevices, seams). Scale is fixed at `1.0`.

normalTexture

A tangent-space normal map Texture (p. 738). Perturbs the surface normal per-pixel to simulate fine geometric detail without additional polygons.

name

An optional human-readable identifier for the material, usually set automatically when loading a Model (p. 450) from a GLTF file. Can be used to look up or display material information at runtime.

Texture

C++ type: `Texture`

Description

Represents a GPU texture resource. `Texture` supports 2D textures, cube maps, and framebuffer textures, covering the full range from loading a simple PNG file to driving a render-to-texture pipeline.

A `Texture` object is a lightweight handle; the actual GPU resource is managed internally by the texture pool and shared between objects that reference the same file path. Textures are loaded lazily — the GPU upload happens the first time the texture is needed for rendering.

`Texture` is most commonly used as a field in Material (p. 734) or passed to `Mesh::setTexture()`.

Properties

Type	Name	Default	Langs
TextureFilter	minFilter	`LINEAR`	C++ | Lua
TextureFilter	magFilter	`LINEAR`	C++ | Lua
TextureWrap	wrapU	`REPEAT`	C++ | Lua
TextureWrap	wrapV	`REPEAT`	C++ | Lua
float	svgScale	`1.0`	C++ | Lua
TextureAlphaBorder	alphaBorder	`AUTO`	C++ | Lua
bool	releaseDataAfterLoad	`true`	C++ | Lua

Methods

Type	Name	Langs
void	setPath	C++ | Lua
void	setData	C++ | Lua
void	setId	C++ | Lua
void	setCubeMap	C++
void	setCubePaths	C++ | Lua
void	setCubeDatas	C++
void	setFramebuffer	C++ | Lua
TextureLoadResult (p. 718)	load	C++ | Lua
void	destroy	C++ | Lua
std::string	getPath	C++ | Lua
std::string	getId	C++ | Lua
unsigned int	getWidth	C++ | Lua
unsigned int	getHeight	C++ | Lua
bool	isTransparent	C++ | Lua
bool	isCubeMap	C++ | Lua
bool	empty	C++ | Lua
bool	isFramebuffer	C++ | Lua
bool	isFramebufferOutdated	C++ | Lua
void	releaseData	C++ | Lua

Enumerations

TextureFilter

Filtering modes applied when the texture is sampled at a scale different from its native resolution.

- **NEAREST** — No interpolation (nearest-neighbour). Produces a pixelated look; useful for pixel art.
- **LINEAR** — Bilinear interpolation between the four nearest texels.
- **NEAREST_MIPMAP_NEAREST** — Nearest-neighbour within the nearest mipmap level.
- **NEAREST_MIPMAP_LINEAR** — Nearest-neighbour, with linear interpolation between mipmap levels.
- **LINEAR_MIPMAP_NEAREST** — Bilinear within the nearest mipmap level.
- **LINEAR_MIPMAP_LINEAR** — Trilinear filtering (bilinear + mipmap interpolation). Best quality for 3D surfaces.

TextureWrap

Controls what happens when UV coordinates fall outside the `[0, 1]` range.

- **REPEAT** — The texture tiles continuously. Default for most use cases.
- **MIRRORED_REPEAT** — The texture tiles but mirrors on every repeat, producing a seamless look.
- **CLAMP_TO_EDGE** — UV values are clamped; the edge pixel is stretched.
- **CLAMP_TO_BORDER** — UV values outside `[0, 1]` sample a fixed border colour.

TextureAlphaBorder

Whether an image loaded from a file gets its alpha border fixed.

- **AUTO** — Fixed on sprites, tilemaps, polygons, UI and particles, kept everywhere else.
- **FIX** — Always fixed.
- **KEEP** — Loaded as the file stores it.

Property details

minFilter / magFilter

- *Setter*: void **setMinFilter**(TextureFilter filter)
- *Getter*: TextureFilter **getMinFilter**() const
- *Setter*: void **setMagFilter**(TextureFilter filter)
- *Getter*: TextureFilter **getMagFilter**() const

`minFilter` applies when the texture is rendered smaller than its native resolution (minification). `magFilter` applies when it is rendered larger (magnification). Only `NEAREST` and `LINEAR` are valid values for `magFilter`.

wrapU / wrapV

- *Setter*: void **setWrapU**(TextureWrap wrapU)
- *Getter*: TextureWrap **getWrapU**() const
- *Setter*: void **setWrapV**(TextureWrap wrapV)
- *Getter*: TextureWrap **getWrapV**() const

Wrapping mode along the horizontal (U) and vertical (V) texture coordinates.

Filter and wrap modes are part of the GPU texture, so an image used with different settings is uploaded once for each combination.

svgScale

- *Setter*: void **setSvgScale**(float scale)
- *Getter*: float **getSvgScale**() const

Rasterization scale for `.svg` path sources (`2.0` = twice the intrinsic resolution). Has no effect on raster images or framebuffer textures. Invalid values (zero or negative) snap back to `1.0`.

The scale is part of the texture's identity: the same SVG file at two different scales produces two independent GPU textures, so different slots can share one file at different resolutions. Changing the scale on an SVG source releases the current GPU texture and reloads at the new resolution.

C++

```
Texture icon("ui/icon.svg");
icon.setSvgScale(4.0f);  // a 24x24 SVG rasterizes as 96x96
```

Lua

```
local icon = Texture("ui/icon.svg")
icon.svgScale = 4  -- a 24x24 SVG rasterizes as 96x96
```

alphaBorder

- *Setter*: void **setAlphaBorder**(TextureAlphaBorder alphaBorder)
- *Getter*: TextureAlphaBorder **getAlphaBorder**() const

Linear filtering also reads the color of fully transparent pixels next to visible ones, which image editors often save as black, and shows it as a grey or dark outline around the image. With the fix, those pixels take the color of their visible neighbors when the file loads, up to 4 pixels out. Alpha and visible pixels do not change.

`AUTO` fixes it where the image is drawn with transparency (sprites, tilemaps, polygons, UI and particles) and keeps the file as is elsewhere. That protects 3D materials drawn opaque, whose alpha is ignored and whose transparent pixels are visible, and data maps that store other values in their channels. Set `FIX` for a 3D texture that is alpha tested or blended, such as foliage.

Like svgScale, the fix is part of the texture's identity: the same file fixed and kept is two GPU textures. It only applies to textures loaded from a file.

C++

```
Texture leaves("models/leaves.png");
leaves.setAlphaBorder(TextureAlphaBorder::FIX);
```

Lua

```
local leaves = Texture("models/leaves.png")
leaves.alphaBorder = TextureAlphaBorder.FIX
```

releaseDataAfterLoad

- *Setter*: void **setReleaseDataAfterLoad**(bool release)
- *Getter*: bool **isReleaseDataAfterLoad**() const

Frees the CPU-side pixel data as soon as the texture reaches the GPU. `true` by default, but textures built from a TextureData (p. 744) keep their pixels: the `Texture(id, data)` constructor, setData and setCubeDatas all clear the flag, as does setFramebuffer. Setting it back on hands those pixels to the texture, so don't free them yourself. See releaseData to free the pixels on demand.

Method details

setPath

• void **setPath**(const std::string& path)

Sets the file path for a 2D texture. The file is not loaded immediately; loading is deferred to the first time the texture is needed for rendering.

C++

```
Texture tex;
tex.setPath("textures/wall.png");
material.baseColorTexture = tex;
```

Lua

```
local tex = Texture()
tex:setPath("textures/wall.png")
Log.print("Texture path: " .. tex:getPath())
```

`Material` fields are read-only in Lua. To assign this path to a mesh, use `mesh:setTexture(tex:getPath())`.

Alternatively, pass the path directly to the constructor: `Texture("textures/wall.png")`.

Setting a new path resets svgScale to `1.0`. The legacy form `"icon.svg?svgScale=4"` is still accepted: the suffix is absorbed into the `svgScale` property and `getPath()` returns the clean file path.

setData

• void **setData**(const std::string& id, TextureData data)

Creates a texture from in-memory pixel data (TextureData (p. 744)). The `id` is an arbitrary string used as a cache key; using the same id for identical pixel data avoids uploading duplicates.

C++

```
TextureData td;
td.loadTextureFromFile("textures/icon.png");

Texture tex;
tex.setData("icon", td);
```

Lua

```
local td = TextureData()
td:loadTextureFromFile("textures/icon.png")

local tex = Texture()
tex:setData("icon", td)
```

setId

• void **setId**(const std::string& id)

Manually assigns a cache key without providing pixel data. Useful for referencing a texture that is already in the pool from another code path.

setCubeMap

• void **setCubeMap**(const std::string& path)

Convenience overload that treats `path` as a directory containing six face images in the conventional naming format (`front`, `back`, `left`, `right`, `up`, `down`). Exact naming depends on the engine's convention.

setCubePaths

- void **setCubePaths**(const std::string& front, const std::string& back, const std::string& left, const std::string& right, const std::string& up, const std::string& down)

Loads a cube map from six individual image files. Typically used for skyboxes or environment reflections.

C++

```
Texture sky;
sky.setCubePaths("sky/px.png", "sky/nx.png",
                 "sky/ny.png", "sky/py.png",
                 "sky/pz.png", "sky/nz.png");
```

Lua

```
local sky = Texture()
sky:setCubePaths("sky/px.png", "sky/nx.png",
                 "sky/ny.png", "sky/py.png",
                 "sky/pz.png", "sky/nz.png")
```

setCubeDatas

- void **setCubeDatas**(const std::string& id, TextureData front, TextureData back, TextureData left, TextureData right, TextureData up, TextureData down)

Creates a cube-map texture from six in-memory TextureData (p. 744) faces. The `id` is used as a pool cache key.

setFramebuffer

- void **setFramebuffer**(Framebuffer (p. 733)* framebuffer)

Points the texture at a render target so it can be sampled like any other texture. Also clears releaseDataAfterLoad, unlike the `Texture(Framebuffer*)` constructor, which leaves it set.

load

- TextureLoadResult (p. 718) **load**()

Forces an immediate CPU-side load of the texture data from disk. This is normally not required because loading is deferred, but you can call it to pre-warm assets during a loading screen.

destroy

- void **destroy**()

Releases the GPU resource and removes the texture from the pool. After calling this, the texture object is empty.

getPath

- std::string **getPath**(size_t index = 0) const

Returns the file path associated with this texture. For cube maps, `index` selects the face (0 = front, 1 = back, 2 = left, 3 = right, 4 = up, 5 = down).

getId

- std::string **getId**() const

Returns the internal cache key used to identify this texture in the pool. For an SVG source with a non-default svgScale, the id encodes the scale (`"icon.svg?svgScale=4"`) because each scale is a separate pool entry; use getPath when you need the plain file path.

getWidth / getHeight

- unsigned int **getWidth**() const
- unsigned int **getHeight**() const

Returns the pixel dimensions of the loaded texture. Returns `0` if the texture has not been loaded yet.

isTransparent

- bool **isTransparent**() const

Returns `true` if the texture image contains pixels with alpha below `1.0`. Used internally by the engine to determine whether the owning mesh should be rendered in the transparent pass.

isCubeMap

- bool **isCubeMap**() const

Returns `true` if this texture is a cube map (six faces).

empty

- bool **empty**() const

Returns `true` if no path, data, or framebuffer has been assigned to this texture.

isFramebufferOutdated

- bool **isFramebufferOutdated**() const

Returns `true` when the render target has been resized or recreated since the texture last bound it, so the caller knows to rebind.

isFramebuffer

- bool **isFramebuffer**() const

Returns `true` if this texture was created from a Framebuffer (p. 733) (render-to-texture).

releaseData

- void **releaseData**()

Frees the CPU-side pixel data after it has been uploaded to the GPU. Reduces memory usage on platforms with limited RAM. The texture remains valid for rendering.

TextureData

CPU pixel buffer for textures. Copying one with `dataOwned` set duplicates its pixels, so the copy outlives the original. Any other copy points at the same pixels.

Properties

Name	Languages
`dataOwned`	C++ | Lua
`svgScale`	C++ | Lua

Methods

Name	Languages
`loadTextureFromFile`	C++ | Lua
`releaseImageData`	C++ | Lua
`hasAlpha`	C++ | Lua
`resizePowerOfTwo`	C++ | Lua
`fitPowerOfTwo`	C++ | Lua
`resize`	C++ | Lua
`crop`	C++ | Lua
`fitSize`	C++ | Lua
`flipVertical`	C++ | Lua
`getColorComponent`	C++ | Lua
`getWidth`	C++ | Lua
`getHeight`	C++ | Lua
`getOriginalWidth`	C++ | Lua
`getOriginalHeight`	C++ | Lua
`getSize`	C++ | Lua
`getColorFormat`	C++ | Lua
`getChannels`	C++ | Lua
`getData`	C++ | Lua
`isTransparent`	C++ | Lua
`getMinNearestPowerOfTwo`	C++ | Lua

PART 6

Building from Source

Overview 746
Build Script 748
Windows 750
Linux 752
macOS 754
Android 756
iOS 758
HTML5 760

Building — Overview

There are two things you might "build" with Doriax:

1. **The editor itself**, from source, using CMake.
2. **Your game project**, exported from the editor to a target platform.

This section focuses on building from source and the per-platform requirements.

> **Desktop and Web builds without a manual toolchain step**
> The Export Window's **Desktop** and **Web** modes compile your game for you — the manual per-platform steps here apply to **Source Code** exports and to building the editor or engine itself. See Export Window (p. 156).

> **Transition in progress**
> Doriax is the continuation of Supernova Engine. Some platform tooling and folder names are still being refreshed under the Doriax name, so a few steps may reference legacy paths. When in doubt, the CMake commands below are the source of truth.

Building the editor from source

Doriax uses **CMake**. The root project builds the desktop editor target `doriax-editor`.

```
git clone https://github.com/doriaxengine/doriax.git
cd doriax
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

- On **single-config** generators (Ninja, Makefiles) the executable is created under `build/`.
- On **multi-config** generators (Visual Studio, Xcode) it is created under a configuration subdirectory such as `build/Release/`.

The root editor build picks its platform backend from the host OS (`API_BACKEND` is a detection, not a choice: `linux`, `win`, or `mac`). Linux and Windows render the editor with OpenGL Core by default and accept `-DGRAPHIC_BACKEND=vulkan`; macOS is always Metal. For the editor, `GRAPHIC_BACKEND` is a **cached** variable, so a build directory keeps the backend it was configured with — reconfiguring without the flag will not silently switch it back.

Building a runtime project from source

The `engine/` directory can also be used as a standalone runtime build. When built as the top-level CMake project, it looks for a project root at `engine/project` unless `PROJECT_ROOT` is set.

```
cmake -S engine -B build-runtime -DCMAKE_BUILD_TYPE=Debug
cmake --build build-runtime --config Debug --target doriax-project
```

Useful runtime variables:

Variable	Purpose
`APP_NAME`	Output target/application name
`PROJECT_ROOT`	Project folder containing assets, Lua, and source files
`GRAPHIC_BACKEND`	`glcore`, `gles3`, `d3d11`, `metal`, or `vulkan`

The target system picks the app backend, so there is no variable to set. See App backend (p. 359).

The build script (p. 748) runs these commands in one step: `python3 engine/doriax.py` here, or `python3 doriax.py` in a Source Code export.

Per-platform guides

Target	Guide
Windows	Building for Windows (p. 750)
Linux	Building for Linux (p. 752)
macOS	Building for macOS (p. 754)
Android	Building for Android (p. 756)
iOS	Building for iOS (p. 758)
HTML5	Building for HTML5 (p. 760)

Build options

Doriax exposes CMake options to control how a project is built. Two common ones control the scripting entry point:

Option	Effect
`NO_CPP_INIT`	Disable the C++ `init()` entry point
`NO_LUA_INIT`	Disable the Lua entry point
`DORIAX_SHARED`	Build the engine as a shared library where supported
`EMSCRIPTEN_THREAD_SUPPORT`	Enable pthread support for Emscripten builds

Pass them at configure time, for example:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug -DNO_CPP_INIT=1
```

This is useful when a project mixes Lua and C++ and you want only one to drive scene setup. See Scripting (p. 195) for the rationale.

Generated files

The build can generate support files before compiling:

File	Source
`generated/shaders.h`	Built from shaders under `engine/shaderlib`
`generated/engine_api_suggestions.h`	Built from Lua binding files and engine headers

These generated files support the editor shader library and code editor suggestions.

Build Script (doriax.py)

`doriax.py` runs the CMake configure and build steps for a Doriax project in one command. Every Source Code export (p. 157) puts it in the export root, next to `CMakeLists.txt`. The engine repository has it at `engine/doriax.py`, where it builds `engine/project` by default.

> **Desktop and Web exports are already built**
> The Export Window's **Desktop** and **Web** modes compile the game for you. The script is for **Source Code** exports.

Requirements

- Python 3.9 or newer
- CMake[24] 3.20 or newer, on `PATH`
- The toolchain of the target platform, see the platform guides (p. 747)
- Optional: Ninja[25], used for Linux, macOS, and Web builds when it is on `PATH`

Usage

From the export folder:

```
python3 doriax.py              # Release build for this computer
python3 doriax.py --debug      # Debug build
python3 doriax.py -g vulkan    # choose the graphics backend
python3 doriax.py -p web       # Emscripten build
```

On Windows, run it with `python` or `py`.

Platforms

`--platform`	Builds on	Generator	Output
`linux`	Linux	Ninja if installed	`build/linux/<App>`
`windows`	Windows	Newest Visual Studio	`build/windows/Release/<App>.exe`
`macos`	macOS	Ninja if installed	`build/macos/<App>`
`macos-xcode`	macOS	Xcode	`build/macos-xcode/Release/<App>.app`
`ios-xcode`	macOS	Xcode	`build/ios-xcode/Release-iphonesimulator/<App>.app`
`web`	Any	Ninja if installed	`build/web/<App>.html`

The default is the computer's own platform. Desktop platforms build only on their own operating system. `<App>` is the project name the export wrote into `CMakeLists.txt`, and Debug builds use `Debug` instead of `Release` in the paths.

Android builds go through Gradle. Open the exported Android Studio workspace instead, see Building for Android (p. 756).

Options

Option	Default	Effect
`-p`, `--platform`	This computer	Target platform

24. https://cmake.org/download/
25. https://ninja-build.org/

Option	Default	Effect
`-d`, `--debug` / `--no-debug`	Release	Build type
`-b`, `--build` / `--no-build`	Build	`--no-build` only configures, for example to open the Xcode or Visual Studio project
`-g`, `--graphic-backend`	Platform default	`glcore`, `gles3`, `metal`, `d3d11`, or `vulkan`, see Graphics backend (p. 358)
`-G`, `--generator`	See Platforms	CMake generator
`-j`, `--jobs`	CPU cores	Parallel build jobs
`-D`, `--define`		Extra CMake definition, repeatable: `-DEMSCRIPTEN_THREAD_SUPPORT=ON`
`-o`, `--output`	`build/<platform>`	Build directory
`-a`, `--appname`	From `CMakeLists.txt`	Executable name (`APP_NAME`)
`-s`, `--project`	`project/`	Project root (`PROJECT_ROOT`)
`-e`, `--doriax`	Script folder	Engine root with the engine `CMakeLists.txt`
`--no-cpp-init` / `--no-lua-init`		Define `NO_CPP_INIT` / `NO_LUA_INIT`, see Scripting entry options (p. 357)
`--em-shell-file`		Emscripten `--shell-file` template for `web`

The build directory keeps its generator and CMake cache between runs, including `--graphic-backend`, `--appname`, and `-D` values. To start over, delete the build directory.

Web builds

The script finds `emcmake` in the `EMSDK`, `EMSCRIPTEN_ROOT`, or `EMSCRIPTEN` folder, or on `PATH`. Running `emsdk_env` sets `EMSDK`. On Windows it builds with Ninja, from `PATH` or from the C++ CMake tools of Visual Studio.

An exported project also gets `<App>.export.html`, the page with the Web settings (p. 60) applied, and a copy of it named `index.html` (unless the project itself is named `index`), so serving the build folder opens the game. Publish it with `<App>.js`, `<App>.wasm`, `favicon.png`, and, when present, `<App>.data`, not the plain Emscripten page `<App>.html`. Serve the files over HTTP as in Building for HTML5 (p. 761).

Xcode builds

`macos-xcode` and `ios-xcode` generate `build/<platform>/<App>.xcodeproj`. CMake copies `assets/` and `lua/` into the app bundle after each build, and the script also adds them to the project's resources after each configure. macOS `glcore` and `vulkan` builds are plain executables, not bundles, so they skip this step.

`ios-xcode` builds for the iOS Simulator. For devices, see Building for iOS (p. 758).

Building for Windows

1. Install dependencies

- CMake[26] 3.20 or newer, added to `PATH`
- **Visual Studio 2022** with Desktop development with C++
- GLFW[27] 3.4 development package, or SDL2 if building with `-DAPI_BACKEND=sdl`
- Python 3 for generated editor API suggestion files

2. Clone the repository

```
git clone https://github.com/doriaxengine/doriax.git
cd doriax
```

3. Configure and build

Using the Visual Studio generator:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -G "Visual Studio 17 2022"
cmake --build build --config Release --target doriax-editor
```

Visual Studio is a **multi-config** generator, so the resulting `doriax-editor` executable is placed under a configuration subdirectory such as `build/Release/`.

The Windows runtime defaults to Direct3D 11 for standalone projects. You can request OpenGL Core instead with `-DGRAPHIC_BACKEND=glcore` when configuring a runtime build.

> **Faster builds with Ninja**
>
> If you have Ninja installed you can use it instead of the Visual Studio generator:
>
> ```
> cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -G "Ninja"
> cmake --build build --config Release --target doriax-editor
> ```
>
> With Ninja the executable is created directly under `build/`.

Building with MinGW (GCC)

The default and recommended Windows toolchain is MSVC. You can instead build with a MinGW-w64 GCC toolchain — for example from MSYS2[28] — with `gcc`, `g++` and `ninja` on `PATH`:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -G "Ninja" ^
  -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
cmake --build build --config Release --target doriax-editor
```

Launch that editor from the same MSYS2 terminal so it finds `g++`, `mingw32-make` and the runtime DLLs. Its **Compiler → Default** then resolves to your GCC toolchain on its own, because the MSVC kits are ABI-incompatible with a MinGW editor and are greyed out.

26. https://cmake.org/download/
27. https://www.glfw.org/

> **A MinGW editor only loads MinGW C++ plugins**
> The editor and your compiled C++ scripts share a single engine instance, so they must use the same C++ ABI. A MinGW-built editor can therefore only build and load **MinGW GCC** plugins — not MSVC ones — while the official MSVC download is the reverse; the two never mix. Most users should stay on the MSVC build and only choose MinGW if they specifically need a GCC plugin toolchain. See C++ Build Setup → Choosing a compiler (p. 209).

Runtime project build

For an exported or standalone runtime project, configure the engine directory and pass the project root if it is not the default `engine/project` folder:

```
cmake -S engine -B build-runtime -DPROJECT_ROOT="C:/path/to/project" -DCMAKE_BUILD_TYPE=Release
cmake --build build-runtime --config Release --target doriax-project
```

Use the generated executable from the selected configuration folder.

In a Source Code export, `python doriax.py` runs both commands with the newest Visual Studio. Pass `-G` to choose another generator. See Build Script (p. 748).

Vulkan backend

To build a Windows runtime with the Vulkan backend, install the Vulkan SDK[29] and set the `VULKAN_SDK` environment variable to your SDK path. Pass `-DGRAPHIC_BACKEND=vulkan` at configure time:

```
cmake -S engine -B build-vulkan ^
  -DPROJECT_ROOT="C:/path/to/project" ^
  -DGRAPHIC_BACKEND=vulkan ^
  -DCMAKE_BUILD_TYPE=Release ^
  -G "Ninja"
cmake --build build-vulkan --config Release --target doriax-project
```

Shaders for exported Vulkan projects are compiled to SPIR-V by the editor's shader builder.

29. https://vulkan.lunarg.com/

Building for Linux

1. Install dependencies

On Debian/Ubuntu-based distributions, install the development packages:

```
sudo apt update
sudo apt install -y \
  build-essential cmake ninja-build pkg-config python3 \
  libglfw3-dev libxi-dev libxcursor-dev libxrandr-dev libxinerama-dev \
  libwayland-dev libxkbcommon-dev wayland-protocols extra-cmake-modules \
  libgl1-mesa-dev libdbus-1-dev libcurl4-openssl-dev
```

Package	Purpose
`build-essential`	C/C++ compiler and toolchain
`cmake`, `ninja-build`	Build system
`pkg-config`	Locating the D-Bus and Wayland modules at configure time
`python3`	Generated editor API suggestion files
`libglfw3-dev`	Windowing / input
`libxi-dev`, `libxcursor-dev`, `libxrandr-dev`, `libxinerama-dev`	X11 input, cursor, and display support
`libwayland-dev`, `libxkbcommon-dev`, `wayland-protocols`	Wayland support through GLFW/native dialogs
`libgl1-mesa-dev`	OpenGL headers
`libdbus-1-dev`	Native file dialog / portal support
`libcurl4-openssl-dev`	HTTP client for the editor's AI assistant (Windows uses native WinHTTP instead)

2. Clone the repository

```
git clone https://github.com/doriaxengine/doriax.git
cd doriax
```

3. Configure and build

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -G "Ninja"
cmake --build build --config Release --target doriax-editor
```

Ninja is a **single-config** generator, so the `doriax-editor` executable is created directly under `build/`.

The editor itself renders with OpenGL Core on Linux. To build it against Vulkan instead, install the Vulkan SDK packages (see Vulkan backend) and configure with `-DGRAPHIC_BACKEND=vulkan`:

```
cmake -S . -B build-vulkan -DGRAPHIC_BACKEND=vulkan -DCMAKE_BUILD_TYPE=Release -G "Ninja"
```

`GRAPHIC_BACKEND` is cached for the editor build, so the build directory keeps the backend it was created with — later `cmake` runs without the flag will not switch it back. Use a separate build directory (or pass `-DGRAPHIC_BACKEND=` again) to change it.

Runtime project build

Linux runtime builds use the native X11 app backend and default to OpenGL Core.

```
cmake -S engine -B build-runtime \
  -DPROJECT_ROOT=/path/to/project \
  -DCMAKE_BUILD_TYPE=Release \
  -G "Ninja"
cmake --build build-runtime --config Release --target doriax-project
```

Assets and Lua files are copied next to the build output when the project contains `assets/` and `lua/` folders.

In a Source Code export, `python3 doriax.py` runs both commands. See Build Script (p. 748).

Vulkan backend

To build a Linux runtime with the Vulkan backend, install the Vulkan SDK development packages and pass `-DGRAPHIC_BACKEND=vulkan` at configure time:

```
sudo apt install -y libvulkan-dev vulkan-tools
cmake -S engine -B build-vulkan \
  -DPROJECT_ROOT=/path/to/project \
  -DGRAPHIC_BACKEND=vulkan \
  -DCMAKE_BUILD_TYPE=Release \
  -G "Ninja"
cmake --build build-vulkan --config Release --target doriax-project
```

Shaders for exported Vulkan projects are compiled to SPIR-V by the editor's shader builder.

Building for macOS

1. Install dependencies

- **Xcode** with the Command Line Tools (`xcode-select --install`)
- **CMake** — install via the official installer[30] or Homebrew:
- Python 3 for generated editor API suggestion files

```
brew install cmake ninja
```

2. Clone the repository

```
git clone https://github.com/doriaxengine/doriax.git
cd doriax
```

3. Configure and build

Using Ninja (single-config):

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -G "Ninja"
cmake --build build --config Release --target doriax-editor
```

The build produces `build/Doriax.app`. The executable inside it is `Doriax.app/Contents/MacOS/Doriax`, the engine runtime sits beside it as `libdoriax.dylib`, and the engine SDK is copied to `Contents/Resources/engine-sdk`.

The editor targets macOS 12.0. To change that, pass `-DCMAKE_OSX_DEPLOYMENT_TARGET=<version>` when configuring.

> **Why the SDK is in `Contents/Resources`**
>
> `codesign` treats everything under `Contents/MacOS` as code and refuses to sign a bundle that has plain files there. The editor resolves the SDK through `FileUtils::getEngineDir()`, which checks `Contents/Resources/engine-sdk` first and falls back to `engine-sdk/` next to the executable, as on the other platforms.

> **Using the Xcode generator**
>
> To work inside Xcode, generate an Xcode project instead:
>
> ```
> cmake -S . -B build -G "Xcode"
> cmake --build build --config Release --target doriax-editor
> ```
>
> Xcode is a **multi-config** generator, so the bundle is placed under a configuration subdirectory such as `build/Release/Doriax.app`.

Runtime project build

macOS runtime builds default to Metal. The Xcode generator with Metal builds an `.app` bundle on the native Apple backend (`platform/apple`). Every other generator or backend builds a plain executable on the native Cocoa backend (`platform/mac`).

30. https://cmake.org/download/

```
cmake -S engine -B build-runtime \
  -DPROJECT_ROOT=/path/to/project \
  -DCMAKE_BUILD_TYPE=Release \
  -G "Ninja"
cmake --build build-runtime --config Release --target doriax-project
```

For Xcode:

```
cmake -S engine -B build-xcode -DPROJECT_ROOT=/path/to/project -G "Xcode"
cmake --build build-xcode --config Release --target doriax-project
```

The Xcode app bundle gets the project's `assets/` and `lua/` folders in `Contents/Resources`, copied after each build. Plain executables get them copied next to the executable instead. See Build Script (p. 748) for building a Source Code export.

The engine sets the macOS deployment target to 12.0 for runtime builds.

> **VSync**
> The project **VSync** setting applies to editor Play mode and supported desktop backends, but macOS Metal runtime builds currently remain synchronized. Configuring an exported Metal project with VSync disabled prints a CMake warning instead of selecting an unsafe zero swap interval.

Vulkan backend

macOS has no native Vulkan driver. Vulkan runs through **MoltenVK**, a translation layer over Metal that ships with the Vulkan SDK[31]. Passing `-DGRAPHIC_BACKEND=vulkan` builds the runtime against it, presenting to a `CAMetalLayer` through `VK_EXT_metal_surface`:

```
cmake -S engine -B build-vulkan \
  -DPROJECT_ROOT=/path/to/project \
  -DGRAPHIC_BACKEND=vulkan \
  -DCMAKE_BUILD_TYPE=Release \
  -G "Ninja"
cmake --build build-vulkan --config Release --target doriax-project
```

If `VULKAN_SDK` is not set, the build looks for the SDK where its installer unpacks it (`~/VulkanSDK/<version>/macOS`), so an editor launched from the Dock — which inherits no shell environment — still finds it.

> **Not runnable yet**
> The renderer binds its resources through `VK_EXT_descriptor_buffer`, and MoltenVK does not implement that extension. A macOS Vulkan build compiles and links, then exits at startup with a message saying so, and the configure step prints a warning up front. **Metal is the supported macOS backend**; Vulkan is here for the day a driver provides the extension. This is also why Vulkan is not offered in the editor's Desktop export.

Shaders for Vulkan projects are compiled to SPIR-V by the editor's shader builder.

31. https://vulkan.lunarg.com/

Building for Android

Doriax can target **Android** for your exported game projects. The runtime supports Android with an OpenGL ES 3.1 backend through the Android Native Activity path.

Requirements

- Android Studio[32] with the Android SDK and NDK
- Java 17 or newer for Gradle builds
- A configured Android device or emulator for testing

Workflow

1. Install Android Studio and ensure the **SDK** and **NDK** are installed via the SDK Manager.
2. Make sure the `ANDROID_HOME` (SDK) and NDK paths are available to your environment.
3. Fill in **Project Settings → Platforms → Android** — package name, version, SDK levels, orientation, architectures, permissions, launcher icons, and the AdMob and Play Billing services. See Android settings (p. 62).
4. Optional: enable **Project Settings → Directories → Native Resource Pack** to ship assets and Lua files in one `resources.pak` instead of as loose files.
5. Export your project from the Doriax editor as **Source Code**, with the Android preset selected in the Export Window's graphic backends.
6. Open the generated Android workspace in Android Studio, then build and run on a device or emulator.

Project configuration

A Source Code export writes the Android settings into the generated workspace, so the manifest and Gradle files do not have to be edited by hand:

Setting	Written to
Package name, version code / name, min and target SDK, architectures, the AdMob and Play Billing libraries	`app/build.gradle`
Permissions, orientation, allow backup, AdMob App ID	`app/src/main/AndroidManifest.xml`
Application name	`app/src/main/res/values/strings.xml`
Fullscreen theme	`app/src/main/res/values/styles.xml` and `MainActivity.java`
Launcher icon, or the adaptive foreground/background pair	`app/src/main/res/drawable` and `mipmap-anydpi-v26`

An export with no architecture selected fails rather than producing an APK that cannot run, so keep at least one ABI ticked.

Ads and in-app purchases

The `AdMob` (p. 610) and `InAppPurchase` (p. 618) classes run on Google's Android libraries, which the workspace template compiles in with their Java wrappers:

Service	Libraries	Wrapper sources
Google AdMob	`play-services-ads` 25.5.0, `user-messaging-platform` 4.0.0	`engine/platform/android/java-admob`

32. https://developer.android.com/studio

Service	Libraries	Wrapper sources
Google Play Billing	`billingclient:billing` 9.1.0	`engine/platform/android/java-billing`

A Source Code export removes the libraries and the wrapper of each service left off in the Android settings, so a game without ads ships without Google's ad SDK. For each service left on, it raises `minSdkVersion` to what the library needs (24 for AdMob, 23 for Play Billing) and `compileSdk` to at least 35. With AdMob on, it also writes the **AdMob App ID** into the manifest. See Monetization (p. 337) for using them.

Resource packaging

The native resource pack is experimental and disabled by default. When enabled, a Source Code export writes `project/assets/resources.pak`, removes the loose asset and Lua contents, and the Android build ships the pack at the APK asset root. Normal engine loaders, Lua modules, and `Data` continue to resolve asset-relative, `asset://`, and `lua://` paths.

A direct `File` handle cannot open packed entries, the full pack must remain below 2 GiB, and its bytes are obfuscated rather than encrypted. See Export Window — Native resource pack (p. 160) for the complete restrictions.

Command-line build

The engine repository contains an Android Studio workspace template under `engine/workspaces/androidstudio/`. Exported projects use the same runtime path. The template builds both optional services in, so its `minSdkVersion` is 24.

```
cd engine/workspaces/androidstudio
./gradlew assembleDebug
```

Android builds link the Android log, EGL/OpenGL ES, OpenSL ES, Game Activity, and Game Frame Pacing libraries.

> **Tooling is being refreshed**
>
> Android project export and the associated workspace tooling are being updated under the Doriax name. Some paths and steps may still reference the legacy Supernova layout while the transition completes. Check the repository[33] for the latest specifics.

33. https://github.com/doriaxengine/doriax

Building for iOS

Doriax can target **iOS** for your exported game projects, using the Metal graphics backend and the native Apple app backend.

Requirements

- A **Mac** with **Xcode** installed
- An Apple Developer account for deploying to physical devices
- CMake (for building the engine from source)

Workflow

1. Install Xcode and its Command Line Tools (`xcode-select --install`).
2. Fill in **Project Settings → Platforms → iOS** — bundle identifier, version name and build number, icon, the status bar / home indicator / high refresh rate options, and the Google AdMob and App Store Purchases services. See iOS settings (p. 62).
3. Export the project as **Source Code** with the iOS backend preset selected. The exporter writes those settings into the generated `Info.plist` and `AppIcon` asset set, so the workspace opens already configured.
4. Open `workspaces/xcode/Doriax.xcodeproj` in the export folder.
5. Select a simulator or a connected device, then build and run from Xcode.

The exported workspace

The workspace builds the game without CMake. It compiles the generated scene and bundle sources and every C++ source the export copied, in groups that mirror `project/`, and searches `project/`, `project/scripts/` and each script directory (p. 211) for headers. Headers are not added to the project navigator: Xcode maps every listed header by name for all targets, so a project `Log.h` would replace the engine's.

The export also sets the project's C++ Standard (p. 59) and the engine capacity macros (p. 359). `ProjectBuild.cmake` (p. 212) does not apply. The **Doriax macOS** scheme of the same workspace builds the game for macOS.

Command-line build

A Source Code export has the workspace under `workspaces/xcode/`, and the engine repository has its template under `engine/workspaces/xcode/`. For simulator testing, use Xcode or `xcodebuild` from that folder:

```
cd workspaces/xcode
xcodebuild build -sdk iphonesimulator -project Doriax.xcodeproj \
    -configuration Debug -scheme "Doriax iOS"
```

iOS runtime builds use Metal and set the deployment target to iOS 15.0 in the engine configuration. CMake builds copy the project's `assets/` and `lua/` folders into the app bundle after each build, since iOS reads them from inside the app.

Google Mobile Ads

The `AdMob` (p. 610) class runs on Google Mobile Ads SDK 13.11.0 and the User Messaging Platform, vendored as static xcframeworks under `engine/platform/apple/GoogleMobileAdsSdkiOS-13.11.0`. The Xcode template links them and compiles the AdMob adapter, `AdMobAdapter.m`, behind the `DORIAX_ADMOB` define.

A Source Code export with **Google AdMob** off in the iOS settings unlinks both frameworks, drops the define, and removes the `GADApplicationIdentifier` and `NSUserTrackingUsageDescription` keys from `Info.plist`. With it on, the export writes the **AdMob App ID** and the **Tracking Description** into those keys.

CMake builds follow the `DORIAX_ADMOB` (p. 358) option, on by default. With CMake 3.28 or newer the xcframeworks are linked whole, and Xcode picks the device or simulator slice; older CMake links the simulator slice only. The SDK also needs the Swift runtime libraries, which a placeholder Swift source in that folder makes Xcode link.

App Store purchases

The `InAppPurchase` (p. 618) class runs on StoreKit 2 through `StoreKitAdapter.swift`, which implements the Objective-C interface declared in `StoreKitAdapter.h`, the target's bridging header. That Swift feature needs Swift 6.1, so building with App Store purchases takes Xcode 16.3 or later. The engine uses the adapter when built with the `DORIAX_STOREKIT` define.

A Source Code export with **App Store Purchases** off in the iOS settings removes the adapter, the bridging header setting and the define. CMake builds follow the `DORIAX_STOREKIT` (p. 358) option, on by default.

> **Tooling is being refreshed**
>
> iOS project export and the Xcode workspace tooling are being updated under the Doriax name. Some paths may still reference the legacy Supernova layout while the transition completes. Check the repository[34] for current details.

34. https://github.com/doriaxengine/doriax

Building for HTML5

Doriax can export to the **web** (HTML5) using Emscripten[35], producing JavaScript and WebAssembly output that runs in the browser.

> **Let the editor build it**
>
> The Export Window's **Web** mode runs this entire build for you — it locates the Emscripten SDK, compiles the project, and copies the `.html` / `.js` / `.wasm` / `.data` files to your destination folder. The manual steps below are for Source Code exports and custom setups. See Export Window (p. 162).
>
> If the SDK is not on `PATH`, point the editor at your `emsdk` folder once in **Edit → Editor Settings** (p. 70) **→ Web**.

Page customization

The generated page is not fixed. **Project Settings → Platforms → Web** sets the page title, a favicon, extra `<head>` markup, whether the canvas follows the browser window, and whether Emscripten's default logo, status line and console are hidden. For full control, supply a **Custom HTML Shell**: any HTML file containing a `{{DORIAX_DEFAULT_HTML}}` marker, which the build replaces with the page Emscripten generated. See Web settings (p. 60).

Game portals

To publish on a web game portal, pick it under **Project Settings → Platforms → Web → Game Portal**: CrazyGames, Poki, GameDistribution, Yandex Games or YouTube Playables. The export builds that portal's SDK in for the `WebPortal` (p. 626) class, so each portal needs its own export. A build configured by hand takes the same choice as `-DDORIAX_WEB_PORTAL=poki` (p. 358), plus `-DDORIAX_WEB_PORTAL_GAME_ID=<id>` for GameDistribution. See Monetization → Web portals (p. 346) for the API and for testing each portal on `localhost`.

1. Install Emscripten

Download and install the Emscripten SDK by following the official installation instructions[36].

After installation, make the SDK available in your environment:

Linux / macOS

```
source <path-to-emsdk>/emsdk_env.sh
```

Windows

Install CMake[37] and a compatible toolchain, and ensure both `cmake` and the Emscripten tools are on your `PATH`. Test with:

```
cmake --version
emcc --version
```

Emscripten builds on Windows with Ninja, not with Visual Studio. The editor and `doriax.py` take Ninja from `PATH`, or else the one the **C++ CMake tools for Windows** component of Visual Studio installs. Without it, install Ninja[38] and add it to `PATH`.

35. https://emscripten.org/
36. https://emscripten.org/docs/getting_started/downloads.html
37. https://cmake.org/download/
38. https://ninja-build.org

2. Build the web target

Configure and build using the Emscripten toolchain. The output is a set of `.js`, `.wasm`, and `.html` files for your project.

```
emcmake cmake -S engine -B build-web -G "Ninja" \
    -DPROJECT_ROOT=/path/to/project \
    -DCMAKE_BUILD_TYPE=Release
cmake --build build-web --config Release --target doriax-project
```

The runtime selects the `gles3` graphics backend and Emscripten app backend for web builds. Assets and Lua files are preloaded into the virtual filesystem when the project contains `assets/` and `lua/` folders. The experimental Native Resource Pack setting does not apply to Web mode — Emscripten still emits its `.data` bundle from the loose folders. Leave packing off if you export Source Code and then run this Emscripten build yourself.

In a Source Code export, `python3 doriax.py -p web` finds the SDK and runs both commands. See Build Script — Web builds (p. 749).

Thread support

Thread support is disabled by default for Emscripten builds. Enable it only when your hosting environment supports the required cross-origin isolation headers:

```
emcmake cmake -S engine -B build-web -G "Ninja" \
    -DPROJECT_ROOT=/path/to/project \
    -DEMSCRIPTEN_THREAD_SUPPORT=ON
```

Web builds use memory growth and start with a 256 MB initial memory configuration.

3. Run it

Opening the generated `.html` file directly from the filesystem can trigger browser security errors. Serve it over HTTP instead:

```
python3 -m http.server
```

Then open http://127.0.0.1:8000 in your browser.

> **Tooling is being refreshed**
>
> The web export pipeline is being updated under the Doriax name. Some steps may still reference the legacy Supernova layout while the transition completes. Check the repository[39] for the latest workflow.

39. https://github.com/doriaxengine/doriax

PART 7

Contributing

Documentation Guide 764

Documentation Guide

The documentation site is built with MkDocs and a custom Doriax theme. Source pages live under `docs/`; generated site output lives under `site/`.

Local preview

```
pip install -r requirements.txt
mkdocs serve
mkdocs build --strict
```

Reference documentation

API reference pages live under `docs/reference/classes/`. Each file documents **one class** with C++ and Lua on the same page (Supernova-style[40]).

Do not use a bulk markdown generator in this repo. Pages are edited by hand.

Inventory tool

To list every bound class, method, and property, run the engine script (reference only):

```
python3 /path/to/doriax/generate_api_suggestions.py \
  engine/core/script/binding /tmp/out.h engine/core
```

Use that output when adding or verifying APIs. Then edit the matching `docs/reference/classes/<name>.md` file and add descriptions for important methods.

Priority for descriptions

Expand these first when touching the reference: `Engine`, `Scene`, `Object`, `Input`, `Camera`, `Body2D`, `Body3D`, `PhysicsSystem`, `Mesh`, `Sprite`, `Action`, `Animation`.

Page structure

1. Short class description
2. Properties / Methods / Events tables with a **Languages** column (`C++ | Lua`)
3. **Method details** for important APIs (tabbed C++ / Lua examples)
4. Enumerations when relevant

File organization

Folder	Purpose
`getting-started/`	Orientation and first-run
`tutorials/`	Step-by-step workflows
`editor/`	Editor panels and export
`manual/`	Concepts (ECS, scripting, events)
`reference/classes/`	Per-class API (hand-maintained)
`building/`	Platform build setup

40. https://github.com/supernovaengine/supernova-docs

Screenshots

Screenshots live in `docs/assets/screenshots/` as PNG files.

- Capture the editor at **1920×1080** in its default layout (**View → Reset Layout**), so every page shows the same editor.
- Use the whole window when a page walks through a workflow. When it documents one panel, menu, or dialog, crop to it at its native size instead of scaling.
- Keep local paths and personal data out of the Output panel and dialogs.
- Name the file after what it shows (`properties-script.png`, `export-desktop.png`) and say the same in the alt text.
- Delete a screenshot once no page references it.

Links

Use relative links. Run `mkdocs build --strict` after nav or path changes.

Versioned documentation

The published site carries one build per release series plus the in-development docs from `main`:

URL	Contents
`docs.doriax.org/`	Latest stable release (canonical URLs)
`docs.doriax.org/0.7/`	The 0.7 release series
`docs.doriax.org/unstable/`	Current `main` branch

`main` is always the unstable docs. Changes land there and reach the stable URLs only when a release is tagged, so a page describing an unreleased feature does not have to be held back.

Each series is built from its highest patch tag (`v0.7.0`, `v0.7.1` → `0.7`, built from `v0.7.1`). Pre-release tags are ignored.

`scripts/build-versions.sh` assembles the whole tree: it builds every series, mirrors the newest one at the root, and writes `versions.json`, which drives the version picker in the header. Old builds read that file at runtime, so they list newer releases without being rebuilt.

`mkdocs serve` builds only the unstable version, and the picker stays hidden because there is no `versions.json` to read. To preview the full versioned site:

```
scripts/build-versions.sh site
python3 -m http.server -d site
```

Releasing a version

Tag the documentation repository when the matching engine release goes out:

```
git tag v0.7.1
git push origin v0.7.1
```

The tag push rebuilds and redeploys every version. Only the tagged commit's own `docs/` is published for that version, so the tag must point at the commit whose docs describe the release.

Correcting published docs means moving the tag, since a tag is the only thing the build reads:

```
git commit -am "Fix physics layer description"   # on main
git tag -f v0.7.1 && git push -f origin v0.7.1
```

When `main` already holds docs for unreleased features, moving the tag there would publish them too. Apply just the fix on a branch from the tag instead:

```
git switch -c release/0.8.4-fix v0.8.4
git cherry-pick <fix>
git tag -f v0.8.4
git push origin release/0.8.4-fix && git push -f origin v0.8.4
```

A tag push runs the workflow stored in the tagged commit. If that one predates the PDFs, the deployment drops them until the next push to `main`.

The `github-pages` environment must allow tags to deploy, or the tag run builds the site and then fails at the deploy step: under **Settings → Environments → github-pages → Deployment branches and tags**, `v*` needs a tag rule alongside the `main` branch rule.

Printable PDF

Every published version carries a PDF of itself, linked as **Download PDF** at the bottom of the sidebar:

URL	Contents
`docs.doriax.org/doriax-docs.pdf`	Latest release, at a link that never changes
`docs.doriax.org/0.7/doriax-docs.pdf`	The 0.7 release series
`docs.doriax.org/unstable/doriax-docs.pdf`	Current `main` branch

The deployment builds them with `scripts/build-pdf.py`, which turns one tag or branch into a single A4 PDF, with a table of contents, page numbers on cross-references, and every language tab printed in full. It renders that ref's own `docs/` with MkDocs, so uncommitted changes do not reach it, and lays the pages out with WeasyPrint. To build one yourself:

```
pip install -r requirements-pdf.txt
scripts/build-pdf.py --ref v0.8.4 --strict
```

WeasyPrint also needs Pango and the Noto and DejaVu fonts, listed in `.github/workflows/docs.yml`. `DOCS_PDF=1 scripts/build-versions.sh site` previews the whole site with its PDFs.

A link that does not resolve, a missing image, or content running off the page is an error with `--strict`. In the deployment it is a warning on the workflow run, so that it never holds back the site. The Reference part follows the API Index (p. 352) rather than the sidebar, so list every new class page there.

Layout and theme changes

Only `docs/` and `mkdocs.yml` come from a release tag. Every version is built with the theme from `main`, so a fix to the header, CSS, or search reaches all published versions on the next build, and a version tagged before a theme feature existed still gets it.

The trade-off is that a theme change has to keep rendering older content. Preview one against the real thing before pushing:

```
scripts/build-versions.sh site        # every version, current theme
DOCS_SHARED_THEME=0 scripts/build-versions.sh site   # each version's own theme
```

`versions.json` is a compatibility contract in the other direction: already published builds parse it with the JavaScript they shipped with. Add fields to it, never rename or remove them.

PART 8

About

FAQ 768
License & Credits 771

Frequently Asked Questions

What is Doriax Engine?

Doriax is a free, open-source 2D/3D game engine with an integrated desktop editor. You author scenes visually, script in **Lua** or **C++**, and export buildable native projects for desktop, mobile, and web. It is the continuation of **Supernova Engine** under a new identity, keeping its ECS-based, data-oriented runtime.

How much does it cost? Can I sell my game?

Doriax is released under the **MIT License** — free for personal and commercial use, with no royalties, revenue share, or seat licenses. Bundled third-party libraries keep their own (compatible) open-source licenses. See License & Credits (p. 771).

Which platforms are supported?

Area	Platforms
Editor	Windows, Linux, macOS
Exported projects	Windows, Linux, macOS, Android, iOS, HTML5 (WebAssembly)
Graphics backends	OpenGL, OpenGL ES 3, Vulkan, Metal, Direct3D 11, WebGPU

Why can't I choose Vulkan when exporting on macOS?

macOS has no native Vulkan driver. Vulkan runs there through MoltenVK, a translation layer on top of Metal, and MoltenVK does not yet implement `VK_EXT_descriptor_buffer` — the feature the renderer uses to bind textures, buffers and samplers. A macOS Vulkan build compiles and links, but exits at startup on every Mac tested so far.

Rather than offer a build that cannot run, the export dialog lists **Metal** (the default and recommended choice) and **OpenGL**. The Vulkan backend itself is in the engine and can be built from the command line; see Vulkan backend (p. 755). Windows and Linux are unaffected — Vulkan is available on both.

Lua or C++ — which should I use?

Both target the same runtime API. **Lua** gives the fastest iteration (no compile step) and is the best starting point; **C++** gives native performance for hot paths. Most projects mix them — see Scripting (p. 195).

Does exporting produce a finished executable?

Exporting produces a **self-contained, buildable native project** (CMake, Gradle, or Xcode depending on the target) with your scenes, assets, scripts, and compiled shaders. You then build it with the platform's standard toolchain — see Exporting a Project (p. 40) and Command-Line Tools (p. 167) for automating this in CI.

Is Doriax production-ready?

Doriax is under active development. The downloadable editor builds track the `main` branch and are **not stable releases** — expect breaking changes while the engine and documentation are refreshed under the Doriax name. Pin a commit for serious projects and follow the repository for release tags.

How does Doriax differ from Godot, Unity, or Unreal?

Doriax aims to be **lightweight and code-transparent**: a small MIT-licensed C++ runtime with an ECS core, where exporting gives you a native project you fully own and can read, modify, and build yourself. It does not try to match the breadth of larger engines' tooling; it focuses on a compact feature set — 2D, 3D, UI, physics (Box2D / Jolt), audio (SoLoud), particles, terrain, and skeletal animation — with both Lua and C++ as first-class languages.

Does Doriax use engine-specific asset formats?

No. Doriax has no equivalent of Godot's `.scn`/`.res` or Unity's imported assets, and no import step that rewrites your files. The runtime opens standard formats directly — `.glb`/`.gltf`/`.obj` for models, PNG/JPG/TGA/BMP/HDR/PSD/SVG for textures, OGG/WAV/MP3/FLAC for audio, and TTF/OTF/TTC for fonts. The file your content tool produces is the file the game loads.

The exception is FBX: the editor converts it to a standard binary glTF, which the game loads in its place. The `.fbx` itself is left as it is.

A native export can optionally combine those original bytes into one `resources.pak`. That is packaging and light obfuscation, not a converted engine format — see Native resource pack (p. 160).

Scenes, bundles, and materials are the exception. They are authored as YAML for the editor and converted at export into C++ that is compiled into your game, rather than shipped as data the runtime parses. See Resources & Assets (p. 238).

Can I load assets at runtime?

Yes, for media. Models, textures, sounds, and fonts can be loaded or swapped from any path at any point during gameplay, with `Engine.asyncLoading = true` moving the work to worker threads and `ResourceProgress` (p. 717) reporting how far along it is. Lua scripts also load at runtime through the `lua://` root.

Those same paths work in a packed native export through engine loaders and `Data`. A direct `File` handle cannot open a packed entry.

Entity hierarchies work differently, because scenes and bundles are compiled rather than parsed. Spawning them on demand is fully supported through `BundleManager.createBundle` (p. 598) for any bundle that existed at build time, but a scene or hierarchy contained in a file the executable was not built with cannot be loaded. That is a deliberate trade: it keeps entity setup statically typed, leaves reflection and deserialization out of the runtime, and keeps startup cost and binary size low on the web and mobile targets. If you need player-supplied or downloaded content, drive it with runtime media loading plus bundles composed in script.

Where do my old Supernova projects stand?

Supernova `0.5.5` was the last release of the legacy engine. The runtime API carried over largely intact (with renames such as `SPROPERTY` → `DPROPERTY` and the `doriax` namespace), but projects should migrate to the Doriax editor workflow. Some internal folders may still mention the previous name during the transition.

Where can I find the editor crash log?

The desktop editor writes timestamped Output-panel messages to `editor.log`. If the editor exits because of a fatal signal or an unhandled exception, it also appends the editor version, executable path, reason, and a backtrace. Include this file when reporting a crash.

The log is beside the active `settings.yaml`. An editor started from an installed desktop launcher normally uses:

Platform	Log path
Windows	`%APPDATA%\Doriax\editor.log`
macOS	`~/Library/Application Support/Doriax/editor.log`
Linux	`$XDG_CONFIG_HOME/doriax/editor.log`, or `~/.config/doriax/editor.log` when `XDG_CONFIG_HOME` is unset

When you run the editor from source or another writable working directory, it may keep both `settings.yaml` and `editor.log` in that directory instead. The definitive rule is that the two files are together.

The editor appends several sessions to the log. At startup, a log larger than 2 MiB is moved to `editor.log.1` before a new `editor.log` is created. If both files exist after a crash, include both so the earlier session is not lost.

Where can I get help or report bugs?

- **Discord** — discord.gg/yXXDyJf3gT[41] for questions and discussion.
- **GitHub** — github.com/doriaxengine/doriax[42] for issues and contributions. For editor crashes, attach the `editor.log` described above.

How can I contribute?

Code contributions go through pull requests on GitHub. Documentation lives in its own repository and accepts fixes through the **Edit this page** links — see the Documentation Guide (p. 764).

41. https://discord.gg/yXXDyJf3gT
42. https://github.com/doriaxengine/doriax

License & Credits

License

Doriax Engine is released under the **MIT License** and is free for both personal and commercial use — no royalties, no strings attached.

Third-party libraries bundled in the repository keep their own respective licenses.

Credits

Doriax Engine is created and maintained by **Eduardo Doria**. It is the continuation of **Supernova Engine** under a new identity, extending the original ECS, data-oriented runtime with a full editor and export pipeline.

Built with

Doriax integrates several open-source libraries, including:

Library	Use
Box2D[43]	2D physics
Jolt Physics[44]	3D physics
Lua[45]	Scripting
meshoptimizer[46]	Mesh detail-level (LOD) simplification

Community & support

- Website[47]
- GitHub[48]
- Discord[49]
- Sponsor on GitHub[50]

Developers who adopt the engine receive full support, with improvements delivered as quickly as possible.

Documentation license

This documentation is licensed under CC BY 4.0[51].

43. https://box2d.org/
44. https://github.com/jrouwe/JoltPhysics
45. https://www.lua.org/
46. https://github.com/zeux/meshoptimizer
47. https://doriax.org
48. https://github.com/doriaxengine/doriax
49. https://discord.gg/yXXDyJf3gT
50. https://github.com/sponsors/eduardodoria
51. https://creativecommons.org/licenses/by/4.0/