
Godot Engine Documentation

Release latest

Juan Linietsky, Ariel Manzur and the Godot community

Dec 08, 2023

ABOUT

Note: Godot's documentation is available in various languages and versions. Expand the "Read the Docs" panel at the bottom of the sidebar to see the list.

Welcome to the official documentation of [Godot Engine](#), the free and open source community-driven 2D and 3D game engine! If you are new to this documentation, we recommend that you read the [introduction page](#) to get an overview of what this documentation has to offer.

The table of contents in the sidebar should let you easily access the documentation for your topic of interest. You can also use the search function in the top-left corner.

GET INVOLVED

Godot Engine is an open source project developed by a community of volunteers. The documentation team can always use your feedback and help to improve the tutorials and class reference. If you don't understand something, or cannot find what you are looking for in the docs, help us make the documentation better by letting us know!

Submit an issue or pull request on the [GitHub repository](#), help us [translate the documentation](#) into your language, or talk to us on the [#documentation](#) channel on the [Godot Contributors Chat](#)!



OFFLINE DOCUMENTATION

To browse the documentation offline, you can use the mirror of the documentation hosted on [DevDocs](#). To enable offline browsing on DevDocs, you need to:

- Click the three dots in the top-left corner, choose Preferences.
- Enable the desired version of the Godot documentation by checking the box next to it in the sidebar.
- Click the three dots in the top-left corner, choose Offline data.
- Click the Install link next to the Godot documentation.

You can also [download an HTML copy](#) for offline reading (updated every Monday). Extract the ZIP archive then open the top-level index.html in a web browser.

For mobile devices or e-readers, you can also [download an ePub copy](#) for offline reading (updated every Monday). Extract the ZIP archive then open the GodotEngine.epub file in an e-book reader application.

2.1 Introduction

```
func _ready():  
    print("Hello world!")
```

Welcome to the official documentation of Godot Engine, the free and open source community-driven 2D and 3D game engine! Behind this mouthful, you will find a powerful yet user-friendly tool that you can use to develop any kind of game, for any platform and with no usage restriction whatsoever.

This page gives a broad overview of the engine and of this documentation, so that you know where to start if you are a beginner or where to look if you need information on a specific feature.

2.1.1 Before you start

The Tutorials and resources page lists video tutorials contributed by the community. If you prefer video to text, consider checking them out. Otherwise, [Getting Started](#) is a great starting point.

In case you have trouble with one of the tutorials or your project, you can find help on the various Community channels, especially the Godot [Discord](#) community and [Q&A](#).

2.1.2 About Godot Engine

A game engine is a complex tool and difficult to present in a few words. Here's a quick synopsis, which you are free to reuse if you need a quick write-up about Godot Engine:

Godot Engine is a feature-packed, cross-platform game engine to create 2D and 3D games from a unified interface. It provides a comprehensive set of common tools, so that users can focus on making games without having to reinvent the wheel. Games can be exported with one click to a number of platforms, including the major desktop platforms (Linux, macOS, Windows), mobile platforms (Android, iOS), as well as Web-based platforms and consoles.

Godot is completely free and open source under the [permissive MIT license](#). No strings attached, no royalties, nothing. Users' games are theirs, down to the last line of engine code. Godot's development is fully independent and community-driven, empowering users to help shape their engine to match their expectations. It is supported by the [Godot Foundation](#) not-for-profit.

2.1.3 Organization of the documentation

This documentation is organized into several sections:

- About contains this introduction as well as information about the engine, its history, its licensing, authors, etc. It also contains the [Frequently asked questions](#).
- Getting Started contains all necessary information on using the engine to make games. It starts with the Step by step tutorial which should be the entry point for all new users. This is the best place to start if you're new!
- The Manual can be read or referenced as needed, in any order. It contains feature-specific tutorials and documentation.
- Contributing gives information related to contributing to Godot, whether to the core engine, documentation, demos or other parts. It describes how to report bugs, how contributor workflows are organized, etc. It also contains sections intended for advanced users and contributors, with information on compiling the engine, contributing to the editor, or developing C++ modules.
- Community is dedicated to the life of Godot's community. It points to various community channels like the [Godot Contributors Chat](#) and [Discord](#) and contains a list of recommended third-party tutorials and materials outside of this documentation.
- Finally, the Class reference documents the full Godot API, also available directly within the engine's script editor. You can find information on all classes, functions, signals and so on here.

In addition to this documentation, you may also want to take a look at the various [Godot demo projects](#).

2.1.4 About this documentation

Members of the Godot Engine community continuously write, correct, edit, and improve this documentation. We are always looking for more help. You can also contribute by opening Github issues or translating the documentation into your language. If you are interested in helping, see [Ways to contribute](#) and [Writing documentation](#), or get in touch with the [Documentation team](#) on [Godot Contributors Chat](#).

All documentation content is licensed under the permissive Creative Commons Attribution 3.0 (CC BY 3.0) license, with attribution to "Juan Linietsky, Ariel Manzur, and the Godot Engine community" unless otherwise noted.

Have fun reading and making games with Godot Engine!

2.2 List of features

This page aims to list all features currently supported by Godot.

Note: This page lists features supported by the current stable version of Godot. Some of these features may not be available in the [LTS release series \(3.x\)](#).

2.2.1 Platforms

Can run both the editor and exported projects:

- Windows 7 and later (64-bit and 32-bit).
- macOS 10.12 and later (64-bit, x86 and ARM).
- Linux (64-bit, x86 and ARM).
 - Binaries are statically linked and can run on any distribution if compiled on an old enough base distribution.
 - Official binaries are compiled on Ubuntu 14.04.
 - 32-bit binaries can be compiled from source.
- Android 6.0 and later (editor support is experimental).
- Web browsers. Experimental in 4.0, using Godot 3.x is recommended instead when targeting HTML5.

Runs exported projects:

- iOS 11.0 and later.
- Consoles.

Godot aims to be as platform-independent as possible and can be ported to new platforms with relative ease.

Note: Projects written in C# using Godot 4 currently cannot be exported to the web platform. To use C# on that platform, consider Godot 3 instead. Android and iOS platform support is available as of Godot 4.2, but is experimental and some limitations apply.

2.2.2 Editor

Features:

- Scene tree editor.
- Built-in script editor.
- Support for external script editors such as Visual Studio Code or Vim.
- GDScript debugger.
 - No support for debugging in threads yet.
- Visual profiler with CPU and GPU time indications for each step of the rendering pipeline.
- Performance monitoring tools, including custom performance monitors.
- Live script reloading.
- Live scene editing.

- Changes will reflect in the editor and will be kept after closing the running project.
- Remote inspector.
 - Changes won't reflect in the editor and won't be kept after closing the running project.
- Live camera replication.
 - Move the in-editor camera and see the result in the running project.
- Built-in offline class reference documentation.
- Use the editor in dozens of languages contributed by the community.

Plugins:

- Editor plugins can be downloaded from the asset library to extend editor functionality.
- Create your own plugins using GDScript to add new features or speed up your workflow.
- Download projects from the asset library in the Project Manager and import them directly.

2.2.3 Rendering

3 rendering methods (running over 2 rendering drivers) are available:

- Forward+, running over Vulkan 1.0 (with optional Vulkan 1.1 and 1.2 features). The most advanced graphics backend, suited for desktop platforms only. Used by default on desktop platforms.
- Forward Mobile, running over Vulkan 1.0 (with optional Vulkan 1.1 and 1.2 features). Less features, but renders simple scenes faster. Suited for mobile and desktop platforms. Used by default on mobile platforms.
- Compatibility, running over OpenGL 3.3 / OpenGL ES 3.0 / WebGL 2.0. The least advanced graphics backend, suited for low-end desktop and mobile platforms. Used by default on the web platform.

2.2.4 2D graphics

- Sprite, polygon and line rendering.
 - High-level tools to draw lines and polygons such as Polygon2D and Line2D, with support for texturing.
- AnimatedSprite2D as a helper for creating animated sprites.
- Parallax layers.
 - Pseudo-3D support including preview in the editor.
- 2D lighting with normal maps and specular maps.
 - Point (omni/spot) and directional 2D lights.
 - Hard or soft shadows (adjustable on a per-light basis).
 - Custom shaders can access a real-time SDF (Signed Distance Field) representation of the 2D scene based on LightOccluder2D nodes, which can be used for improved 2D lighting effects including 2D global illumination.
- Font rendering using bitmaps, rasterization using FreeType or multi-channel signed distance fields (MSDF).
 - Bitmap fonts can be exported using tools like BMFont, or imported from images (for fixed-width fonts only).

- Dynamic fonts support monochrome fonts as well as colored fonts (e.g. for emoji). Supported formats are TTF, OTF, WOFF1 and WOFF2.
 - Dynamic fonts support optional font outlines with adjustable width and color.
 - Dynamic fonts support variable fonts and OpenType features including ligatures.
 - Dynamic fonts support simulated bold and italic when the font file lacks those styles.
 - Dynamic fonts support oversampling to keep fonts sharp at higher resolutions.
 - Dynamic fonts support subpixel positioning to make fonts crisper at low sizes.
 - Dynamic fonts support LCD subpixel optimizations to make fonts even crisper at low sizes.
 - Signed distance field fonts can be scaled at any resolution without requiring re-rasterization. Multi-channel usage makes SDF fonts scale down to lower sizes better compared to monochrome SDF fonts.
- GPU-based particles with support for custom particle shaders.
 - CPU-based particles.
 - Optional 2D HDR rendering for better glow capabilities.

2.2.5 2D tools

- TileMaps for 2D tile-based level design.
- 2D camera with built-in smoothing and drag margins.
- Path2D node to represent a path in 2D space.
 - Can be drawn in the editor or generated procedurally.
 - PathFollow2D node to make nodes follow a Path2D.
- 2D geometry helper class.

2.2.6 2D physics

Physics bodies:

- Static bodies.
- Animatable bodies (for objects moving only by script or animation, such as doors and platforms).
- Rigid bodies.
- Character bodies.
- Joints.
- Areas to detect bodies entering or leaving it.

Collision detection:

- Built-in shapes: line, box, circle, capsule, world boundary (infinite plane).
- Collision polygons (can be drawn manually or generated from a sprite in the editor).

2.2.7 3D graphics

- HDR rendering with sRGB.
- Perspective, orthographic and frustum-offset cameras.
- When using the Forward+ backend, a depth prepass is used to improve performance in complex scenes by reducing the cost of overdraw.
- Variable rate shading on supported GPUs in Forward+ and Forward Mobile.

Physically-based rendering (built-in material features):

- Follows the Disney PBR model.
- Supports Burley, Lambert, Lambert Wrap (half-Lambert) and Toon diffuse shading modes.
- Supports Schlick-GGX, Toon and Disabled specular shading modes.
- Uses a roughness-metallic workflow with support for ORM textures.
- Uses horizon specular occlusion (Filament model) to improve material appearance.
- Normal mapping.
- Parallax/relief mapping with automatic level of detail based on distance.
- Detail mapping for the albedo and normal maps.
- Sub-surface scattering and transmittance.
- Screen-space refraction with support for material roughness (resulting in blurry refraction).
- Proximity fade (soft particles) and distance fade.
- Distance fade can use alpha blending or dithering to avoid going through the transparent pipeline.
- Dithering can be determined on a per-pixel or per-object basis.

Real-time lighting:

- Directional lights (sun/moon). Up to 4 per scene.
- Omnidirectional lights.
- Spot lights with adjustable cone angle and attenuation.
- Specular, indirect light, and volumetric fog energy can be adjusted on a per-light basis.
- Adjustable light "size" for fake area lights (will also make shadows blurrier).
- Optional distance fade system to fade distant lights and their shadows, improving performance.
- When using the Forward+ backend (default on desktop), lights are rendered with clustered forward optimizations to decrease their individual cost. Clustered rendering also lifts any limits on the number of lights that can be used on a mesh.
- When using the Forward Mobile backend, up to 8 omni lights and 8 spot lights can be displayed per mesh resource. Baked lighting can be used to overcome this limit if needed.

Shadow mapping:

- DirectionalLight: Orthogonal (fastest), PSSM 2-split and 4-split. Supports blending between splits.
- OmniLight: Dual paraboloid (fast) or cubemap (slower but more accurate). Supports colored projector textures in the form of panoramas.
- SpotLight: Single texture. Supports colored projector textures.

- Shadow normal offset bias and shadow pancaking to decrease the amount of visible shadow acne and peter-panning.
- PCSS (Percentage Closer Soft Shadows)-like shadow blur based on the light size and distance from the surface the shadow is cast on.
- Adjustable shadow blur on a per-light basis.

Global illumination with indirect lighting:

- Baked lightmaps (fast, but can't be updated at run-time).
 - Supports baking indirect light only or baking both direct and indirect lighting. The bake mode can be adjusted on a per-light basis to allow for hybrid light baking setups.
 - Supports lighting dynamic objects using automatic and manually placed probes.
 - Optionally supports directional lighting and rough reflections based on spherical harmonics.
 - Lightmaps are baked on the GPU using compute shaders (much faster compared to CPU lightmapping). Baking can only be performed from the editor, not in exported projects.
 - Supports GPU-based denoising with JNLM, or CPU/GPU-based denoising with OIDN.
- Voxel-based GI probes. Supports dynamic lights and dynamic occluders, while also supporting reflections. Requires a fast baking step which can be performed in the editor or at run-time (including from an exported project).
- Signed-distance field GI designed for large open worlds. Supports dynamic lights, but not dynamic occluders. Supports reflections. No baking required.
- Screen-space indirect lighting (SSIL) at half or full resolution. Fully real-time and supports any kind of emissive light source (including decals).
- VoxelGI and SDFGI use a deferred pass to allow for rendering GI at half resolution to improve performance (while still having functional MSAA support).

Reflections:

- Voxel-based reflections (when using GI probes) and SDF-based reflections (when using signed distance field GI). Voxel-based reflections are visible on transparent surfaces, while rough SDF-based reflections are visible on transparent surfaces.
- Fast baked reflections or slow real-time reflections using `ReflectionProbe`. Parallax box correction can optionally be enabled.
- Screen-space reflections with support for material roughness.
- Reflection techniques can be mixed together for greater accuracy or scalability.
- When using the Forward+ backend (default on desktop), reflection probes are rendered with clustered forward optimizations to decrease their individual cost. Clustered rendering also lifts any limits on the number of reflection probes that can be used on a mesh.
- When using the Forward Mobile backend, up to 8 reflection probes can be displayed per mesh resource.

Decals:

- Supports albedo, emissive, ORM (Occlusion Roughness Metallic), and normal mapping.
- Texture channels are smoothly overlaid on top of the underlying material, with support for normal/ORM-only decals.
- Support for normal fade to fade the decal depending on its incidence angle.

- Does not rely on run-time mesh generation. This means decals can be used on complex skinned meshes with no performance penalty, even if the decal moves every frame.
- Support for nearest, bilinear, trilinear or anisotropic texture filtering (configured globally).
- Optional distance fade system to fade distant lights and their shadows, improving performance.
- When using the Forward+ backend (default on desktop), decals are rendered with clustered forward optimizations to decrease their individual cost. Clustered rendering also lifts any limits on the number of decals that can be used on a mesh.
- When using the Forward Mobile backend, up to 8 decals can be displayed per mesh resource.

Sky:

- Panorama sky (using an HDRI).
- Procedural sky and Physically-based sky that respond to the `DirectionalLights` in the scene.
- Support for custom sky shaders, which can be animated.
- The radiance map used for ambient and specular light can be updated in real-time depending on the quality settings chosen.

Fog:

- Exponential depth fog.
- Exponential height fog.
- Support for automatic fog color depending on the sky color (aerial perspective).
- Support for sun scattering in the fog.
- Support for controlling how much fog rendering should affect the sky, with separate controls for traditional and volumetric fog.
- Support for making specific materials ignore fog.

Volumetric fog:

- Global volumetric fog that reacts to lights and shadows.
- Volumetric fog can take indirect light into account when using `VoxelGI` or `SDFGI`.
- Fog volume nodes that can be placed to add fog to specific areas (or remove fog from specific areas). Supported shapes include box, ellipse, cone, cylinder, and 3D texture-based density maps.
- Each fog volume can have its own custom shader.
- Can be used together with traditional fog.

Particles:

- GPU-based particles with support for subemitters (2D + 3D), trails (2D + 3D), attractors (3D only) and collision (2D + 3D).
 - 3D particle attractor shapes supported: box, sphere and 3D vector fields.
 - 3D particle collision shapes supported: box, sphere, baked signed distance field and real-time heightmap (suited for open world weather effects).
 - 2D particle collision is handled using a signed distance field generated in real-time based on `LightOccluder2D` nodes in the scene.
 - Trails can use the built-in ribbon trail and tube trail meshes, or custom meshes with skeletons.
 - Support for custom particle shaders with manual emission.

- CPU-based particles.

Post-processing:

- Tonemapping (Linear, Reinhard, Filmic, ACES).
- Automatic exposure adjustments based on viewport brightness (and manual exposure override).
- Near and far depth of field with adjustable bokeh simulation (box, hexagon, circle).
- Screen-space ambient occlusion (SSAO) at half or full resolution.
- Glow/bloom with optional bicubic upscaling and several blend modes available: Screen, Soft Light, Add, Replace, Mix.
- Glow can have a colored dirt map texture, acting as a lens dirt effect.
- Glow can be used as a screen-space blur effect.
- Color correction using a one-dimensional ramp or a 3D LUT texture.
- Roughness limiter to reduce the impact of specular aliasing.
- Brightness, contrast and saturation adjustments.

Texture filtering:

- Nearest, bilinear, trilinear or anisotropic filtering.
- Filtering options are defined on a per-use basis, not a per-texture basis.

Texture compression:

- Basis Universal (slow, but results in smaller files).
- BPTC for high-quality compression (not supported on macOS).
- ETC2 (not supported on macOS).
- S3TC (not supported on mobile/Web platforms).

Anti-aliasing:

- Temporal antialiasing (TAA).
- AMD FidelityFX Super Resolution 2.2 antialiasing (FSR2), which can be used at native resolution as a form of high-quality temporal antialiasing.
- Multi-sample antialiasing (MSAA), for both 2D antialiasing and 3D antialiasing.
- Fast approximate antialiasing (FXAA).
- Super-sample antialiasing (SSAA) using bilinear 3D scaling and a 3D resolution scale above 1.0.
- Alpha antialiasing, MSAA alpha to coverage and alpha hashing on a per-material basis.

Resolution scaling:

- Support for rendering 3D at a lower resolution while keeping 2D rendering at the original scale. This can be used to improve performance on low-end systems or improve visuals on high-end systems.
- Resolution scaling uses bilinear filtering, AMD FidelityFX Super Resolution 1.0 (FSR1) or AMD FidelityFX Super Resolution 2.2 (FSR2).
- Texture mipmap LOD bias is adjusted automatically to improve quality at lower resolution scales. It can also be modified with a manual offset.

Most effects listed above can be adjusted for better performance or to further improve quality. This can be helpful when using Godot for offline rendering.

2.2.8 3D tools

- Built-in meshes: cube, cylinder/cone, (hemi)sphere, prism, plane, quad, torus, ribbon, tube.
- GridMaps for 3D tile-based level design.
- Constructive solid geometry (intended for prototyping).
- Tools for procedural geometry generation.
- Path3D node to represent a path in 3D space.
 - Can be drawn in the editor or generated procedurally.
 - PathFollow3D node to make nodes follow a Path3D.
- 3D geometry helper class.
- Support for exporting the current scene as a glTF 2.0 file, both from the editor and at run-time from an exported project.

2.2.9 3D physics

Physics bodies:

- Static bodies.
- Animatable bodies (for objects moving only by script or animation, such as doors and platforms).
- Rigid bodies.
- Character bodies.
- Vehicle bodies (intended for arcade physics, not simulation).
- Joints.
- Soft bodies.
- Ragdolls.
- Areas to detect bodies entering or leaving it.

Collision detection:

- Built-in shapes: cuboid, sphere, capsule, cylinder, world boundary (infinite plane).
- Generate triangle collision shapes for any mesh from the editor.
- Generate one or several convex collision shapes for any mesh from the editor.

2.2.10 Shaders

- 2D: Custom vertex, fragment, and light shaders.
- 3D: Custom vertex, fragment, light, and sky shaders.
- Text-based shaders using a shader language inspired by GLSL.
- Visual shader editor.
 - Support for visual shader plugins.

2.2.11 Scripting

General:

- Object-oriented design pattern with scripts extending nodes.
- Signals and groups for communicating between scripts.
- Support for cross-language scripting.
- Many 2D, 3D and 4D linear algebra data types such as vectors and transforms.

GDScript:

- High-level interpreted language with optional static typing.
- Syntax inspired by Python. However, GDScript is not based on Python.
- Syntax highlighting is provided on GitHub.
- Use threads to perform asynchronous actions or make use of multiple processor cores.

C#:

- Packaged in a separate binary to keep file sizes and dependencies down.
- Supports .NET 6 and higher.
 - Full support for the C# 10.0 syntax and features.
- Supports Windows, Linux, and macOS. As of 4.2 experimental support for Android and iOS is also available (requires a .NET 7.0 project for Android and 8.0 for iOS).
 - On the Android platform only some architectures are supported: arm64 and x64.
 - On the iOS platform only some architectures are supported: arm64.
 - The web platform is currently unsupported. To use C# on that platform, consider Godot 3 instead.
- Using an external editor is recommended to benefit from IDE functionality.

GDExtension (C, C++, Rust, D, ...):

- When you need it, link to native libraries for higher performance and third-party integrations.
 - For scripting game logic, GDScript or C# are recommended if their performance is suitable.
- Official GDExtension bindings for C and C++.
 - Use any build system and language features you wish.
- Actively developed GDExtension bindings for D, Haxe, Swift, and Rust bindings provided by the community. (Some of these bindings may be experimental and not production-ready).

2.2.12 Audio

Features:

- Mono, stereo, 5.1 and 7.1 output.
- Non-positional and positional playback in 2D and 3D.
 - Optional Doppler effect in 2D and 3D.
- Support for re-routable audio buses and effects with dozens of effects included.
- Support for polyphony (playing several sounds from a single AudioStreamPlayer node).

- Support for random volume and pitch.
- Support for real-time pitch scaling.
- Support for sequential/random sample selection, including repetition prevention when using random sample selection.
- Listener2D and Listener3D nodes to listen from a position different than the camera.
- Support for procedural audio generation.
- Audio input to record microphones.
- MIDI input.
 - No support for MIDI output yet.

APIs used:

- Windows: WASAPI.
- macOS: CoreAudio.
- Linux: PulseAudio or ALSA.

2.2.13 Import

- Support for custom import plugins.

Formats:

- Images: See Importing images.
- Audio:
 - WAV with optional IMA-ADPCM compression.
 - Ogg Vorbis.
 - MP3.
- 3D scenes: See Importing 3D scenes.
 - glTF 2.0 (recommended).
 - .blend (by calling Blender's glTF export functionality transparently).
 - FBX (by calling [FBX2glTF](#) transparently).
 - Collada (.dae).
 - Wavefront OBJ (static scenes only, can be loaded directly as a mesh or imported as a 3D scene).
- Support for loading glTF 2.0 scenes at run-time, including from an exported project.
- 3D meshes use [Mikktspace](#) to generate tangents on import, which ensures consistency with other 3D applications such as Blender.

2.2.14 Input

- Input mapping system using hardcoded input events or remappable input actions.
 - Axis values can be mapped to two different actions with a configurable deadzone.
 - Use the same code to support both keyboards and gamepads.
- Keyboard input.
 - Keys can be mapped in "physical" mode to be independent of the keyboard layout.
- Mouse input.
 - The mouse cursor can be visible, hidden, captured or confined within the window.
 - When captured, raw input will be used on Windows and Linux to sidestep the OS' mouse acceleration settings.
- Gamepad input (up to 8 simultaneous controllers).
- Pen/tablet input with pressure support.

2.2.15 Navigation

- A* algorithm in 2D and 3D.
- Navigation meshes with dynamic obstacle avoidance in 2D and 3D.
- Generate navigation meshes from the editor or at run-time (including from an exported project).

2.2.16 Networking

- Low-level TCP networking using StreamPeer and TCPServer.
- Low-level UDP networking using PacketPeer and UDPServer.
- Low-level HTTP requests using HTTPClient.
- High-level HTTP requests using HTTPRequest.
 - Supports HTTPS out of the box using bundled certificates.
- High-level multiplayer API using UDP and ENet.
 - Automatic replication using remote procedure calls (RPCs).
 - Supports unreliable, reliable and ordered transfers.
- WebSocket client and server, available on all platforms.
- WebRTC client and server, available on all platforms.
- Support for UPnP to sidestep the requirement to forward ports when hosting a server behind a NAT.

2.2.17 Internationalization

- Full support for Unicode including emoji.
- Store localization strings using CSV or gettext.
 - Support for generating gettext POT and PO files from the editor.
- Use localized strings in your project automatically in GUI elements or by using the `tr()` function.
- Support for pluralization and translation contexts when using gettext translations.
- Support for bidirectional typesetting, text shaping and OpenType localized forms.

- Automatic UI mirroring for right-to-left locales.
- Support for pseudolocalization to test your project for i18n-friendliness.

2.2.18 Windowing and OS integration

- Spawn multiple independent windows within a single process.
- Move, resize, minimize, and maximize windows spawned by the project.
- Change the window title and icon.
- Request attention (will cause the title bar to blink on most platforms).
- Fullscreen mode.
 - Uses borderless fullscreen by default on Windows for fast alt-tabbing, but can optionally use exclusive fullscreen to reduce input lag.
- Borderless windows (fullscreen or non-fullscreen).
- Ability to keep a window always on top.
- Global menu integration on macOS.
- Execute commands in a blocking or non-blocking manner (including running multiple instances of the same project).
- Open file paths and URLs using default or custom protocol handlers (if registered on the system).
- Parse custom command line arguments.
- Any Godot binary (editor or exported project) can be used as a headless server by starting it with the `--headless` command line argument. This allows running the engine without a GPU or display server.

2.2.19 Mobile

- In-app purchases on Android and iOS.
- Support for advertisements using third-party modules.

2.2.20 XR support (AR and VR)

- Out of the box support for OpenXR.
 - Including support for popular desktop headsets like the Valve Index, WMR headsets, and Quest over Link.
- Support for Android based headsets using OpenXR through a plugin.
 - Including support for popular stand alone headsets like the Meta Quest 1/2/3 and Pro, Pico 4, Magic Leap 2, and Lynx R1.
- Other devices supported through an XR plugin structure.
- Various advanced toolkits are available that implement common features required by XR applications.

2.2.21 GUI system

Godot's GUI is built using the same Control nodes used to make games in Godot. The editor UI can easily be extended in many ways using add-ons.

Nodes:

- Buttons.
- Checkboxes, check buttons, radio buttons.
- Text entry using LineEdit (single line) and TextEdit (multiple lines). TextEdit also supports code editing features such as displaying line numbers and syntax highlighting.
- Dropdown menus using PopupMenu and OptionButton.
- Scrollbars.
- Labels.
- RichTextLabel for text formatted using BBCode, with support for animated custom effects.
- Trees (can also be used to represent tables).
- Color picker with RGB and HSV modes.
- Controls can be rotated and scaled.

Sizing:

- Anchors to keep GUI elements in a specific corner, edge or centered.
- Containers to place GUI elements automatically following certain rules.
 - Stack layouts.
 - Grid layouts.
 - Flow layouts (similar to autowrapping text).
 - Margin, centered and aspect ratio layouts.
 - Draggable splitter layouts.
- Scale to multiple resolutions using the `canvas_items` or `viewport stretch` modes.
- Support any aspect ratio using anchors and the `expand stretch` aspect.

Theming:

- Built-in theme editor.
 - Generate a theme based on the current editor theme settings.
- Procedural vector-based theming using `StyleBoxFlat`.
 - Supports rounded/beveled corners, drop shadows, per-border widths and antialiasing.
- Texture-based theming using `StyleBoxTexture`.

Godot's small distribution size can make it a suitable alternative to frameworks like Electron or Qt.

2.2.22 Animation

- Direct kinematics and inverse kinematics.
- Support for animating any property with customizable interpolation.
- Support for calling methods in animation tracks.
- Support for playing sounds in animation tracks.
- Support for Bézier curves in animation.

2.2.23 File formats

- Scenes and resources can be saved in text-based or binary formats.
 - Text-based formats are human-readable and more friendly to version control.
 - Binary formats are faster to save/load for large scenes/resources.
- Read and write text or binary files using FileAccess.
 - Can optionally be compressed or encrypted.
- Read and write JSON files.
- Read and write INI-style configuration files using ConfigFile.
 - Can (de)serialize any Godot datatype, including Vector2/3, Color, ...
- Read XML files using XMLParser.
- Load and save images, audio/video, fonts and ZIP archives in an exported project without having to go through Godot's import system.
- Pack game data into a PCK file (custom format optimized for fast seeking), into a ZIP archive, or directly into the executable for single-file distribution.
- Export additional PCK files that can be read by the engine to support mods and DLCs.

2.2.24 Miscellaneous

- Video playback with built-in support for Ogg Theora.
- Movie Maker mode to record videos from a running project with synchronized audio and perfect frame pacing.
- Low-level access to servers which allows bypassing the scene tree's overhead when needed.
- Command line interface for automation.
 - Export and deploy projects using continuous integration platforms.
 - [Shell completion scripts](#) are available for Bash, zsh and fish.
 - Print colored text to standard output on all platforms using `print_rich`.
- Support for C++ modules statically linked into the engine binary.
- Engine and editor written in C++17.
 - Can be compiled using GCC, Clang and MSVC. MinGW is also supported.
 - Friendly towards packagers. In most cases, system libraries can be used instead of the ones provided by Godot. The build system doesn't download anything. Builds can be fully reproducible.
- Licensed under the permissive MIT license.

- Open development process with contributions welcome.

See also:

The [Godot proposals repository](#) lists features that have been requested by the community and may be implemented in future Godot releases.

2.3 Frequently asked questions

2.3.1 What can I do with Godot? How much does it cost? What are the license terms?

Godot is [Free and open source Software](#) available under the [OSI-approved MIT license](#). This means it is free as in "free speech" as well as in "free beer."

In short:

- You are free to download and use Godot for any purpose: personal, non-profit, commercial, or otherwise.
- You are free to modify, distribute, redistribute, and remix Godot to your heart's content, for any reason, both non-commercially and commercially.

All the contents of this accompanying documentation are published under the permissive Creative Commons Attribution 3.0 ([CC BY 3.0](#)) license, with attribution to "Juan Linietsky, Ariel Manzur and the Godot Engine community."

Logos and icons are generally under the same Creative Commons license. Note that some third-party libraries included with Godot's source code may have different licenses.

For full details, look at the [COPYRIGHT.txt](#) as well as the [LICENSE.txt](#) and [LOGO_LICENSE.txt](#) files in the Godot repository.

Also, see [the license page on the Godot website](#).

2.3.2 Which platforms are supported by Godot?

For the editor:

- Windows
- macOS
- Linux, *BSD
- Android (experimental)
- [Web](#) (experimental)

For exporting your games:

- Windows
- macOS
- Linux, *BSD
- Android
- iOS
- Web

Both 32- and 64-bit binaries are supported where it makes sense, with 64 being the default. Official macOS builds support Apple Silicon natively as well as x86_64.

Some users also report building and using Godot successfully on ARM-based systems with Linux, like the Raspberry Pi.

The Godot team can't provide an open source console export due to the licensing terms imposed by console manufacturers. Regardless of the engine you use, though, releasing games on consoles is always a lot of work. You can read more about Console support in Godot.

For more on this, see the sections on exporting and compiling Godot yourself.

Note: Godot 3 also had support for Universal Windows Platform (UWP). This platform port was removed in Godot 4 due to lack of maintenance, and it being deprecated by Microsoft. It is still available in the current stable release of Godot 3 for interested users.

2.3.3 Which programming languages are supported in Godot?

The officially supported languages for Godot are GDScript, C#, and C++. See the subcategories for each language in the scripting section.

If you are just starting out with either Godot or game development in general, GDScript is the recommended language to learn and use since it is native to Godot. While scripting languages tend to be less performant than lower-level languages in the long run, for prototyping, developing Minimum Viable Products (MVPs), and focusing on Time-To-Market (TTM), GDScript will provide a fast, friendly, and capable way of developing your games.

Note that C# support is still relatively new, and as such, you may encounter some issues along the way. C# support is also currently missing on the web platform. Our friendly and hard-working development community is always ready to tackle new problems as they arise, but since this is an open source project, we recommend that you first do some due diligence yourself. Searching through discussions on [open issues](#) is a great way to start your troubleshooting.

As for new languages, support is possible via third parties with GDExtensions. (See the question about plugins below). Work is currently underway, for example, on unofficial bindings for Godot to [Python](#) and [Nim](#).

2.3.4 What is GDScript and why should I use it?

GDScript is Godot's integrated scripting language. It was built from the ground up to maximize Godot's potential in the least amount of code, affording both novice and expert developers alike to capitalize on Godot's strengths as fast as possible. If you've ever written anything in a language like Python before, then you'll feel right at home. For examples and a complete overview of the power GDScript offers you, check out the GDScript scripting guide.

There are several reasons to use GDScript, especially when you are prototyping, in alpha/beta stages of your project, or are not creating the next AAA title. The most salient reason is the overall reduction of complexity.

The original intent of creating a tightly integrated, custom scripting language for Godot was two-fold: first, it reduces the amount of time necessary to get up and running with Godot, giving developers a rapid way of exposing themselves to the engine with a focus on productivity; second, it reduces the overall burden of maintenance, attenuates the dimensionality of issues, and allows the developers of the engine to focus on squashing bugs and improving features related to the engine core, rather than spending a lot of time trying to get a small set of incremental features working across a large set of languages.

Since Godot is an open source project, it was imperative from the start to prioritize a more integrated and seamless experience over attracting additional users by supporting more familiar programming languages, especially when supporting those more familiar languages would result in a worse experience. We understand if you would rather use another language in Godot (see the list of supported options above). That being said, if you haven't given GDScript a try, try it for three days. Just like Godot, once you see how powerful it is and rapid your development becomes, we think GDScript will grow on you.

More information about getting comfortable with GDScript or dynamically typed languages can be found in the [GDScript: An introduction to dynamic languages tutorial](#).

2.3.5 What were the motivations behind creating GDScript?

In the early days, the engine used the [Lua](#) scripting language. Lua can be fast thanks to LuaJIT, but creating bindings to an object-oriented system (by using fallbacks) was complex and slow and took an enormous amount of code. After some experiments with [Python](#), that also proved difficult to embed.

The main reasons for creating a custom scripting language for Godot were:

1. Poor threading support in most script VMs, and Godot uses threads (Lua, Python, Squirrel, JavaScript, ActionScript, etc.).
2. Poor class-extending support in most script VMs, and adapting to the way Godot works is highly inefficient (Lua, Python, JavaScript).
3. Many existing languages have horrible interfaces for binding to C++, resulting in a large amount of code, bugs, bottlenecks, and general inefficiency (Lua, Python, Squirrel, JavaScript, etc.). We wanted to focus on a great engine, not a great number of integrations.
4. No native vector types (Vector3, Transform3D, etc.), resulting in highly reduced performance when using custom types (Lua, Python, Squirrel, JavaScript, ActionScript, etc.).
5. Garbage collector results in stalls or unnecessarily large memory usage (Lua, Python, JavaScript, ActionScript, etc.).
6. Difficulty integrating with the code editor for providing code completion, live editing, etc. (all of them).

GDScript was designed to curtail the issues above, and more.

2.3.6 What 3D model formats does Godot support?

You can find detailed information on supported formats, how to export them from your 3D modeling software, and how to import them for Godot in the [Importing 3D scenes documentation](#).

2.3.7 Will [insert closed SDK such as FMOD, GameWorks, etc.] be supported in Godot?

The aim of Godot is to create a free and open source MIT-licensed engine that is modular and extendable. There are no plans for the core engine development community to support any third-party, closed-source/proprietary SDKs, as integrating with these would go against Godot's ethos.

That said, because Godot is open source and modular, nothing prevents you or anyone else interested in adding those libraries as a module and shipping your game with them, as either open- or closed-source.

To see how support for your SDK of choice could still be provided, look at the [Plugins](#) question below.

If you know of a third-party SDK that is not supported by Godot but that offers free and open source integration, consider starting the integration work yourself. Godot is not owned by one person; it belongs to the community, and it grows along with ambitious community contributors like you.

2.3.8 How can I extend Godot?

For extending Godot, like creating Godot Editor plugins or adding support for additional languages, take a look at [EditorPlugins](#) and tool scripts.

Also, see the official blog post on [GDExtension](#), a way to develop native extensions for Godot:

- [Introducing GDNative's successor, GDExtension](#)

You can also take a look at the [GDScript](#) implementation, the Godot modules, as well as the [Jolt physics engine integration](#) for Godot. This would be a good starting point to see how another third-party library integrates with Godot.

2.3.9 How do I install the Godot editor on my system (for desktop integration)?

Since you don't need to actually install Godot on your system to run it, this means desktop integration is not performed automatically. There are two ways to overcome this. You can install Godot from [Steam](#) (all platforms), [Scoop](#) (Windows), [Homebrew](#) (macOS) or [Flathub](#) (Linux). This will automatically perform the required steps for desktop integration.

Alternatively, you can manually perform the steps that an installer would do for you:

Windows

- Move the Godot executable to a stable location (i.e. outside of your Downloads folder), so you don't accidentally move it and break the shortcut in the future.
- Right-click the Godot executable and choose Create Shortcut.
- Move the created shortcut to %APPDATA%\Microsoft\Windows\Start Menu\Programs. This is the user-wide location for shortcuts that will appear in the Start menu. You can also pin Godot in the task bar by right-clicking the executable and choosing Pin to Task Bar.

macOS

Drag the extracted Godot application to /Applications/Godot.app, then drag it to the Dock if desired. Spotlight will be able to find Godot as long as it's in /Applications or ~/Applications.

Linux

- Move the Godot binary to a stable location (i.e. outside of your Downloads folder), so you don't accidentally move it and break the shortcut in the future.
- Rename and move the Godot binary to a location present in your PATH environment variable. This is typically /usr/local/bin/godot or /usr/bin/godot. Doing this requires administrator privileges, but this also allows you to run the Godot editor from a terminal by entering godot.
 - If you cannot move the Godot editor binary to a protected location, you can keep the binary somewhere in your home directory, and modify the Path= line in the .desktop file linked below to contain the full absolute path to the Godot binary.
- Save [this .desktop file](#) to \$HOME/.local/share/applications/. If you have administrator privileges, you can also save the .desktop file to /usr/local/share/applications to make the shortcut available for all users.

2.3.10 Is the Godot editor a portable application?

In its default configuration, Godot is semi-portable. Its executable can run from any location (including non-writable locations) and never requires administrator privileges.

However, configuration files will be written to the user-wide configuration or data directory. This is usually a good approach, but this means configuration files will not carry across machines if you copy the folder containing the Godot executable. See [File paths in Godot projects](#) for more information.

If true portable operation is desired (e.g. for use on an USB stick), follow the steps in [Self-contained mode](#).

2.3.11 Why does Godot use Vulkan or OpenGL instead of Direct3D?

Godot aims for cross-platform compatibility and open standards first and foremost. OpenGL and Vulkan are the technologies that are both open and available on (nearly) all platforms. Thanks to this design decision, a project developed with Godot on Windows will run out of the box on Linux, macOS, and more.

Since Godot only has a few people working on its renderer, we would prefer having fewer rendering backends to maintain. On top of that, using a single API on all platforms allows for greater consistency with fewer platform-specific issues.

In the long term, we may develop a Direct3D 12 renderer for Godot (mainly for Xbox), but Vulkan and OpenGL will remain the default rendering backends on all platforms, including Windows.

2.3.12 Why does Godot aim to keep its core feature set small?

Godot intentionally does not include features that can be implemented by add-ons unless they are used very often. One example of something not used often is advanced artificial intelligence functionality.

There are several reasons for this:

- Code maintenance and surface for bugs. Every time we accept new code in the Godot repository, existing contributors often take the responsibility of maintaining it. Some contributors don't always stick around after getting their code merged, which can make it difficult for us to maintain the code in question. This can lead to poorly maintained features with bugs that are never fixed. On top of that, the "API surface" that needs to be tested and checked for regressions keeps increasing over time.
- Ease of contribution. By keeping the codebase small and tidy, it can remain fast and easy to compile from source. This makes it easier for new contributors to get started with Godot, without requiring them to purchase high-end hardware.
- Keeping the binary size small for the editor. Not everyone has a fast Internet connection. Ensuring that everyone can download the Godot editor, extract it and run it in less than 5 minutes makes Godot more accessible to developers in all countries.
- Keeping the binary size small for export templates. This directly impacts the size of projects exported with Godot. On mobile and web platforms, keeping file sizes low is important to ensure fast installation and loading on underpowered devices. Again, there are many countries where high-speed Internet is not readily available. To add to this, strict data usage caps are often in effect in those countries.

For all the reasons above, we have to be selective of what we can accept as core functionality in Godot. This is why we are aiming to move some core functionality to officially supported add-ons in future versions of Godot. In terms of binary size, this also has the advantage of making you pay only for what you actually use in your project. (In the meantime, you can compile custom export templates with unused features disabled to optimize the distribution size of your project.)

2.3.13 How should assets be created to handle multiple resolutions and aspect ratios?

This question pops up often and it's probably thanks to the misunderstanding created by Apple when they originally doubled the resolution of their devices. It made people think that having the same assets in different resolutions was a good idea, so many continued towards that path. That originally worked to a point and only for Apple devices, but then several Android and Apple devices with different resolutions and aspect ratios were created, with a very wide range of sizes and DPIs.

The most common and proper way to achieve this is to, instead, use a single base resolution for the game and only handle different screen aspect ratios. This is mostly needed for 2D, as in 3D, it's just a matter of camera vertical or horizontal FOV.

1. Choose a single base resolution for your game. Even if there are devices that go up to 1440p and devices that go down to 400p, regular hardware scaling in your device will take care of this at little or no performance cost. The most common choices are either near 1080p (1920x1080) or 720p (1280x720). Keep in mind the higher the resolution, the larger your assets, the more memory they will take and the longer the time it will take for loading.
2. Use the stretch options in Godot; canvas items stretching while keeping aspect ratios works best. Check the [Multiple resolutions](#) tutorial on how to achieve this.
3. Determine a minimum resolution and then decide if you want your game to stretch vertically or horizontally for different aspect ratios, or if there is one aspect ratio and you want black bars to appear instead. This is also explained in [Multiple resolutions](#).
4. For user interfaces, use the anchoring to determine where controls should stay and move. If UIs are more complex, consider learning about Containers.

And that's it! Your game should work in multiple resolutions.

2.3.14 When is the next release of Godot out?

When it's ready! See [When is the next release out?](#) for more information.

2.3.15 Which Godot version should I use for a new project?

We recommend using Godot 4.x for new projects, but depending on the feature set you need, it may be better to use 3.x instead. See [Which version should I use for a new project?](#) for more information.

2.3.16 Should I upgrade my project to use new Godot versions?

Some new versions are safer to upgrade to than others. In general, whether you should upgrade depends on your project's circumstances. See [Should I upgrade my project to use new engine versions?](#) for more information.

2.3.17 I would like to contribute! How can I get started?

Awesome! As an open source project, Godot thrives off of the innovation and the ambition of developers like you.

The best way to start contributing to Godot is by using it and reporting any [issues](#) that you might experience. A good bug report with clear reproduction steps helps your fellow contributors fix bugs quickly and efficiently. You can also report issues you find in the [online documentation](#).

If you feel ready to submit your first PR, pick any issue that resonates with you from one of the links above and try your hand at fixing it. You will need to learn how to compile the engine from sources, or how to build the documentation. You also need to get familiar with Git, a version control system that Godot developers use.

We explain how to work with the engine source, how to edit the documentation, and what other ways to contribute are there in our documentation for contributors.

2.3.18 I have a great idea for Godot. How can I share it?

We are always looking for suggestions about how to improve the engine. User feedback is the main driving force behind our decision-making process, and limitations that you might face while working on your project are a great data point for us when considering engine enhancements.

If you experience a usability problem or are missing a feature in the current version of Godot, start by discussing it with our [community](#). There may be other, perhaps better, ways to achieve the desired result that community members could suggest. And you can learn if other users experience the same issue, and figure out a good solution together.

If you come up with a well-defined idea for the engine, feel free to open a [proposal issue](#). Try to be specific and concrete while describing your problem and your proposed solution — only actionable proposals can be considered. It is not required, but if you want to implement it yourself, that's always appreciated!

If you only have a general idea without specific details, you can open a [proposal discussion](#). These can be anything you want, and allow for a free-form discussion in search of a solution. Once you find one, a proposal issue can be opened.

Please, read the [readme](#) document before creating a proposal to learn more about the process.

2.3.19 Is it possible to use Godot to create non-game applications?

Yes! Godot features an extensive built-in UI system, and its small distribution size can make it a suitable alternative to frameworks like Electron or Qt.

When creating a non-game application, make sure to enable low-processor mode in the Project Settings to decrease CPU and GPU usage.

Check out [Material Maker](#) and [Pixelorama](#) for examples of open source applications made with Godot.

2.3.20 Is it possible to use Godot as a library?

Godot is meant to be used with its editor. We recommend you give it a try, as it will most likely save you time in the long term. There are no plans to make Godot usable as a library, as it would make the rest of the engine more convoluted and difficult to use for casual users.

If you want to use a rendering library, look into using an established rendering engine instead. Keep in mind rendering engines usually have smaller communities compared to Godot. This will make it more difficult to find answers to your questions.

2.3.21 What user interface toolkit does Godot use?

Godot does not use a standard GUI (Graphical User Interface) toolkit like GTK, Qt or wxWidgets. Instead, Godot uses its own user interface toolkit, rendered using OpenGL ES or Vulkan. This toolkit is exposed in the form of Control nodes, which are used to render the editor (which is written in C++). These Control nodes can also be used in projects from any scripting language supported by Godot.

This custom toolkit makes it possible to benefit from hardware acceleration and have a consistent appearance across all platforms. On top of that, it doesn't have to deal with the LGPL licensing caveats that come with GTK or Qt. Lastly, this means Godot is "eating its own dog food" since the editor itself is one of the most complex users of Godot's UI system.

This custom UI toolkit [can't be used as a library](#), but you can still [use Godot to create non-game applications by using the editor](#).

2.3.22 Why does Godot use the SCons build system?

Godot uses the [SCons](#) build system. There are no plans to switch to a different build system in the near future. There are many reasons why we have chosen SCons over other alternatives. For example:

- Godot can be compiled for a dozen different platforms: all PC platforms, all mobile platforms, many consoles, and WebAssembly.
- Developers often need to compile for several of the platforms at the same time, or even different targets of the same platform. They can't afford reconfiguring and rebuilding the project each time. SCons can do this with no sweat, without breaking the builds.
- SCons will never break a build no matter how many changes, configurations, additions, removals etc.
- Godot's build process is not simple. Several files are generated by code (binders), others are parsed (shaders), and others need to offer customization (modules). This requires complex logic which is easier to write in an actual programming language (like Python) rather than using a mostly macro-based language only meant for building.
- Godot build process makes heavy use of cross-compiling tools. Each platform has a specific detection process, and all these must be handled as specific cases with special code written for each.

Please try to keep an open mind and get at least a little familiar with SCons if you are planning to build Godot yourself.

2.3.23 Why does Godot not use STL (Standard Template Library)?

Like many other libraries (Qt as an example), Godot does not make use of STL (with a few exceptions such as threading primitives). We believe STL is a great general-purpose library, but we had special requirements for Godot.

- STL templates create very large symbols, which results in huge debug binaries. We use few templates with very short names instead.
- Most of our containers cater to special needs, like `Vector`, which uses copy on write and we use to pass data around, or the `RID` system, which requires $O(1)$ access time for performance. Likewise, our hash map implementations are designed to integrate seamlessly with internal engine types.
- Our containers have memory tracking built-in, which helps better track memory usage.
- For large arrays, we use pooled memory, which can be mapped to either a preallocated buffer or virtual memory.
- We use our custom `String` type, as the one provided by STL is too basic and lacks proper internationalization support.

2.3.24 Why does Godot not use exceptions?

We believe games should not crash, no matter what. If an unexpected situation happens, Godot will print an error (which can be traced even to script), but then it will try to recover as gracefully as possible and keep going.

Additionally, exceptions significantly increase the binary size for the executable and result in increased compile times.

2.3.25 Does Godot use an ECS (Entity Component System)?

Godot does not use an ECS and relies on inheritance instead. While there is no universally better approach, we found that using an inheritance-based approach resulted in better usability while still being fast enough for most use cases.

That said, nothing prevents you from making use of composition in your project by creating child Nodes with individual scripts. These nodes can then be added and removed at run-time to dynamically add and remove behaviors.

More information about Godot's design choices can be found in [this article](#).

2.3.26 Why does Godot not force users to implement DOD (Data-Oriented Design)?

While Godot internally attempts to use cache coherency as much as possible, we believe users don't need to be forced to use DOD practices.

DOD is mostly a cache coherency optimization that can only provide significant performance improvements when dealing with dozens of thousands of objects which are processed every frame with little modification. That is, if you are moving a few hundred sprites or enemies per frame, DOD won't result in a meaningful improvement in performance. In such a case, you should consider a different approach to optimization.

The vast majority of games do not need this and Godot provides handy helpers to do the job for most cases when you do.

If a game needs to process such a large amount of objects, our recommendation is to use C++ and GDExtensions for performance-heavy tasks and GDScript (or C#) for the rest of the game.

2.3.27 How can I support Godot development or contribute?

See [Ways to contribute](#).

2.3.28 Who is working on Godot? How can I contact you?

See the corresponding page on the [Godot website](#).

2.4 Complying with licenses

2.4.1 What are licenses?

Godot is created and distributed under the [MIT License](#). It doesn't have a sole owner either, as every contributor that submits code to the project does it under this same license and keeps ownership of the contribution.

The license is the legal requirement for you (or your company) to use and distribute the software (and derivative projects, including games made with it). Your game or project can have a different license, but it still needs to comply with the original one.

Note: This section covers compliance with licenses from a user perspective. If you are interested in licence compliance as a contributor, you can find guidelines [here](#).

Warning: In your project's credits screen, remember to also list third-party notices for assets you're using, such as textures, models, sounds, music and fonts.

Free assets in particular often come with licenses that require attribution. Double-check their license before using those assets in a project.

2.4.2 Requirements

In the case of the MIT license, the only requirement is to include the license text somewhere in your game or derivative project.

This text reads as follows:

This game uses Godot Engine, available under the following license:

Copyright (c) 2014-present Godot Engine contributors. Copyright (c) 2007-2014 Juan Linietsky, Ariel Manzur.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Note: Your games do not need to be under the same license. You are free to release your Godot projects under any license and to create commercial games with the engine.

2.4.3 Inclusion

The license does not specify how it has to be included, so anything is valid as long as it can be displayed under some condition. These are the most common approaches (only need to implement one of them, not all).

Tip: Godot provides several methods to get license information in the Engine singleton. This allows you to source the license information directly from the engine binary, which prevents the information from becoming outdated if you update engine versions.

For the engine itself:

- `Engine.get_license_text`

For third-party components used by the engine:

- `Engine.get_license_info`
- `Engine.get_copyright_info`

For miscellaneous engine contributor information. You don't have to include these ones in your project, but they're listed here for reference:

- `Engine.get_author_info`
 - `Engine.get_donor_info`
-

Credits screen

Include the above license text somewhere in the credits screen. It can be at the bottom after showing the rest of the credits. Most large studios use this approach with open source licenses.

Licenses screen

Some games have a special menu (often in the settings) to display licenses. This menu is typically accessed with a button called Third-party Licenses or Open Source Licenses.

Output log

Printing the licensing text using the `print()` function may be enough on platforms where a global output log is readable. This is the case on desktop platforms, Android and HTML5 (but not iOS).

Accompanying file

If the game is distributed on desktop platforms, a file containing the license can be added to the software that is installed to the user PC.

Printed manual

If the game includes printed manuals, license text can be included there.

Link to the license

The Godot Engine developers consider that a link to godotengine.org/license in your game documentation or credits would be an acceptable way to satisfy the license terms.

2.4.4 Third-party licenses

Godot itself contains software written by [third parties](#). Most of it does not require license inclusion, but some do. Make sure to do it if these are compiled in your Godot export template. If you're using the official export templates, all libraries are enabled. This means you need to provide attribution for all the libraries listed below.

Here's a list of libraries requiring attribution:

FreeType

Godot uses [FreeType](#) to render fonts. Its license requires attribution, so the following text must be included together with the Godot license:

Portions of this software are copyright © <year> The FreeType Project (www.freetype.org).
All rights reserved.

Note: <year> should correspond to the value from the FreeType version used in your build. This information can be found in the editor by opening the Help > About dialog and going to the Third-party Licenses tab.

ENet

Godot includes the [ENet](#) library to handle high-level multiplayer. ENet has similar licensing terms as Godot:

Copyright (c) 2002-2020 Lee Salzman

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

mbed TLS

If the project is exported with Godot 3.1 or later, it includes [mbed TLS](#). The Apache license needs to be complied to by including the following text:

Copyright The Mbed TLS Contributors

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Note: If you exported your project using a custom build with specific modules disabled, you don't need to list the disabled modules' licenses in your exported project.

2.5 Godot release policy

Godot's release policy is in constant evolution. The description below provides a general idea of what to expect, but what will actually happen depends on the choices of core contributors and the needs of the community at a given time.

2.5.1 Godot versioning

Godot loosely follows [Semantic Versioning](#) with a major.minor.patch versioning system, albeit with an interpretation of each term adapted to the complexity of a game engine:

- The major version is incremented when major compatibility breakages happen which imply significant porting work to move projects from one major version to another.

For example, porting Godot projects from Godot 3.x to Godot 4.x requires running the project through a conversion tool, and then performing a number of further adjustments manually for what the tool could not do automatically.

- The minor version is incremented for feature releases that do not break compatibility in a major way. Minor compatibility breakage in very specific areas may happen in minor versions, but the vast majority of projects should not be affected or require significant porting work.

This is because Godot, as a game engine, covers many areas like rendering, physics, and scripting. Fixing bugs or implementing new features in one area might sometimes require changing a feature's behavior or modifying a class's interface, even if the rest of the engine API remains backwards compatible.

Tip: Upgrading to a new minor version is recommended for all users, but some testing is necessary to ensure that your project still behaves as expected.

- The patch version is incremented for maintenance releases which focus on fixing bugs and security issues, implementing new requirements for platform support, and backporting safe usability enhancements. Patch releases are backwards compatible.

Patch versions may include minor new features which do not impact the existing API, and thus have no risk of impacting existing projects.

Tip: Updating to new patch versions is therefore considered safe and strongly recommended to all users of a given stable branch.
















We call major.minor combinations stable branches. Each stable branch starts with a major.minor release (without the 0 for patch) and is further developed for maintenance releases in a Git branch of the same name (for example patch updates for the 4.0 stable branch are developed in the 4.0 Git branch).

2.5.2 Release support timeline

Stable branches are supported at least until the next stable branch is released and has received its first patch update. In practice, we support stable branches on a best effort basis for as long as they have active users who need maintenance updates.

Whenever a new major version is released, we make the previous stable branch a long-term supported release, and do our best to provide fixes for issues encountered by users of that branch who cannot port complex projects to the new major version. This was the case for the 2.1 branch, and is the case for the 3.6 branch.

In a given minor release series, only the latest patch release receives support. If you experience an issue using an older patch release, please upgrade to the latest patch release of that series and test again before reporting an issue on GitHub.

Version	Release date	Support level
Godot 4.3 (master)	April 2024 (estimate)	 Development. Receives new features, usability and performance improvements, as well as bug fixes, while under development.
Godot 4.2	November 2023	 Receives fixes for bugs and security issues, as well as patches that enable platform support.
Godot 4.1	July 2023	 Receives fixes for bugs and security issues, as well as patches that enable platform support.
Godot 4.0	March 2023	 No longer supported (last update: 4.0.4).
Godot 3.6 (3.x, LTS)	Q1 2024 (estimate)	 Beta. Receives new features, usability and performance improvements, as well as bug fixes, while under development.
Godot 3.5	August 2022	 Receives fixes for bugs and security issues, as well as patches that enable platform support.
Godot 3.4	November 2021	 No longer supported (last update: 3.4.5).
Godot 3.3	April 2021	 No longer supported (last update: 3.3.4).
Godot 3.2	January 2020	 No longer supported (last update: 3.2.3).
Godot 3.1	March 2019	 No longer supported (last update: 3.1.2).
Godot 3.0	January 2018	 No longer supported (last update: 3.0.6).
Godot 2.1	July 2016	 No longer supported (last update: 2.1.6).
Godot 2.0	February 2016	 No longer supported (last update: 2.0.4.1).
Godot 1.1	May 2015	 No longer supported.
Godot 1.0	December 2014	 No longer supported.

Legend:  Full support –  Partial support –  No support (end of life) –  Development version

Pre-release Godot versions aren't intended to be used in production and are provided for testing purposes only.

See also:

See [Upgrading from Godot 3 to Godot 4](#) for instructions on migrating a project from Godot 3.x to 4.x.

2.5.3 Which version should I use for a new project?

We recommend using Godot 4.x for new projects, as the Godot 4.x series will be supported long after 3.x stops receiving updates in the future. One caveat is that a lot of third-party documentation hasn't been updated for Godot 4.x yet. If you have to follow a tutorial designed for Godot 3.x, we recommend keeping [Upgrading from Godot 3 to Godot 4](#) open in a separate tab to check which methods have been renamed (if you get a script error while trying to use a specific node or method that was renamed in Godot 4.x).

If your project requires a feature that is missing in 4.x (such as GLES2/WebGL 1.0), you should use Godot 3.x for a new project instead.

2.5.4 Should I upgrade my project to use new engine versions?

Note: Upgrading software while working on a project is inherently risky, so consider whether it's a good idea for your project before attempting an upgrade. Also, make backups of your project or use version control to prevent losing data in case the upgrade goes wrong.

That said, we do our best to keep minor and especially patch releases compatible with existing projects.

The general recommendation is to upgrade your project to follow new patch releases, such as upgrading from 4.0.2 to 4.0.3. This ensures you get bug fixes, security updates and platform support updates (which is especially important for mobile platforms). You also get continued support, as only the last patch release receives support on official community platforms.

For minor releases, you should determine whether it's a good idea to upgrade on a case-by-case basis. We've made a lot of effort in making the upgrade process as seamless as possible, but some breaking changes may be present in minor releases, along with a greater risk of regressions. Some fixes included in minor releases may also change a class' expected behavior as required to fix some bugs. This is especially the case in classes marked as experimental in the documentation.

Major releases bring a lot of new functionality, but they also remove previously existing functionality and may raise hardware requirements. They also require much more work to upgrade to compared to minor releases. As a result, we recommend sticking with the major release you've started your project with if you are happy with how your project currently works. For example, if your project was started with 3.5, we recommend upgrading to 3.5.2 and possibly 3.6 in the future, but not to 4.0+, unless your project really needs the new features that come with 4.0+.

2.5.5 When is the next release out?

While Godot contributors aren't working under any deadlines, we strive to publish minor releases relatively frequently.

In particular, after the very length release cycle for 4.0, we are pivoting to a faster paced development workflow, with the 4.1 release expected within late Q2 / early Q3 2023.

Frequent minor releases will enable us to ship new features faster (possibly as experimental), get user feedback quickly, and iterate to improve those features and their usability. Likewise, the general user experience will be improved more steadily with a faster path to the end users.

Maintenance (patch) releases are released as needed with potentially very short development cycles, to provide users of the current stable branch with the latest bug fixes for their production needs.

The 3.6 release is still planned and should be the last stable branch of Godot 3.x. It will be a Long-Term Support (LTS) release, which we plan to support for as long as users still need it (due to missing features in Godot 4.x, or having published games which they need to keep updating for platform requirements).

2.5.6 What are the criteria for compatibility across engine versions?

Note: This section is intended to be used by contributors to determine which changes are safe for a given release. The list is not exhaustive; it only outlines the most common situations encountered during Godot's development.

The following changes are acceptable in patch releases:

- Fixing a bug in a way that has no major negative impact on most projects, such as a visual or physics bug. Godot's physics engine is not deterministic, so physics bug fixes are not considered to break

compatibility. If fixing a bug has a negative impact that could impact a lot of projects, it should be made optional (e.g. using a project setting or separate method).

- Adding a new optional parameter to a method.
- Small-scale editor usability tweaks.

Note that we tend to be more conservative with the fixes we allow in each subsequent patch release. For instance, 4.0.1 may receive more impactful fixes than 4.0.4 would.

The following changes are acceptable in minor releases, but not patch releases:

- Significant new features.
- Renaming a method parameter. In C#, method parameters can be passed by name (but not in GDScript). As a result, this can break some projects that use C#.
- Deprecating a method, member variable, or class. This is done by adding a deprecated flag to its class reference, which will show up in the editor. When a method is marked as deprecated, it's slated to be removed in the next major release.
- Changes that affect the default project theme's visuals.
- Bug fixes which significantly change the behavior or the output, with the aim to meet user expectations better. In comparison, in patch releases, we may favor keeping a buggy behavior so we don't break existing projects which likely already rely on the bug or use a workaround.
- Performance optimizations that result in visual changes.

The following changes are considered compatibility-breaking and can only be performed in a new major release:

- Renaming or removing a method, member variable, or class.
- Modifying a node's inheritance tree by making it inherit from a different class.
- Changing the default value of a project setting value in a way that affects existing projects. To only affect new projects, the project manager should write a modified `project.godot` instead.

Since Godot 5.0 hasn't been branched off yet, we currently discourage making compatibility-breaking changes of this kind.

Note: When modifying a method's signature in any fashion (including adding an optional parameter), a `GDEExtension` compatibility method must be created. This ensures that existing `GDEExtensions` continue to work across patch and minor releases, so that users don't have to recompile them. See [pull request #76446](#) for more information.

2.6 Documentation changelog

The documentation is continually being improved. New releases include new pages, fixes and updates to existing pages, and many updates to the class reference. Below is a list of new pages added since version 3.0.

Note: This document only contains new pages so not all changes are reflected, many pages have been substantially updated but are not reflected in this document.

2.6.1 New pages since version 4.1

C#

- C# diagnostics

Development

- 2D coordinate systems and 2D transforms

Migrating

- Upgrading from Godot 4.1 to Godot 4.2

I/O

- Runtime file loading and saving

Platform-specific

- Godot Android library

2.6.2 New pages since version 4.0

Development

- Internal rendering architecture
- Using sanitizers

Migrating

- Upgrading from Godot 4.0 to Godot 4.1

Physics

- Troubleshooting physics issues

2.6.3 New pages since version 3.6

2D

- 2D antialiasing

3D

- 3D antialiasing
- Faking global illumination
- Introduction to global illumination
- Mesh level of detail (LOD)
- Occlusion culling
- Signed distance field global illumination (SDFGI)
- Using decals
- Visibility ranges (HLOD)

- Volumetric fog and fog volumes
- Variable rate shading
- Physical light and camera units

Animation

- Creating movies

Assets pipeline

- Retargeting 3D Skeletons

Development

- Custom platform ports

Migrating

- Upgrading from Godot 3 to Godot 4

Physics

- Large world coordinates

Scripting

- Custom performance monitors
- C# collections
- C# global classes
- C# Variant

Shaders

- Using compute shaders

Workflow

- Pull request review process

XR

- Introducing XR tools
- The XR action map
- Deploying to Android

2.6.4 New pages since version 3.5

None.

2.6.5 New pages since version 3.4

3D

- [3D text](#)

Animation

- [Playing videos](#)

Editor

- [Managing editor features](#)

2.6.6 New pages since version 3.3

C++

- [C++ usage guidelines](#)

GDScript

- [GDScript documentation comments](#)

2.6.7 New pages since version 3.2

3D

- [3D rendering limitations](#)

About

- [Troubleshooting](#)
- [List of features](#)
- [Godot release policy](#)

Best practices

- [Version control systems](#)

Community

- [Best practices for engine contributors](#)
- [Bisecting regressions](#)
- [Editor and documentation localization](#)

Development

- Introduction to editor development
- Editor style guide
- Common engine methods and macros
- Validation layers
- GDScript grammar
- Configuring an IDE: Code::Blocks

Editor

- Default editor shortcuts
- Using the Web editor

Export

- Exporting for dedicated servers

Input

- Controllers, gamepads, and joysticks

Math

- Random number generation

Platform-specific

- Plugins for iOS
- Creating iOS plugins
- HTML5 shell class reference

Physics

- Collision shapes (2D)
- Collision shapes (3D)

Shaders

- Shaders style guide

Scripting

- Debugger panel
- Creating script templates
- Evaluating expressions
- What is GDExtension?
- GDScript warning system (split from Static typing in GDScript)

User Interface (UI)

- Control node gallery

2.6.8 New pages since version 3.1

Project workflow

- Gradle builds for Android

2D

- 2D sprite animation

Audio

- Recording with microphone
- Sync the gameplay with audio and music

Math

- Beziers, curves and paths
- Interpolation

Inputs

- Input examples

Internationalization

- Localization using gettext

Shading

- Your First Shader Series:
 - Introduction to shaders
 - Your first 2D shader
 - Your first 3D shader
 - Your second 3D shader
- Using VisualShaders

Networking

- WebRTC

Plugins

- [Godot Android plugins](#)
- [Inspector plugins](#)
- [Visual Shader plugins](#)

Multi-threading

- [Using multiple threads](#)

Creating content

Procedural geometry series:

- [Procedural geometry](#)
- [Using the ArrayMesh](#)
- [Using the SurfaceTool](#)
- [Using the MeshDataTool](#)
- [Using ImmediateMesh](#)

Optimization

- [Optimization using MultiMeshes](#)
- [Optimization using Servers](#)

Legal

- [Complying with licenses](#)

2.6.9 New pages since version 3.0

Step by step

- [Using signals](#)
- [Exporting](#)

Scripting

- [Static typing in GDScript](#)

Project workflow

Best Practices:

- [Introduction](#)
- [Applying object-oriented principles in Godot](#)
- [Scene organization](#)
- [When to use scenes versus scripts](#)
- [Autoloads versus regular nodes](#)
- [When and how to avoid using nodes for everything](#)
- [Godot interfaces](#)

- Godot notifications
- Data preferences
- Logic preferences

2D

- 2D lights and shadows
- 2D meshes

3D

- Prototyping levels with CSG
- Animating thousands of fish with MultiMeshInstance3D
- Controlling thousands of fish with Particles

Physics

- Ragdoll system
- Using SoftBody

Animation

- 2D skeletons
- Using AnimationTree

GUI

- Using Containers

Viewports

- Using a Viewport as a texture
- Custom post-processing

Shading

- Converting GLSL to Godot shaders
- Advanced post-processing

Shading Reference:

- Introduction to shaders
- Shading language
- Spatial shaders
- CanvasItem shaders
- Particle shaders

Plugins

- Making main screen plugins
- 3D gizmo plugins

Platform-specific

- Custom HTML page for Web export

Multi-threading

- Thread-safe APIs

Creating content

- Making trees

Miscellaneous

- Fixing jitter, stutter and input lag
- Running code in the editor
- Change scenes manually

Compiling

- Optimizing a build for size
- Compiling with PCK encryption key

Engine development

- Binding to external libraries

2.7 Introduction

This series will introduce you to Godot and give you an overview of its features.

In the following pages, you will get answers to questions such as "Is Godot for me?" or "What can I do with Godot?". We will then introduce the engine's most essential concepts, run you through the editor's interface, and give you tips to make the most of your time learning it.

2.7.1 Introduction to Godot

This article is here to help you figure out whether Godot might be a good fit for you. We will introduce some broad features of the engine to give you a feel for what you can achieve with it and answer questions such as "what do I need to know to get started?".

This is by no means an exhaustive overview. We will introduce many more features in this getting started series.

What is Godot?

Godot is a general-purpose 2D and 3D game engine designed to support all sorts of projects. You can use it to create games or applications you can then release on desktop or mobile, as well as on the web.

You can also create console games with it, although you either need strong programming skills or a developer to port the game for you.

Note: The Godot team can't provide an open source console export due to the licensing terms imposed by console manufacturers. Regardless of the engine you use, though, releasing games on consoles is always a lot of work. You can read more on that here: [Console support in Godot](#).

What can the engine do?

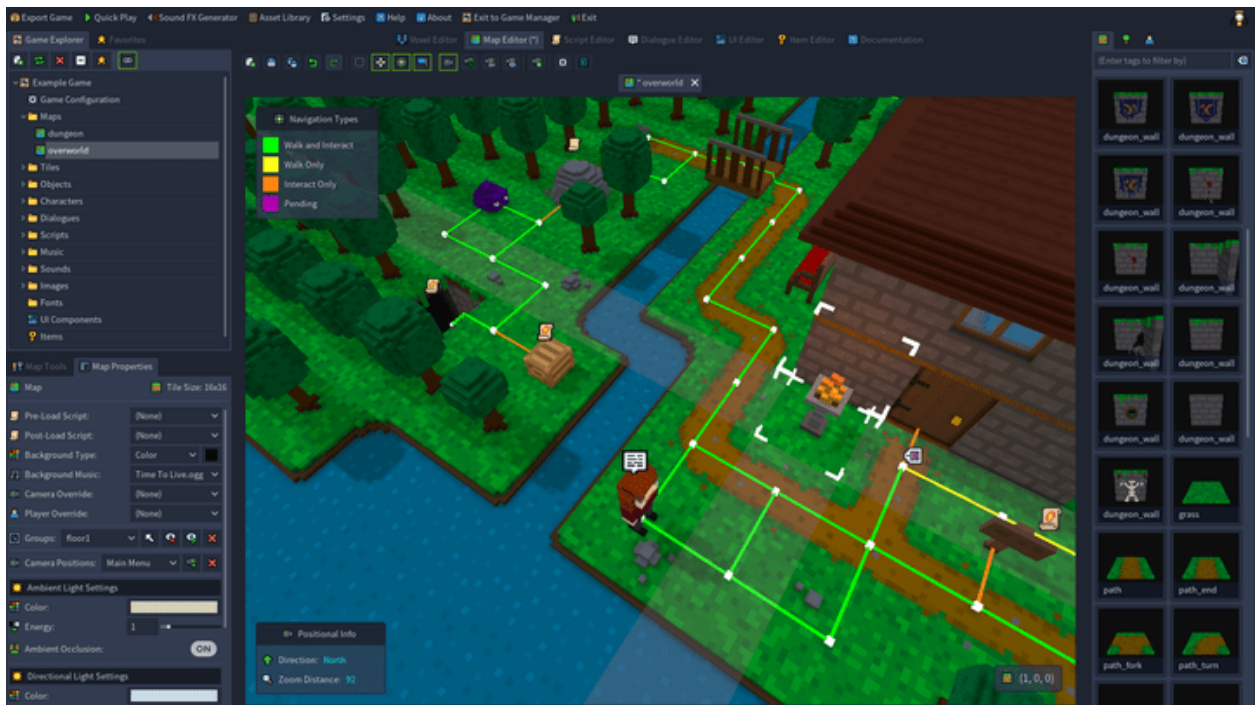
Godot was initially developed in-house by an Argentinian game studio. Its development started in 2001, and the engine was rewritten and improved tremendously since its open source release in 2014.

Some examples of games created with Godot include *Ex-Zodiac* and *Helms of Fury*.





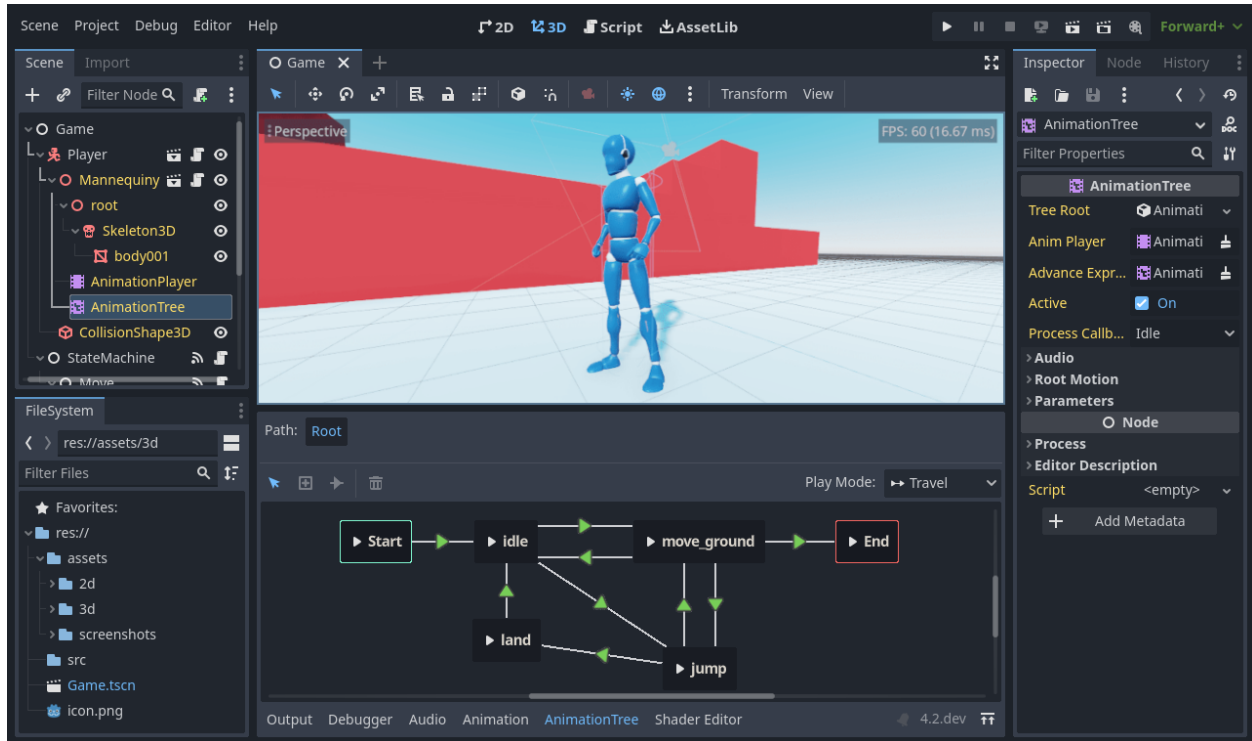
As for applications, the open source pixel art drawing program Pixelorama is powered by Godot, and so is the voxel RPG creator RPG in a box.



You can find many more examples in the official showcase videos.

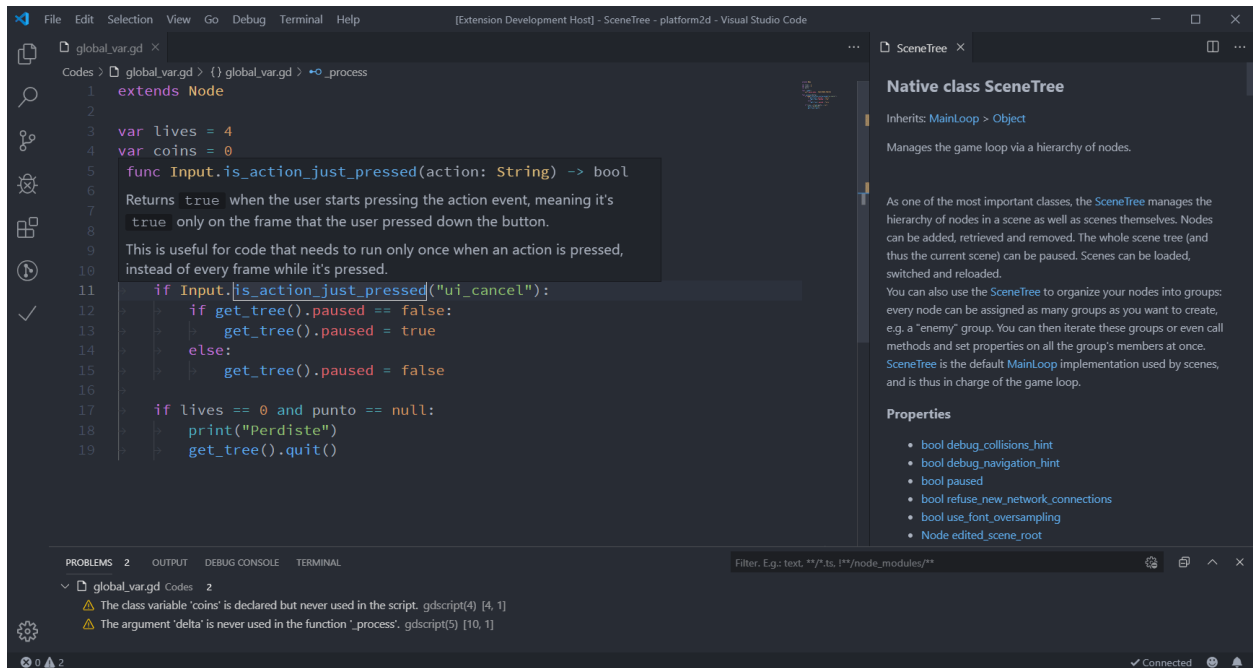
How does it work and look?

Godot comes with a fully-fledged game editor with integrated tools to answer the most common needs. It includes a code editor, an animation editor, a tilemap editor, a shader editor, a debugger, a profiler, and more.



The team strives to offer a feature-rich game editor with a consistent user experience. While there is always room for improvement, the user interface keeps getting refined.

Of course, if you prefer, you can work with external programs. We officially support importing 3D scenes designed in [Blender](#) and maintain plugins to code in [VSCode](#) and [Emacs](#) for GDScript and C#. We also support Visual Studio for C# on Windows.



Programming languages

Let's talk about the available programming languages.

You can code your games using GDScript, a Godot-specific and tightly integrated language with a light weight syntax, or C#, which is popular in the games industry. These are the two main scripting languages we support.

With the GDEXTENSION technology, you can also write gameplay or high-performance algorithms in C or C++ without recompiling the engine. You can use this technology to integrate third-party libraries and other Software Development Kits (SDK) in the engine.

Of course, you can also directly add modules and features to the engine, as it's completely free and open source.

What do I need to know to use Godot?

Godot is a feature-packed game engine. With its thousands of features, there is a lot to learn. To make the most of it, you need good programming foundations. While we try to make the engine accessible, you will benefit a lot from knowing how to think like a programmer first.

Godot relies on the object-oriented programming paradigm. Being comfortable with concepts such as classes and objects will help you code efficiently in it.

If you are entirely new to programming, we recommend following the [CS50 open courseware](#) from Harvard University. It's a great free course that will teach you everything you need to know to be off to a good start. It will save you countless hours and hurdles learning any game engine afterward.

Note: In CS50, you will learn multiple programming languages. Don't be afraid of that: programming languages have many similarities. The skills you learn with one language transfer well to others.

We will provide you with more Godot-specific learning resources in Learning new features.

In the next part, you will get an overview of the engine's essential concepts.

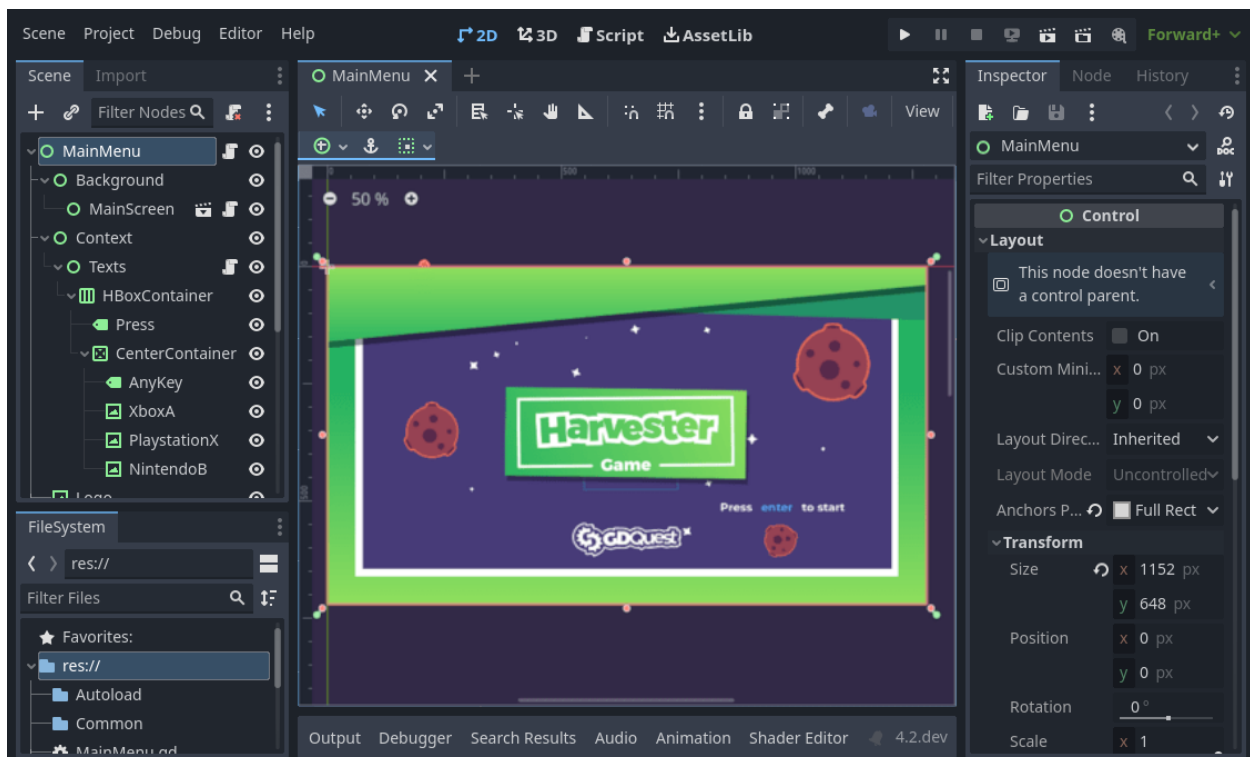
2.7.2 Overview of Godot's key concepts

Every game engine revolves around abstractions you use to build your applications. In Godot, a game is a tree of nodes that you group together into scenes. You can then wire these nodes so they can communicate using signals.

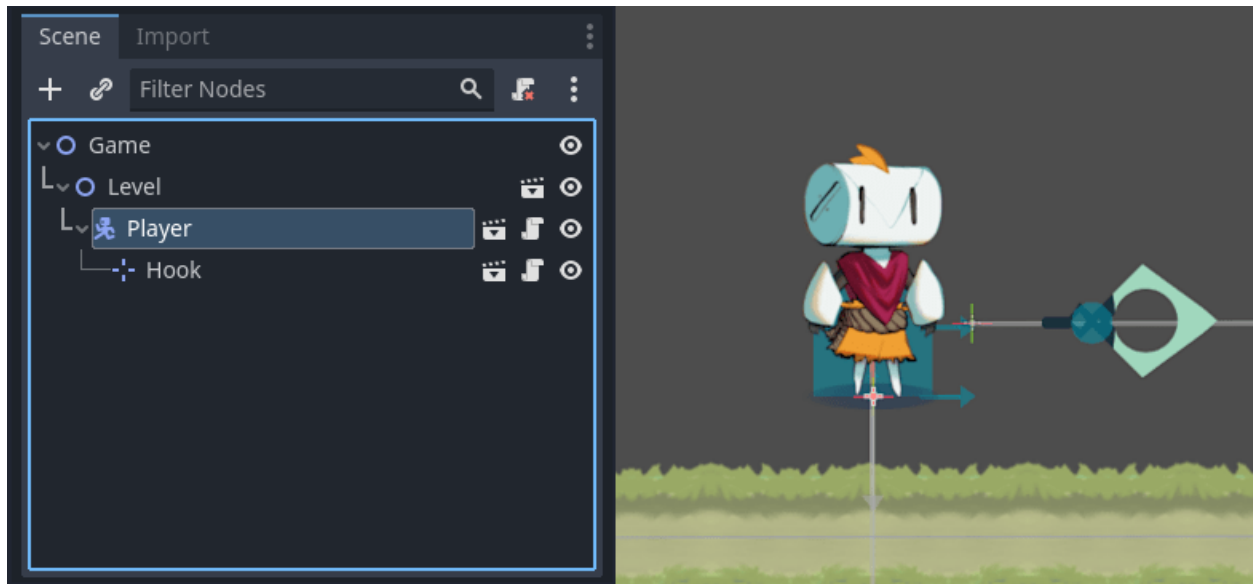
These are the four concepts you will learn here. We're going to look at them briefly to give you a sense of how the engine works. In the getting started series, you will get to use them in practice.

Scenes

In Godot, you break down your game in reusable scenes. A scene can be a character, a weapon, a menu in the user interface, a single house, an entire level, or anything you can think of. Godot's scenes are flexible; they fill the role of both prefabs and scenes in some other game engines.

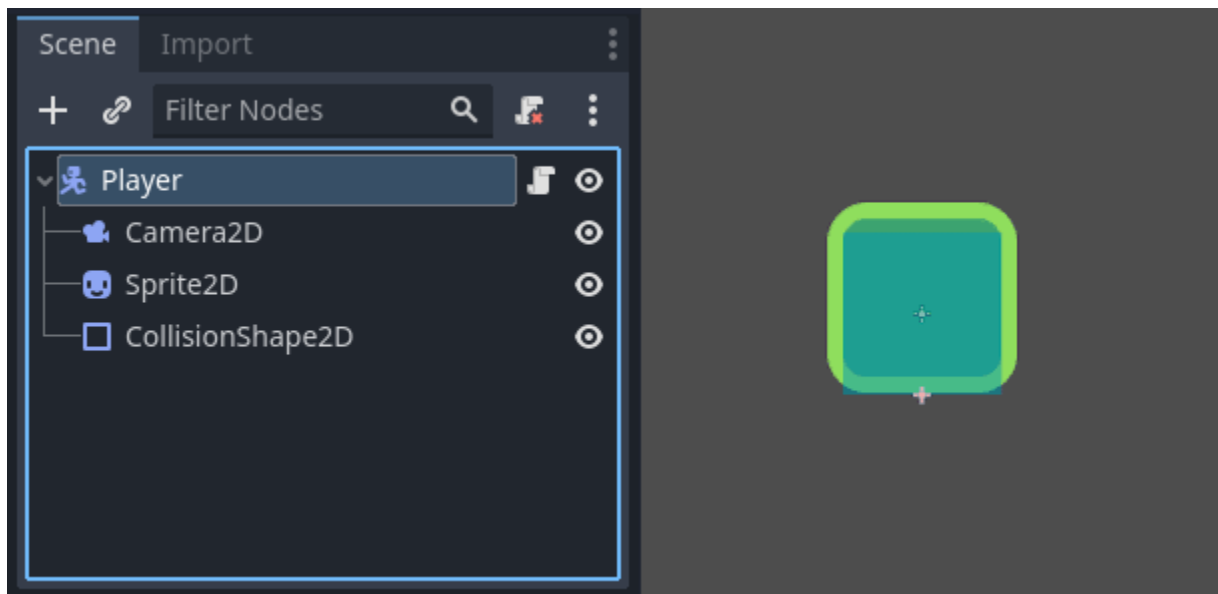


You can also nest scenes. For example, you can put your character in a level, and drag and drop a scene as a child of it.



Nodes

A scene is composed of one or more nodes. Nodes are your game's smallest building blocks that you arrange into trees. Here's an example of a character's nodes.

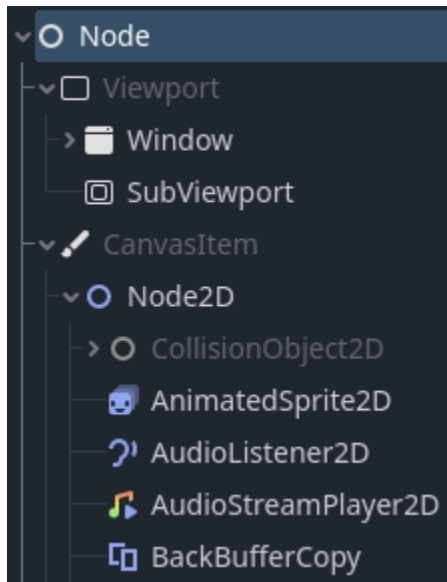


It is made of a `CharacterBody2D` node named "Player", a `Camera2D`, a `Sprite2D`, and a `CollisionShape2D`.

Note: The node names end with "2D" because this is a 2D scene. Their 3D counterparts have names that end with "3D". Be aware that "Spatial" Nodes are now called "Node3D" starting with Godot 4.

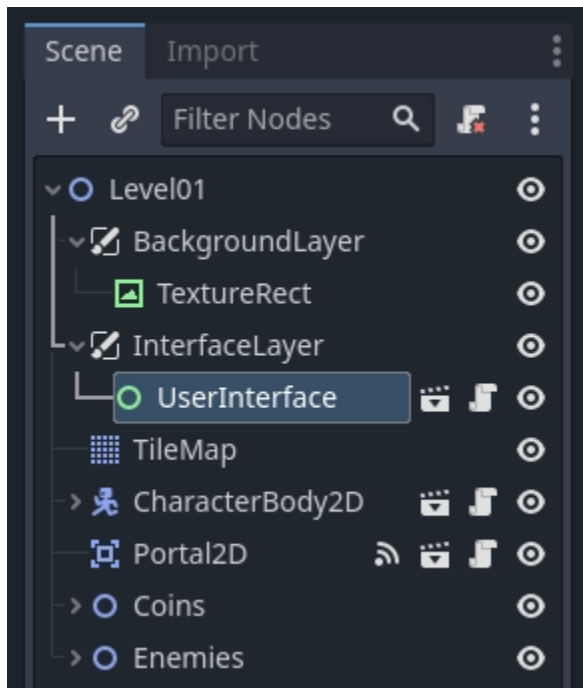
Notice how nodes and scenes look the same in the editor. When you save a tree of nodes as a scene, it then shows as a single node, with its internal structure hidden in the editor.

Godot provides an extensive library of base node types you can combine and extend to build more powerful ones. 2D, 3D, or user interface, you will do most things with these nodes.



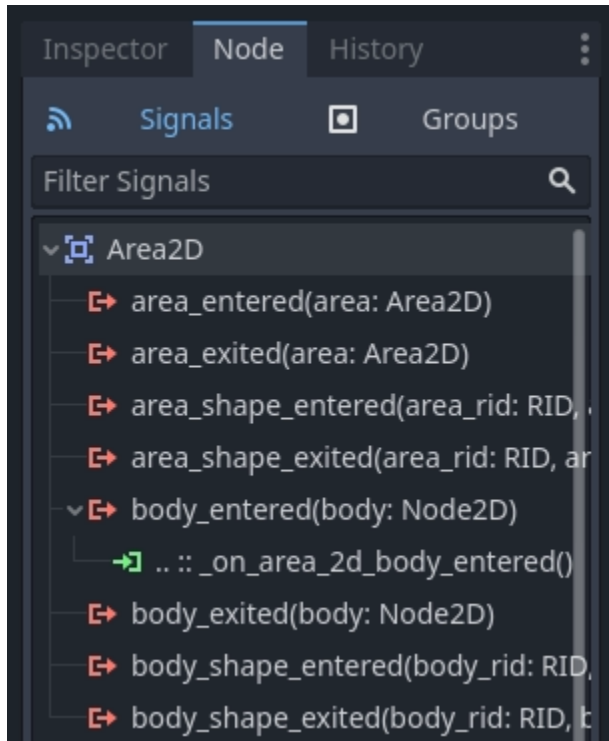
The scene tree

All your game's scenes come together in the scene tree, literally a tree of scenes. And as scenes are trees of nodes, the scene tree also is a tree of nodes. But it's easier to think of your game in terms of scenes as they can represent characters, weapons, doors, or your user interface.



Signals

Nodes emit signals when some event occurs. This feature allows you to make nodes communicate without hard-wiring them in code. It gives you a lot of flexibility in how you structure your scenes.



Note: Signals are Godot's version of the observer pattern. You can read more about it here: <https://gameprogrammingpatterns.com/observer.html>

For example, buttons emit a signal when pressed. You can connect to this signal to run code in reaction to this event, like starting the game or opening a menu.

Other built-in signals can tell you when two objects collided, when a character or monster entered a given area, and much more. You can also define new signals tailored to your game.

Summary

Nodes, scenes, the scene tree, and signals are four core concepts in Godot that you will manipulate all the time.

Nodes are your game's smallest building blocks. You combine them to create scenes that you then combine and nest into the scene tree. You can then use signals to make nodes react to events in other nodes or different scene tree branches.

After this short breakdown, you probably have many questions. Bear with us as you will get many answers throughout the getting started series.

2.7.3 First look at Godot's editor

This page will give you a brief overview of Godot's interface. We're going to look at the different main screens and docks to help you situate yourself.

See also:

For a comprehensive breakdown of the editor's interface and how to use it, see the [Editor manual](#).

The Project Manager

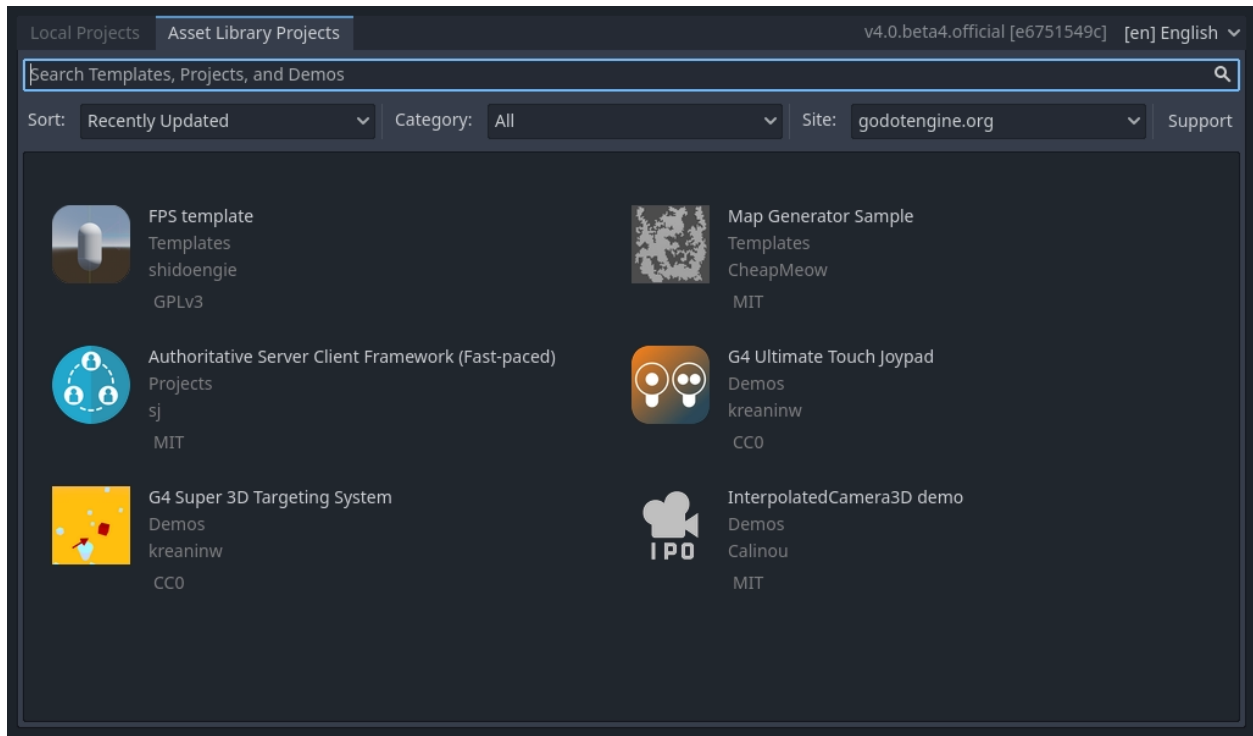
When you launch Godot, the first window you see is the Project Manager. In the default tab Local Projects, you can manage existing projects, import or create new ones, and more.



At the top of the window, there is another tab named "Asset Library Projects". You can search for demo projects in the open source asset library, which includes many projects developed by the community.

See also:

To learn the Project Manager's ins and outs, read [Using the Project Manager](#).

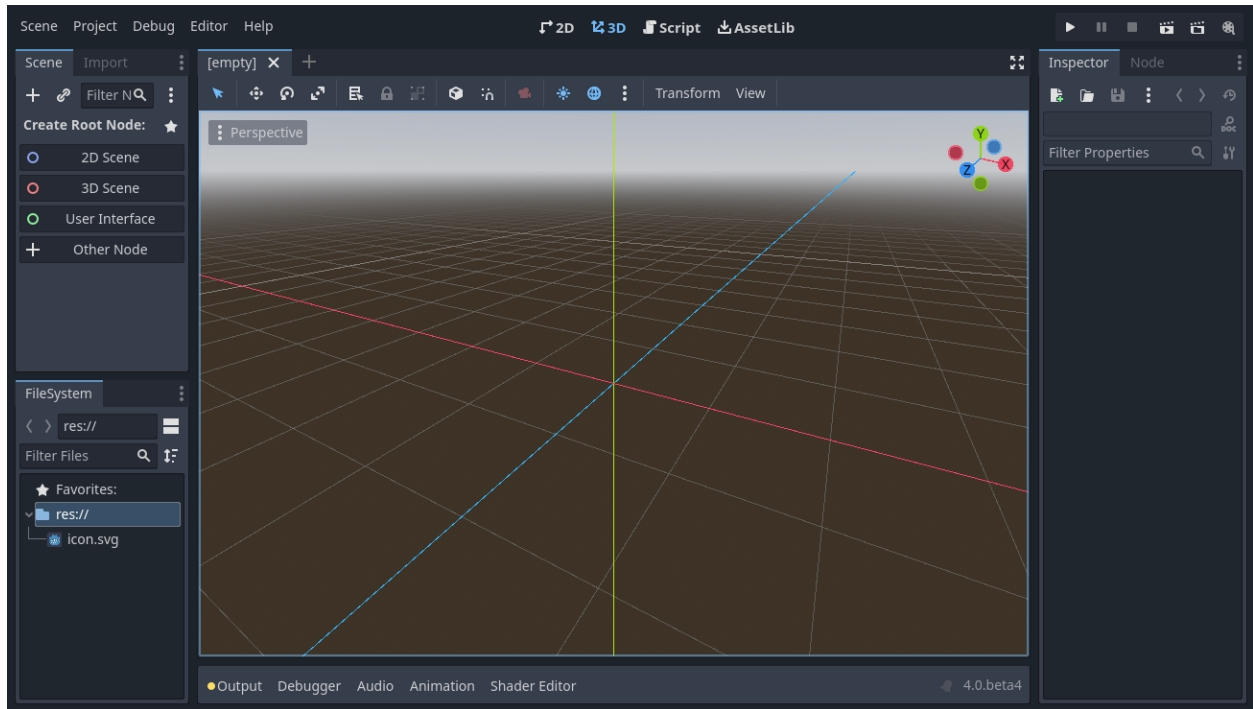


You can also change the editor's language using the drop-down menu to the right of the engine's version in the window's top-right corner. By default, it is in English (EN).



First look at Godot's editor

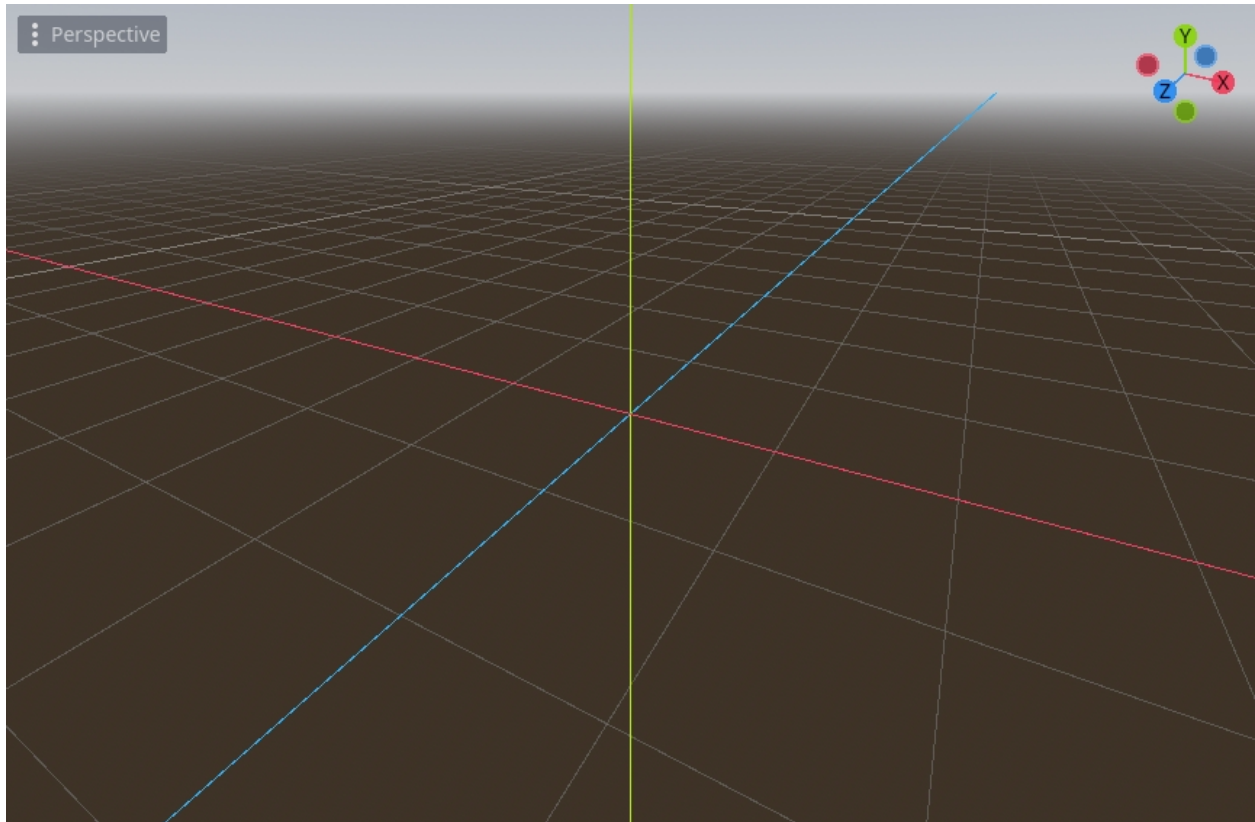
When you open a new or an existing project, the editor's interface appears. Let's look at its main areas.



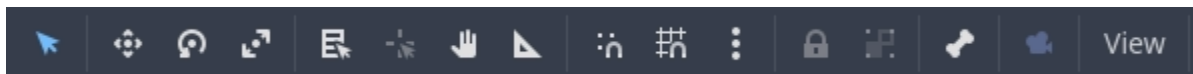
By default, it features menus, main screens, and playtest buttons along the window's top edge.



In the center is the viewport with its toolbar at the top, where you'll find tools to move, scale, or lock the scene's nodes.



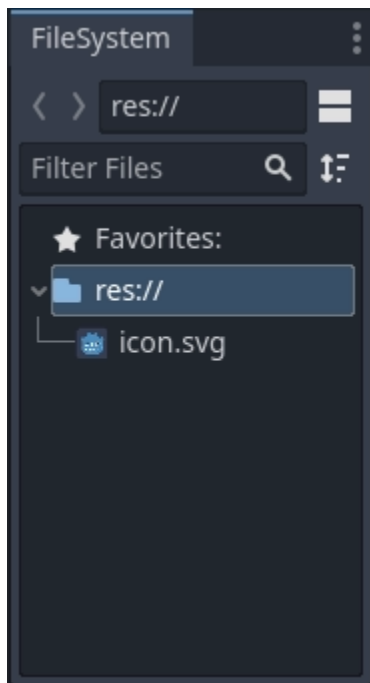
On either side of the viewport sit the docks. And at the bottom of the window lies the bottom panel. The toolbar changes based on the context and selected node. Here is the 2D toolbar.



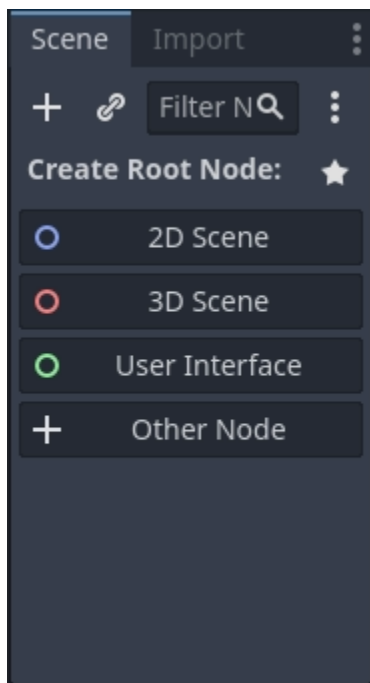
Below is the 3D one.



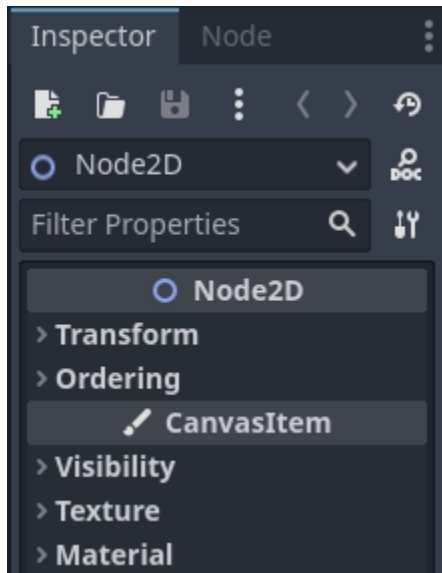
Let's look at the docks. The FileSystem dock lists your project files, including scripts, images, audio samples, and more.



The Scene dock lists the active scene's nodes.



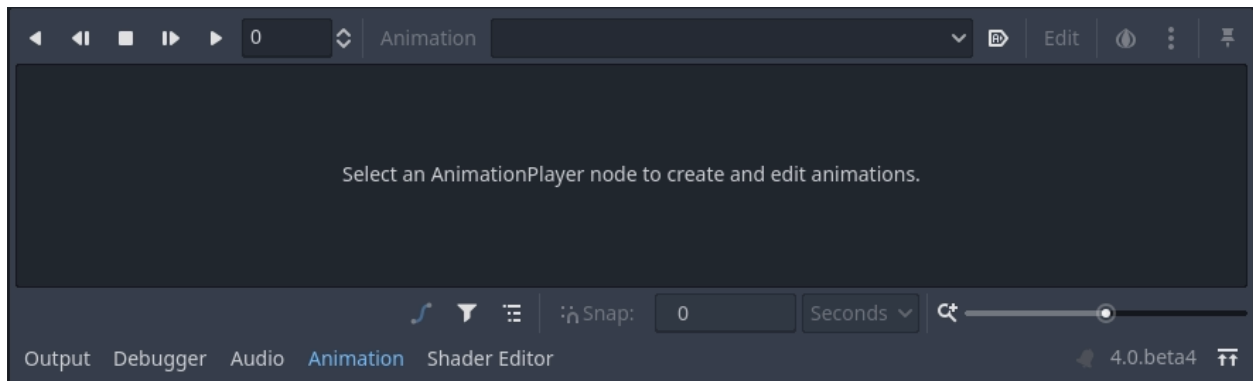
The Inspector allows you to edit the properties of a selected node.



The bottom panel, situated below the viewport, is the host for the debug console, the animation editor, the audio mixer, and more. They can take precious space, that's why they're folded by default.



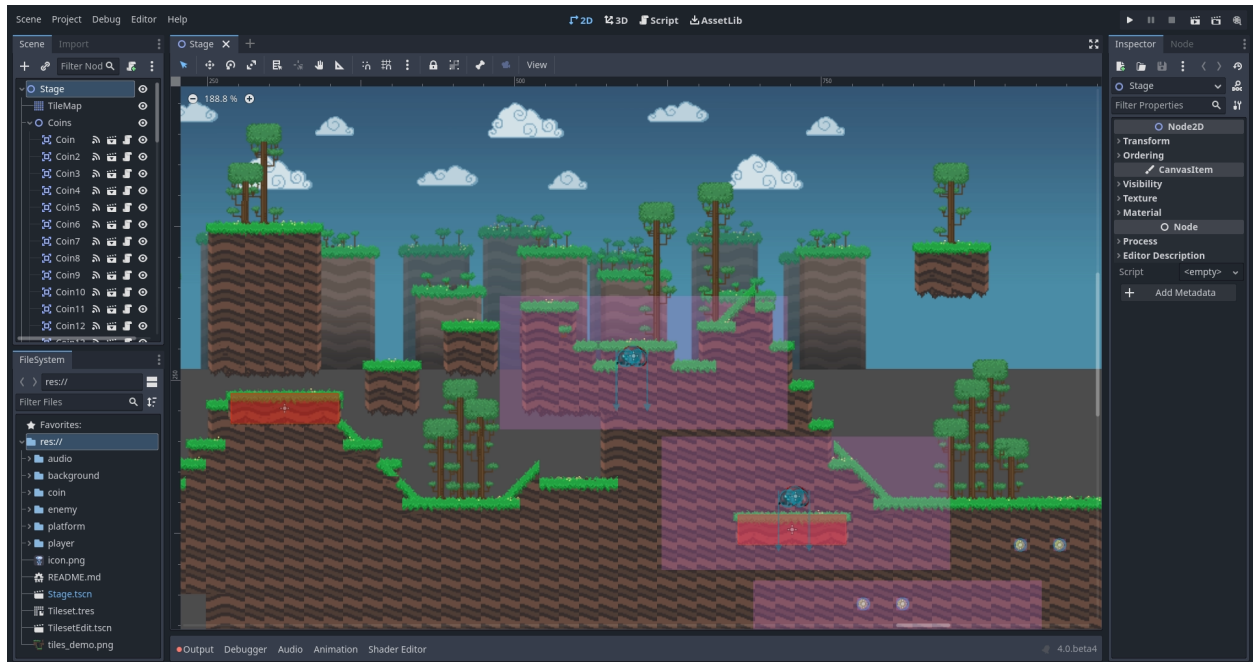
When you click on one, it expands vertically. Below, you can see the animation editor opened.



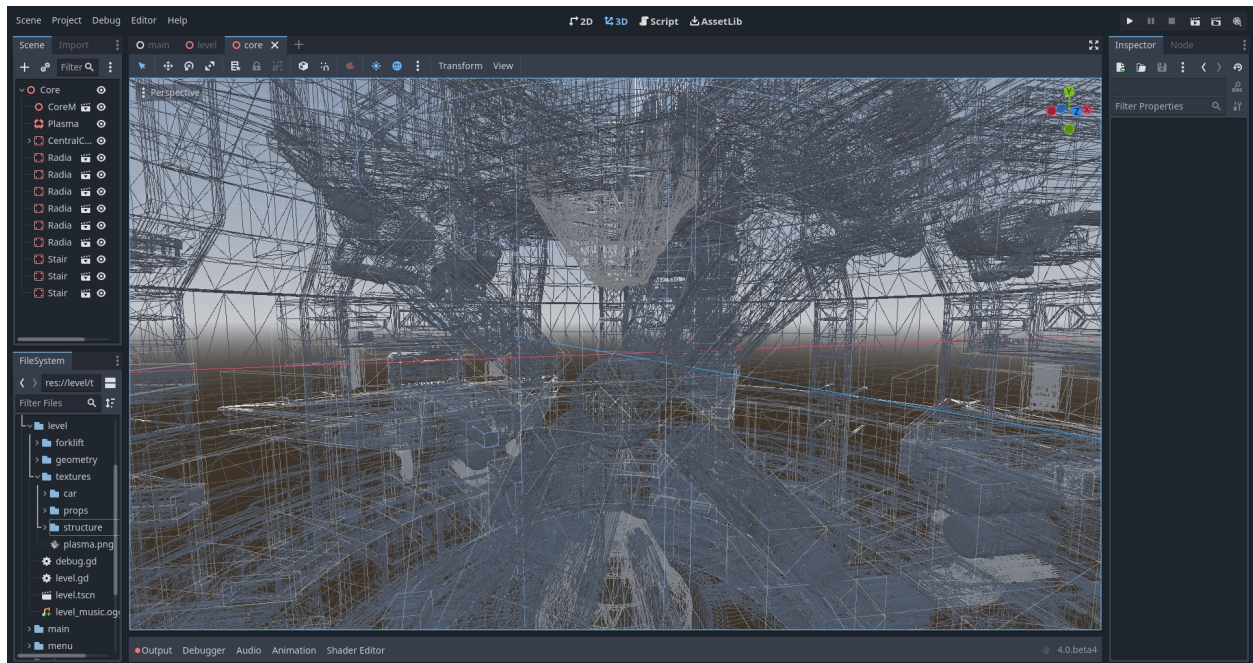
The four main screens

There are four main screen buttons centered at the top of the editor: 2D, 3D, Script, and AssetLib.

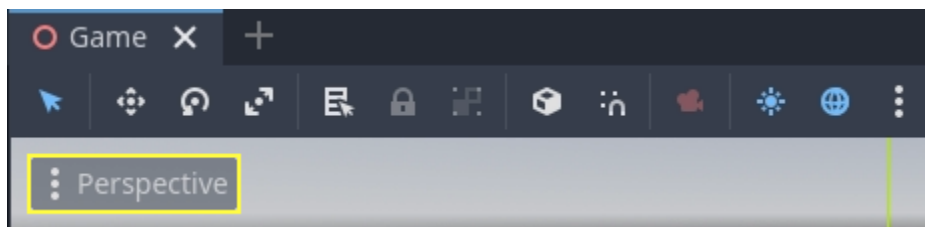
You'll use the 2D screen for all types of games. In addition to 2D games, the 2D screen is where you'll build your interfaces.



In the 3D screen, you can work with meshes, lights, and design levels for 3D games.

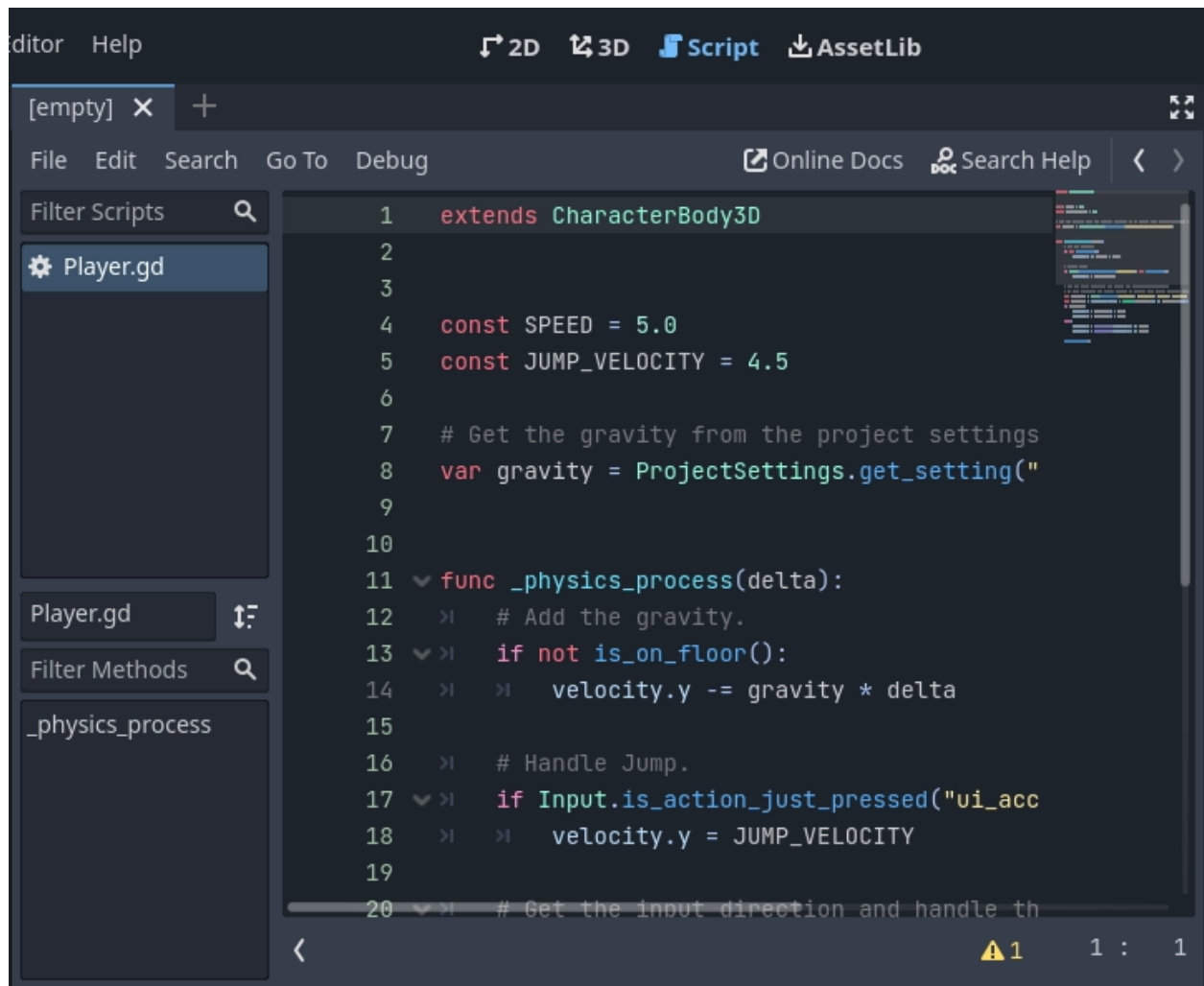


Notice the perspective button under the toolbar. Clicking on it opens a list of options related to the 3D view.

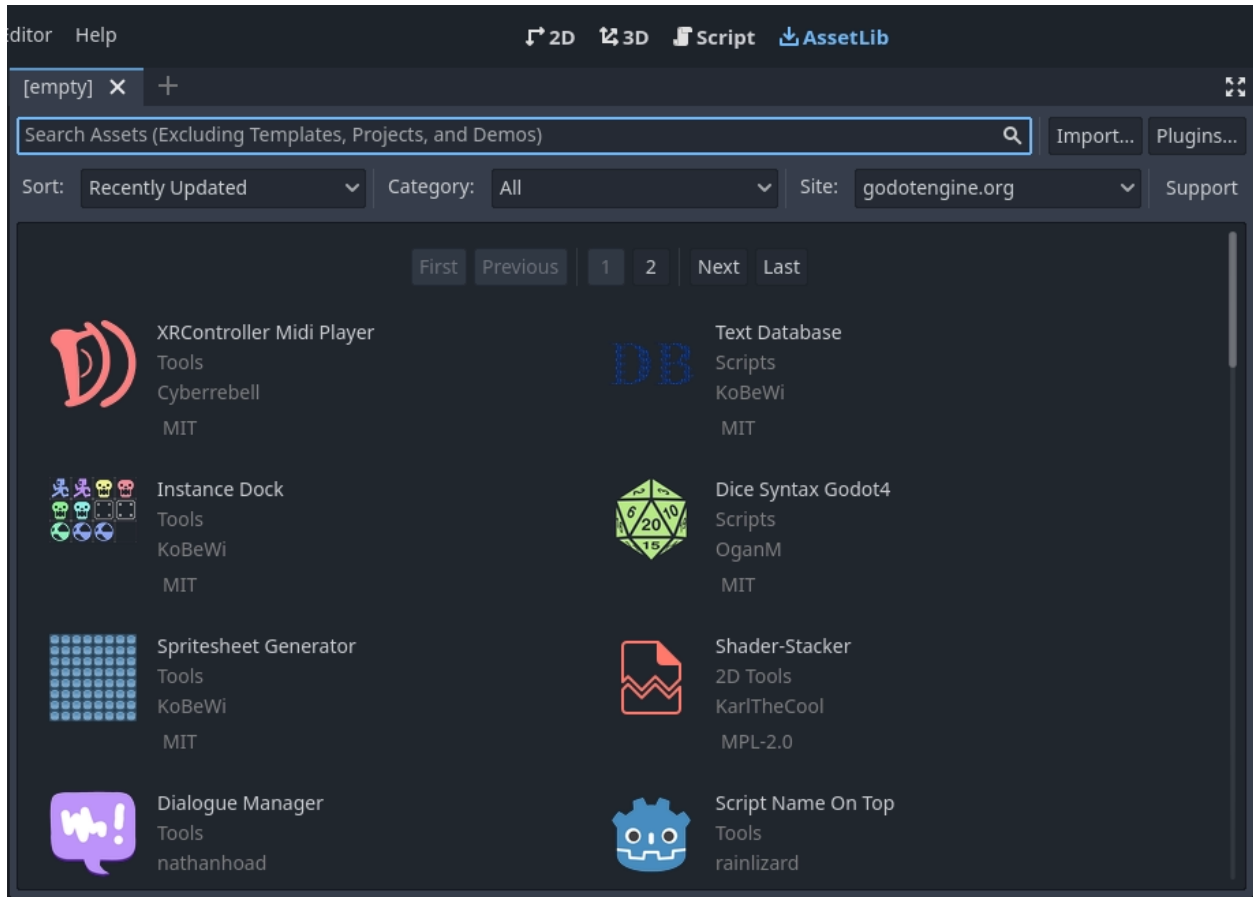


Note: Read Introduction to 3D for more detail about the 3D main screen.

The Script screen is a complete code editor with a debugger, rich auto-completion, and built-in code reference.



Finally, the AssetLib is a library of free and open source add-ons, scripts, and assets to use in your projects.



See also:

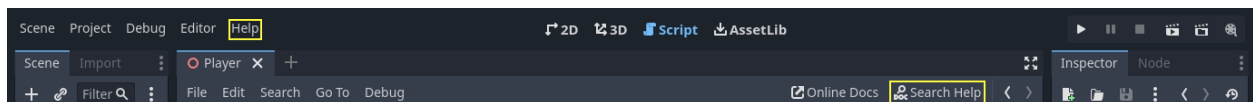
You can learn more about the asset library in [About the Asset Library](#).

Integrated class reference

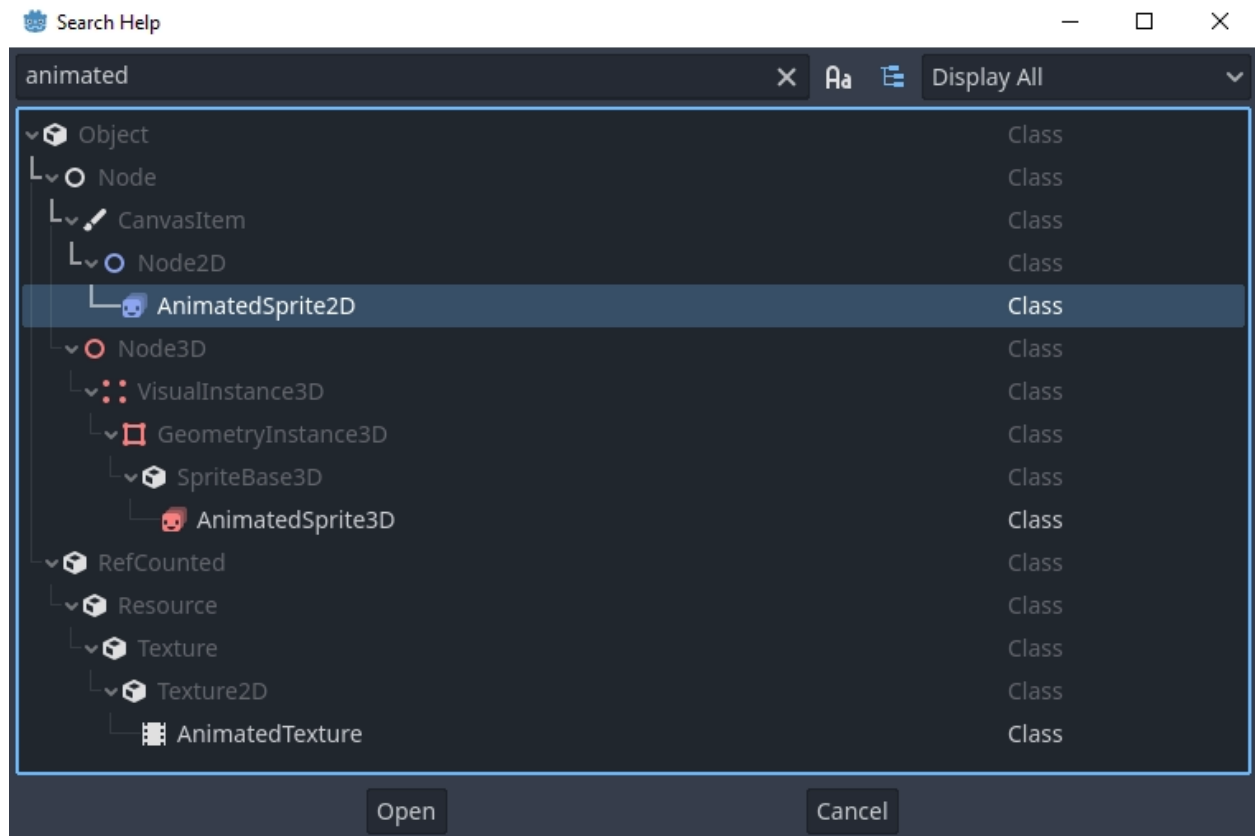
Godot comes with a built-in class reference.

You can search for information about a class, method, property, constant, or signal by any one of the following methods:

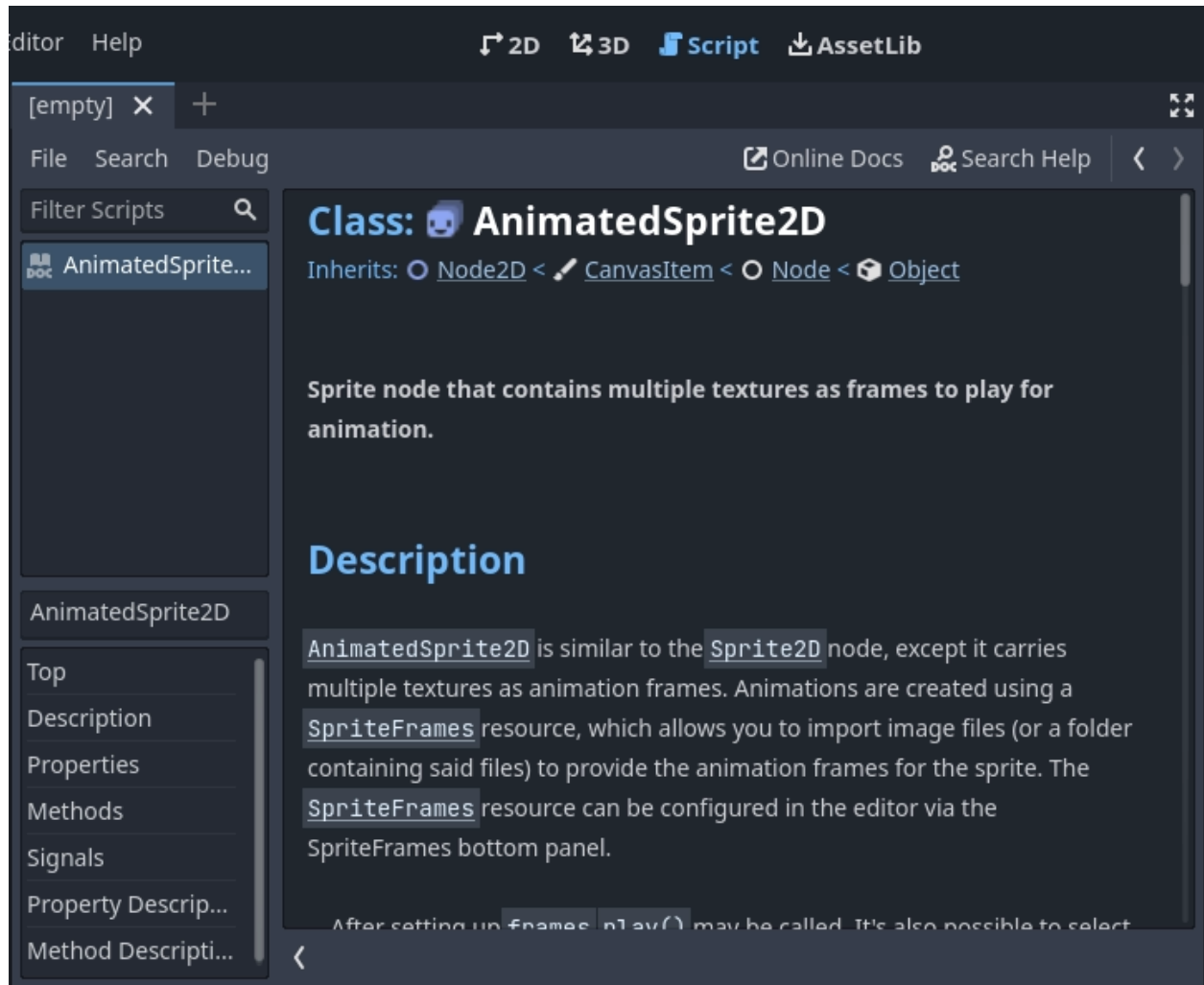
- Pressing F1 (or Alt + Space on macOS, or fn + F1 for laptops with a fn key) anywhere in the editor.
- Clicking the "Search Help" button in the top-right of the Script main screen.
- Clicking on the Help menu and Search Help.
- Clicking while pressing the Ctrl key on a class name, function name, or built-in variable in the script editor.



When you do any of these, a window pops up. Type to search for any item. You can also use it to browse available objects and methods.



Double-click on an item to open the corresponding page in the script main screen.



2.7.4 Learning new features

Godot is a feature-rich game engine. There is a lot to learn about it. This page explains how you can use the online manual, built-in code reference, and join online communities to learn new features and techniques.

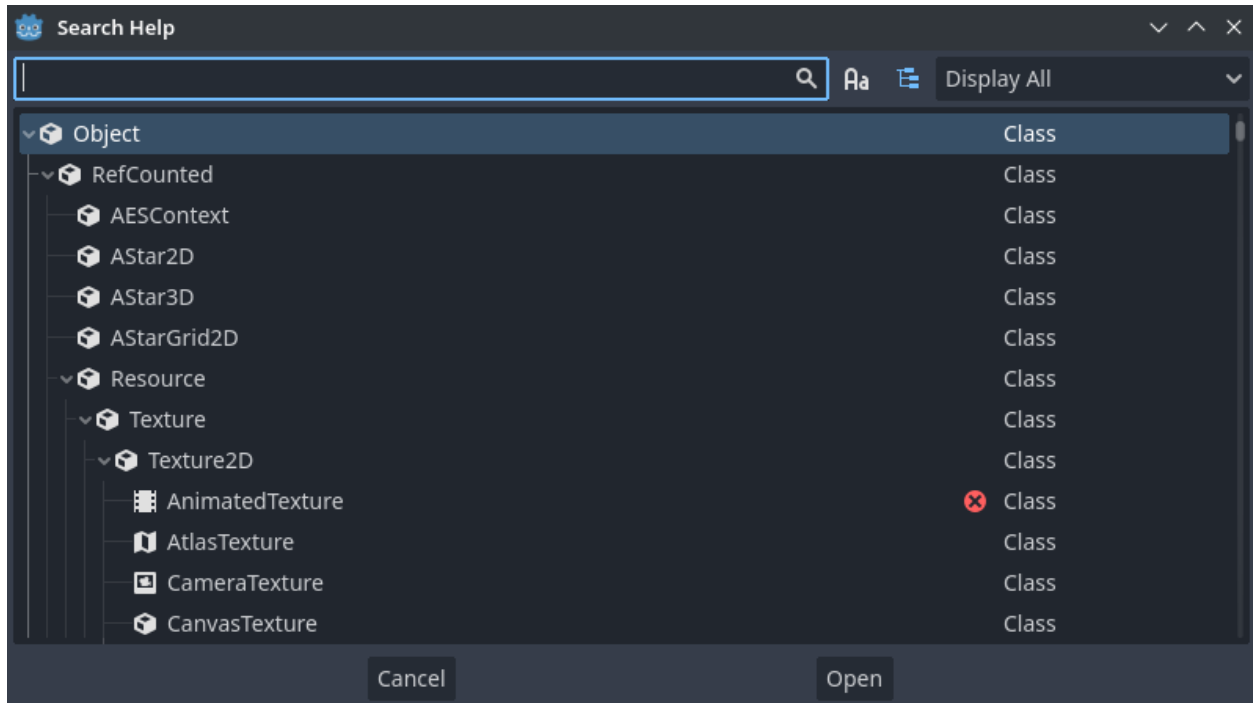
Making the most of this manual

What you are reading now is the user manual. It documents each of the engine's concepts and available features. When learning a new topic, you can start by browsing the corresponding section of this website. The left menu allows you to explore broad topics while the search bar will help you find more specific pages. If a page exists for a given theme, it will often link to more related content.



The manual has a companion class reference that explains each Godot class's available functions and properties when programming. While the manual covers general features, concepts, and how to use the editor, the reference is all about using Godot's scripting API (Application Programming Interface). You can access

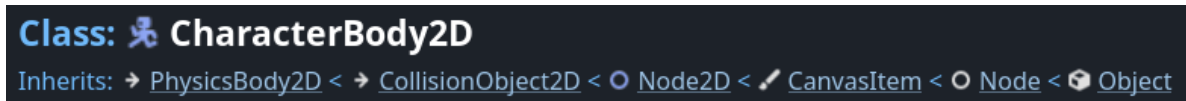
it both online and offline. We recommend browsing the reference offline, from within the Godot editor. To do so, go to Help -> Search Help or press F1.



To browse it online, head to the manual's Class Reference section.

A class reference's page tells you:

1. Where the class exists in the inheritance hierarchy. You can click the top links to jump to parent classes and see the properties and methods a type inherits.



2. A summary of the class's role and use cases.
3. An explanation of the class's properties, methods, signals, enums, and constants.
4. Links to manual pages further detailing the class.

Note: If the manual or class reference is missing or has insufficient information, please open an Issue in the official [godot-docs](#) GitHub repository to report it.

You can Ctrl-click any underlined text like the name of a class, property, method, signal, or constant to jump to it.

Learning to think like a programmer

Teaching programming foundations and how to think like a game developer is beyond the scope of Godot's documentation. If you're new to programming, we recommend two excellent free resources to get you started:

1. Harvard university offers a free courseware to learn to program, [CS50](#). It will teach you programming fundamentals, how code works, and how to think like a programmer. These skills are essential to become a game developer and learn any game engine efficiently. You can see this course as an investment that will save you time and trouble when you learn to create games.
2. If you prefer books, check out the free ebook [Automate The Boring Stuff With Python](#) by Al Sweigart.

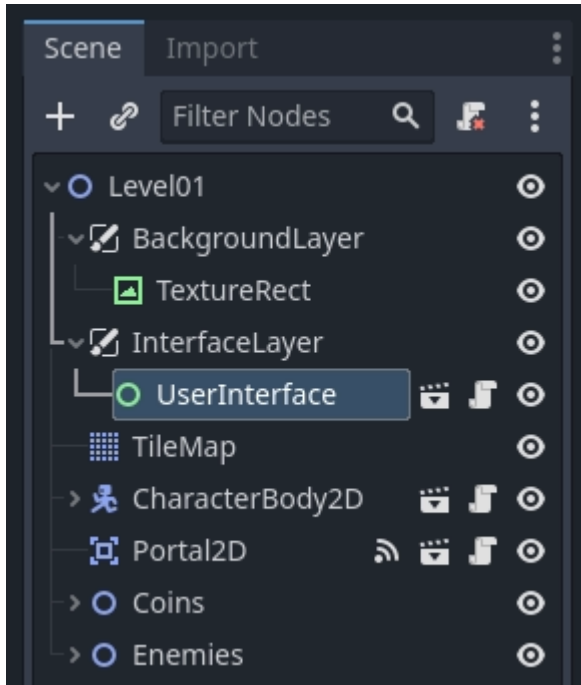
Learning with the community

Godot has a growing community of users. If you're stuck on a problem or need help to better understand how to achieve something, you can ask other users for help on one of the many [active communities](#).

The best place to ask questions and find already answered ones is the official [Questions & Answers](#) site. These responses show up in search engine results and get saved, allowing other users to benefit from discussions on the platform. Once you have asked a question there, you can share its link on other social platforms. Before asking a question, be sure to look for existing answers that might solve your problem on this website or using your preferred search engine.

Asking questions well and providing details will help others answer you faster and better. When asking questions, we recommend including the following information:

1. Describe your goal. You want to explain what you are trying to achieve design-wise. If you are having trouble figuring out how to make a solution work, there may be a different, easier solution that accomplishes the same goal.
2. If there is an error involved, share the exact error message. You can copy the exact error message in the editor's Debugger bottom panel by clicking the Copy Error icon. Knowing what it says can help community members better identify how you triggered the error.
3. If there is code involved, share a code sample. Other users won't be able to help you fix a problem without seeing your code. Share the code as text directly. To do so, you can copy and paste a short code snippet in a chat box, or use a website like [Pastebin](#) to share long files.
4. Share a screenshot of your Scene dock along with your written code. Most of the code you write affects nodes in your scenes. As a result, you should think of those scenes as part of your source code.



Also, please don't take a picture with your phone, the low quality and screen reflections can make it hard to understand the image. Your operating system should have a built-in tool to take screenshots with the `PrtSc` (Print Screen) key.

Alternatively, you can use a program like [ShareX](#) on Windows or [FlameShot](#) on Linux.

5. Sharing a video of your running game can also be really useful to troubleshoot your game. You can use programs like [OBS Studio](#) and [Screen to GIF](#) to capture your screen.

You can then use a service like [streamable](#) or a cloud provider to upload and share your videos for free.

6. If you're not using the stable version of Godot, please mention the version you're using. The answer can be different as available features and the interface evolve rapidly.

Following these guidelines will maximize your chances of getting the answer you're looking for. They will save time both for you and the persons helping you.

Community tutorials

This manual aims to provide a comprehensive reference of Godot's features. Aside from the 2D and 3D getting started series, it does not contain tutorials to implement specific game genres. If you're looking for a tutorial about creating a role-playing game, a platformer, or other, please see [Tutorials and resources](#), which lists content made by the Godot community.

2.7.5 Godot's design philosophy

Now that you've gotten your feet wet, let's talk about Godot's design.

Every game engine is different and fits different needs. Not only do they offer a range of features, but the design of each engine is unique. This leads to different workflows and different ways to form your games' structures. This all stems from their respective design philosophies.

This page is here to help you understand how Godot works, starting with some of its core pillars. It is not a list of available features, nor is it an engine comparison. To know if any engine can be a good fit for your project, you need to try it out for yourself and understand its design and limitations.

Please watch [Godot explained in 5 minutes](#) if you're looking for an overview of the engine's features.

Object-oriented design and composition

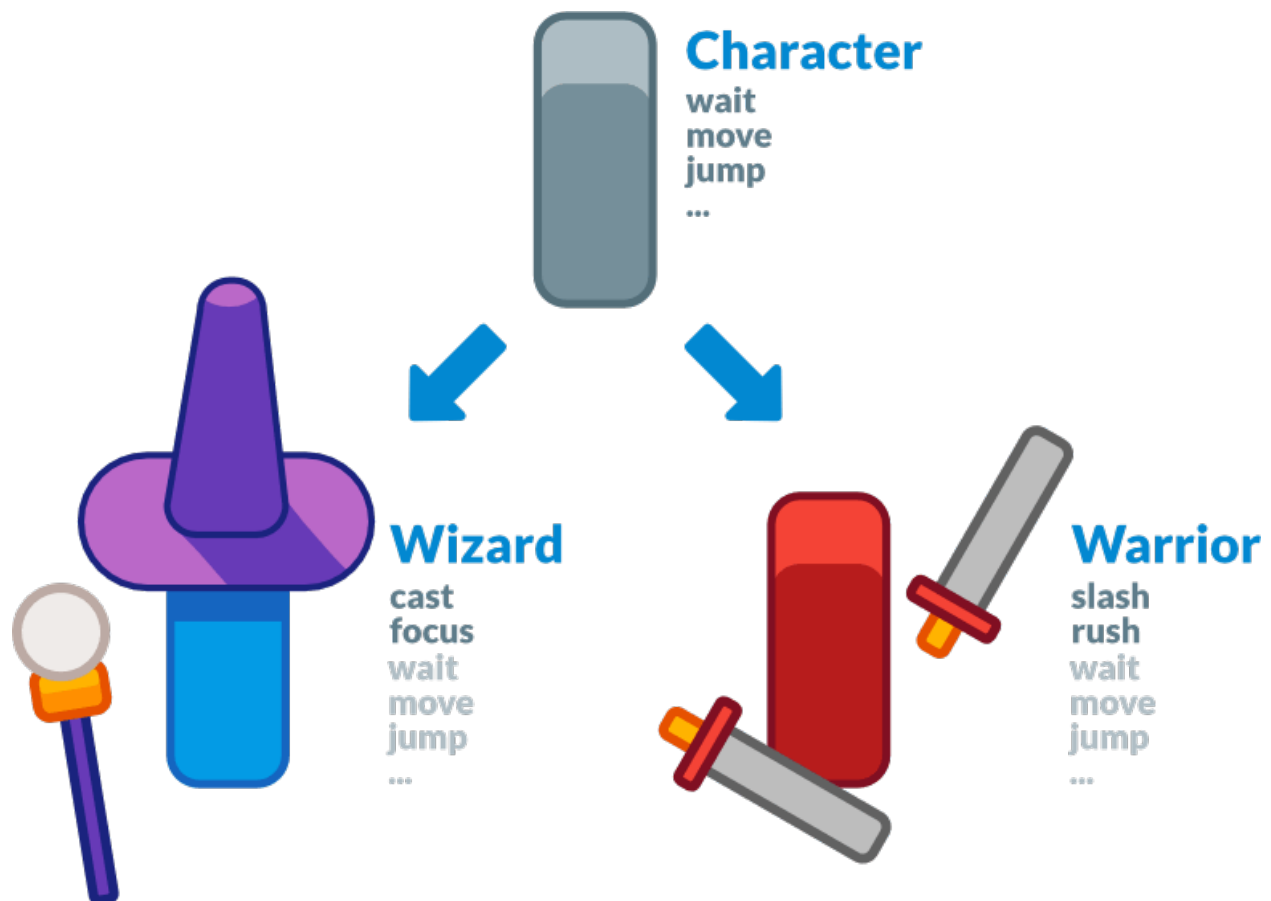
Godot embraces object-oriented design at its core with its flexible scene system and Node hierarchy. It tries to stay away from strict programming patterns to offer an intuitive way to structure your game.

For one, Godot lets you compose or aggregate scenes. It's like nested prefabs: you can create a `BlinkingLight` scene and a `BrokenLantern` scene that uses the `BlinkingLight`. Then, create a city filled with `BrokenLanterns`. Change the `BlinkingLight`'s color, save, and all the `BrokenLanterns` in the city will update instantly.

On top of that, you can inherit from any scene.

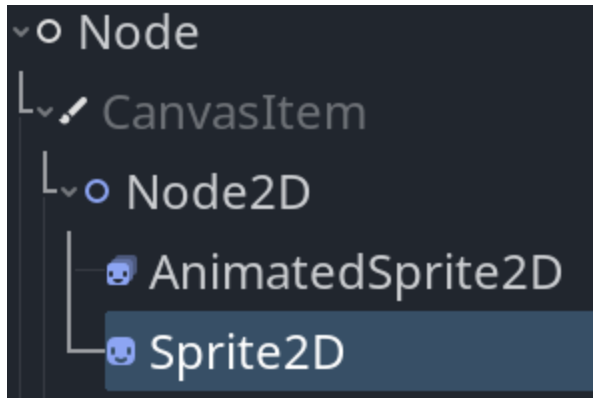
A Godot scene could be a `Weapon`, a `Character`, an `Item`, a `Door`, a `Level`, part of a level... anything you'd like. It works like a class in pure code, except you're free to design it by using the editor, using only the code, or mixing and matching the two.

It's different from prefabs you find in several 3D engines, as you can then inherit from and extend those scenes. You may create a `Magician` that extends your `Character`. Modify the `Character` in the editor and the `Magician` will update as well. It helps you build your projects so that their structure matches the game's design.



Also note that Godot offers many different types of objects called nodes, each with a specific purpose. Nodes are part of a tree and always inherit from their parents up to the `Node` class. Although the engine does feature some nodes like collision shapes that a parent physics body will use, most nodes work independently from one another.

In other words, Godot's nodes do not work like components in some other game engines.



Sprite2D is a Node2D, a CanvasItem and a Node. It has all the properties and features of its three parent classes, like transforms or the ability to draw custom shapes and render with a custom shader.

All-inclusive package

Godot tries to provide its own tools to answer most common needs. It has a dedicated scripting workspace, an animation editor, a tilemap editor, a shader editor, a debugger, a profiler, the ability to hot-reload locally and on remote devices, etc.



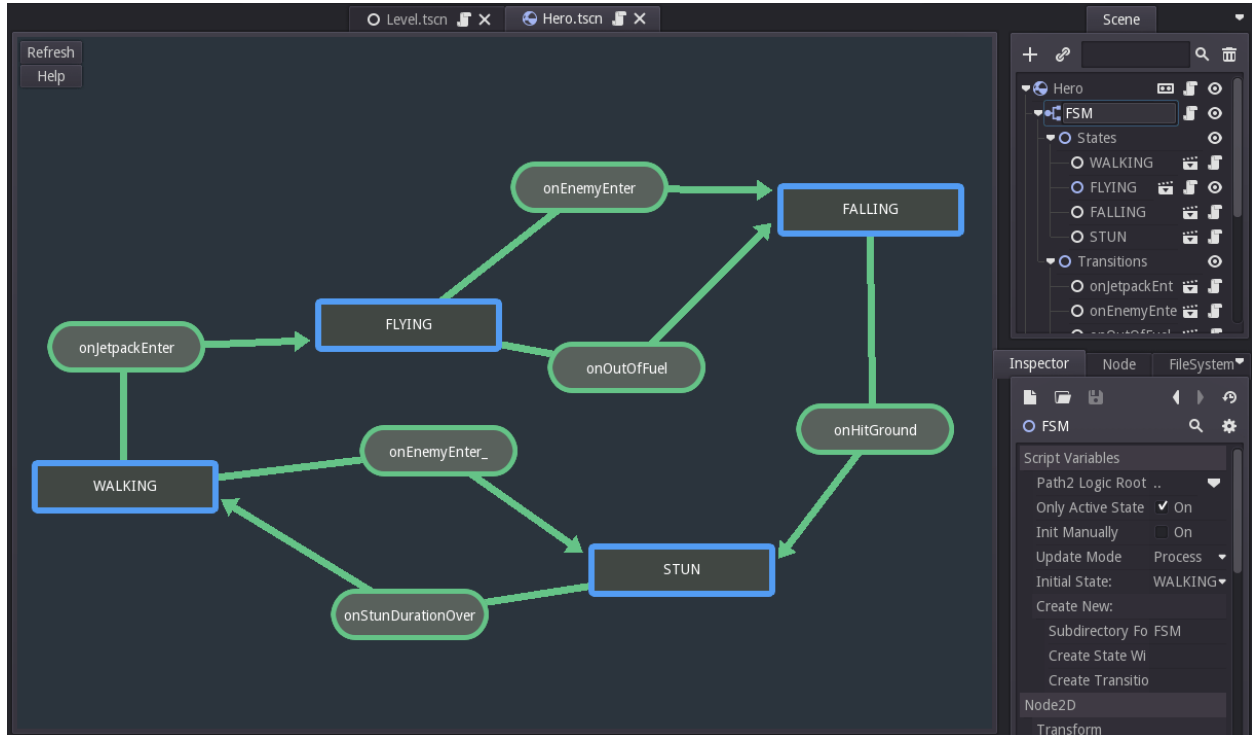
The goal is to offer a full package to create games and a continuous user experience. You can still work with external programs as long as there is an import plugin available in Godot for it. Or you can create one, like the [Tiled Map Importer](#).

That is also partly why Godot offers its own programming language GDScript along with C#. GDScript is designed for the needs of game developers and game designers, and is tightly integrated in the engine and the editor.

GDScript lets you write code using an indentation-based syntax, yet it detects types and offers a static language's quality of auto-completion. It is also optimized for gameplay code with built-in types like Vectors and Colors.

Note that with GDEXTENSION, you can write high-performance code using compiled languages like C, C++, Rust, D, Haxe, or Swift without recompiling the engine.

Note that the 3D workspace doesn't feature as many tools as the 2D workspace. You'll need external programs or add-ons to edit terrains, animate complex characters, and so on. Godot provides a complete API to extend the editor's functionality using game code. See [The Godot editor is a Godot game](#) below.



A State Machine editor plugin in Godot 2 by kubecz3k. It lets you manage states and transitions visually.

Open source

Godot offers a fully open source codebase under the MIT license. This means all the technologies that ship with it have to be Free (as in freedom) as well. For the most part, they're developed from the ground up by contributors.

Anyone can plug in proprietary tools for the needs of their projects — they just won't ship with the engine. This may include Google AdMob, or FMOD. Any of these can come as third-party plugins instead.

On the other hand, an open codebase means you can learn from and extend the engine to your heart's content. You can also debug games easily, as Godot will print errors with a stack trace, even if they come from the engine itself.

Note: This does not affect the work you do with Godot in any way: there's no strings attached to the engine or anything you make with it.

Community-driven

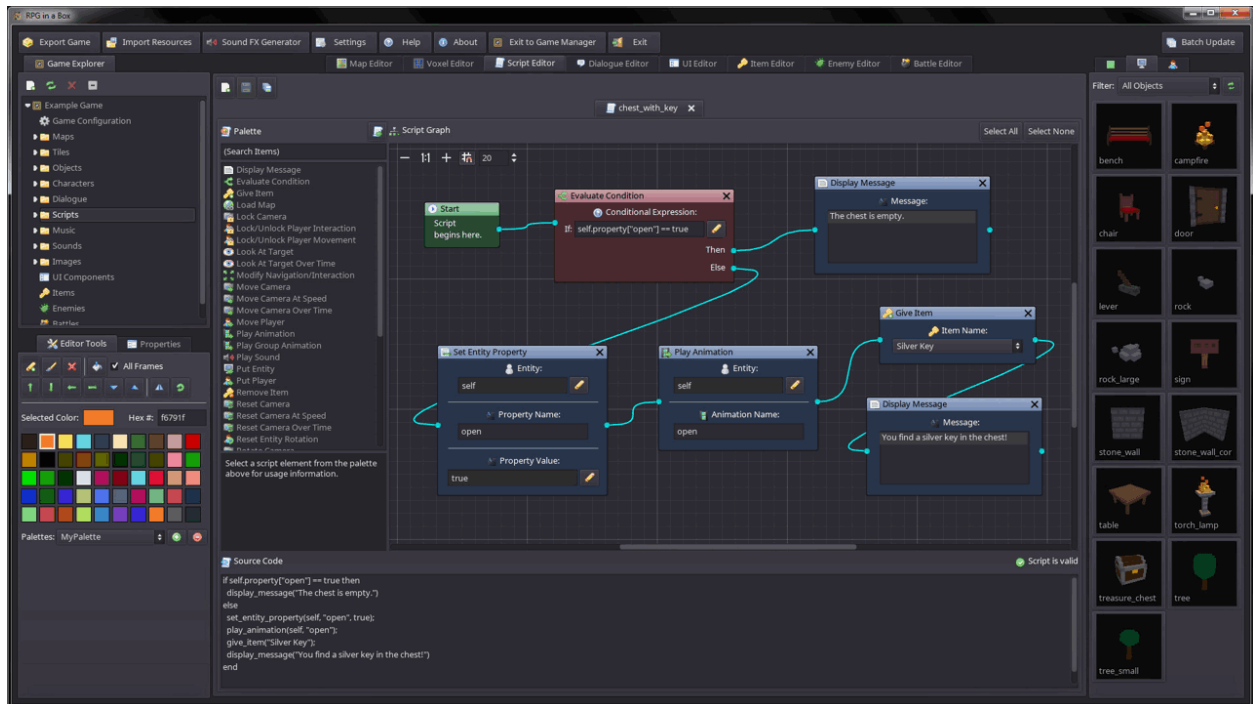
Godot is made by its community, for the community, and for all game creators out there. It's the needs of the users and open discussions that drive the core updates. New features from the core developers often focus on what will benefit the most users first.

That said, although a handful of core developers work on it full-time, the project has over 600 contributors at the time of writing. Benevolent programmers work on features they may need themselves, so you'll see improvements in all corners of the engine at the same time in every major release.

The Godot editor is a Godot game

The Godot editor runs on the game engine. It uses the engine's own UI system, it can hot-reload code and scenes when you test your projects, or run game code in the editor. This means you can use the same code and scenes for your games, or build plugins and extend the editor.

This leads to a reliable and flexible UI system, as it powers the editor itself. With the `@tool` annotation, you can run any game code in the editor.



RPG in a Box is a voxel RPG editor made with Godot 2. It uses Godot's UI tools for its node-based programming system and for the rest of the interface.

Put the `@tool` annotation at the top of any GDScript file and it will run in the editor. This lets you import and export plugins, create plugins like custom level editors, or create scripts with the same nodes and API you use in your projects.

Note: The editor is fully written in C++ and is statically compiled into the binary. This means you can't import it as a typical project that would have a `project.godot` file.

Separate 2D and 3D engines

Godot offers dedicated 2D and 3D rendering engines. As a result, the base unit for 2D scenes is pixels. Even though the engines are separate, you can render 2D in 3D, 3D in 2D, and overlay 2D sprites and interfaces over your 3D world.

2.8 Step by step

This series builds upon the Introduction to Godot and will get you started with the editor and the engine. You will learn more about nodes and scenes, code your first classes with GDScript, use signals to make nodes communicate with one another, and more.

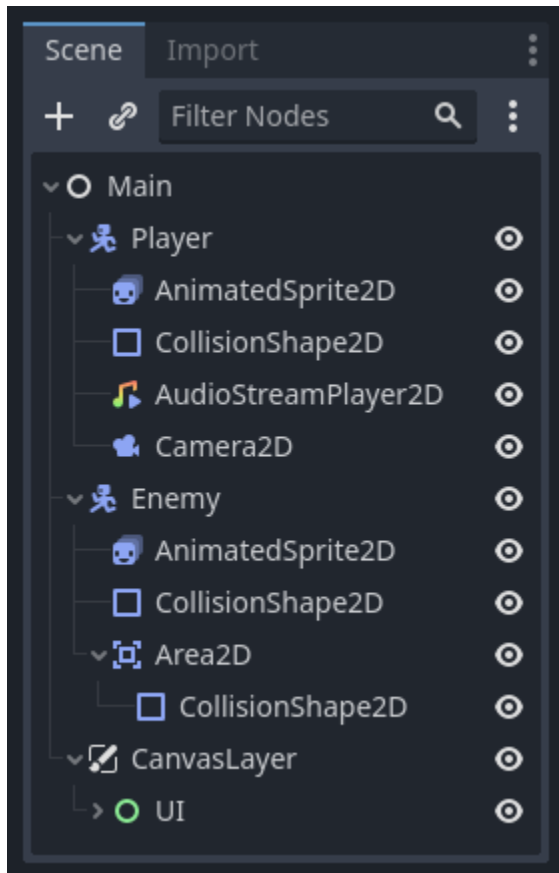
The following lessons are here to prepare you for Your first 2D game, a step-by-step tutorial where you will code a game from scratch. By the end of it, you will have the necessary foundations to explore more features in other sections. We also included links to pages that cover a given topic in-depth where appropriate.

2.8.1 Nodes and Scenes

In Overview of Godot's key concepts, we saw that a Godot game is a tree of scenes and that each scene is a tree of nodes. In this lesson, we explain a bit more about them. You will also create your first scene.

Nodes

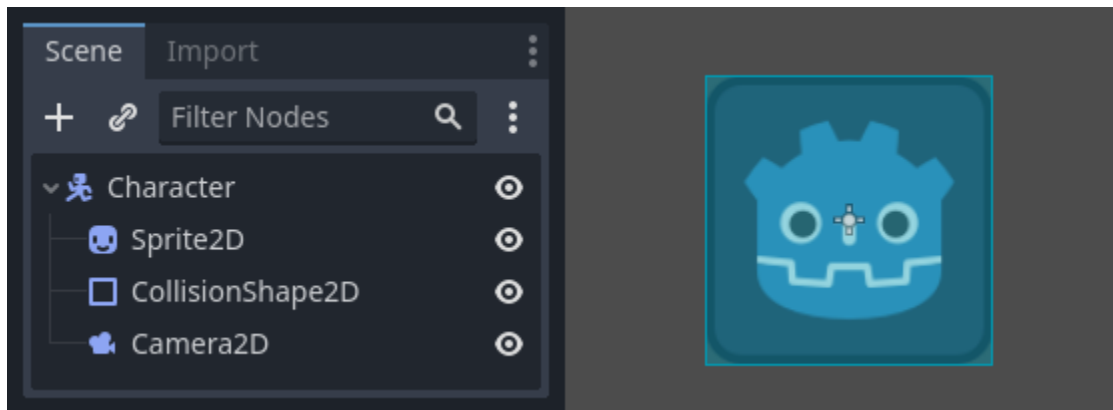
Nodes are the fundamental building blocks of your game. They are like the ingredients in a recipe. There are dozens of kinds that can display an image, play a sound, represent a camera, and much more.



All nodes have the following characteristics:

- A name.
- Editable properties.
- They receive callbacks to update every frame.
- You can extend them with new properties and functions.
- You can add them to another node as a child.

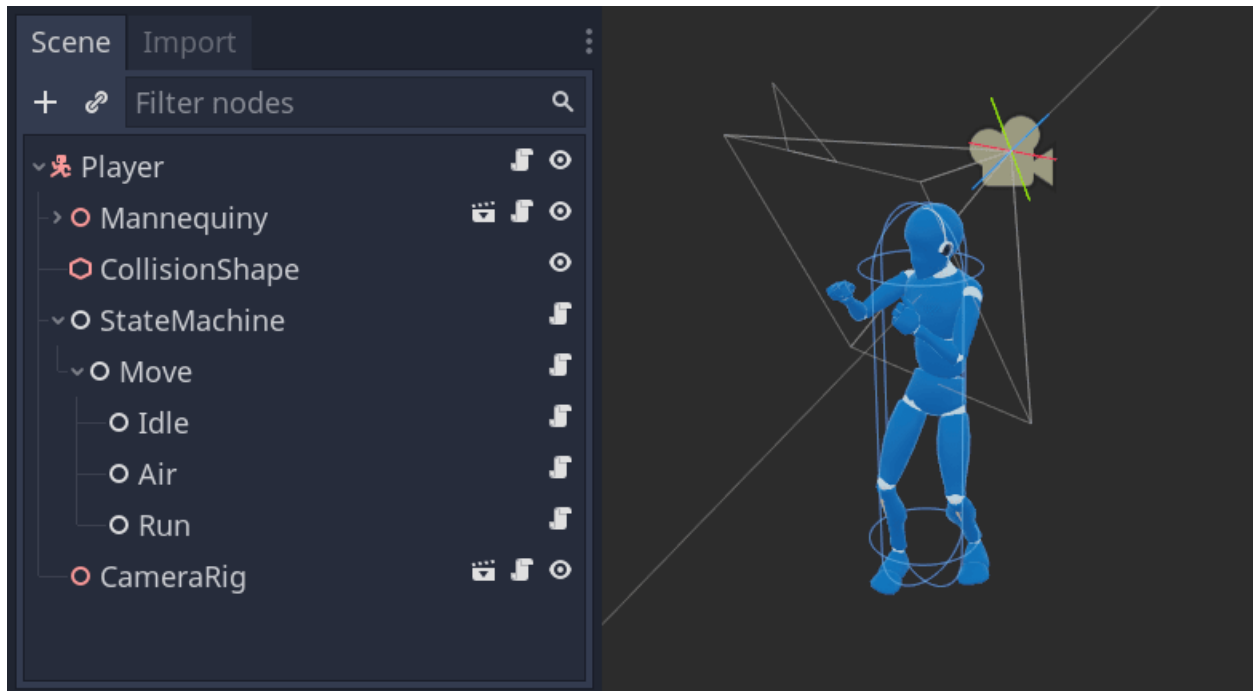
The last characteristic is important. Together, nodes form a tree, which is a powerful feature to organize projects. Since different nodes have different functions, combining them produces more complex behavior. As we saw before, you can build a playable character the camera follows using a `CharacterBody2D` node, a `Sprite2D` node, a `Camera2D` node, and a `CollisionShape2D` node.



Scenes

When you organize nodes in a tree, like our character, we call this construct a scene. Once saved, scenes work like new node types in the editor, where you can add them as a child of an existing node. In that case, the instance of the scene appears as a single node with its internals hidden.

Scenes allow you to structure your game's code however you want. You can compose nodes to create custom and complex node types, like a game character that runs and jumps, a life bar, a chest with which you can interact, and more.



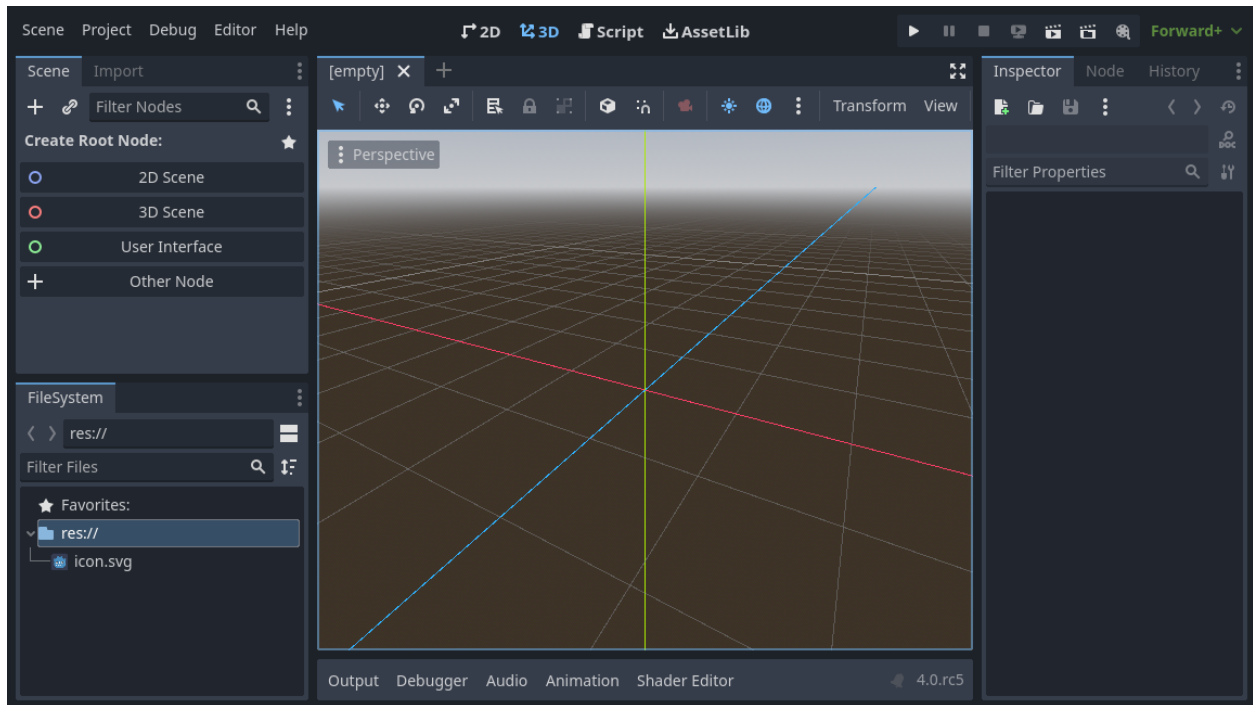
The Godot editor essentially is a scene editor. It has plenty of tools for editing 2D and 3D scenes, as well as user interfaces. A Godot project can contain as many of these scenes as you need. The engine only requires one as your application's main scene. This is the scene Godot will first load when you or a player runs the game.

On top of acting like nodes, scenes have the following characteristics:

1. They always have one root node, like the "Character" in our example.
2. You can save them to your local drive and load them later.
3. You can create as many instances of a scene as you'd like. You could have five or ten characters in your game, created from your Character scene.

Creating your first scene

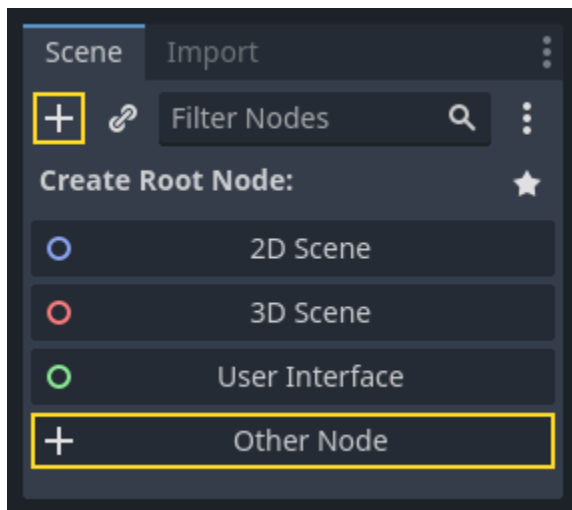
Let's create our first scene with a single node. To do so, you will need to create a new project first. After opening the project, you should see an empty editor.



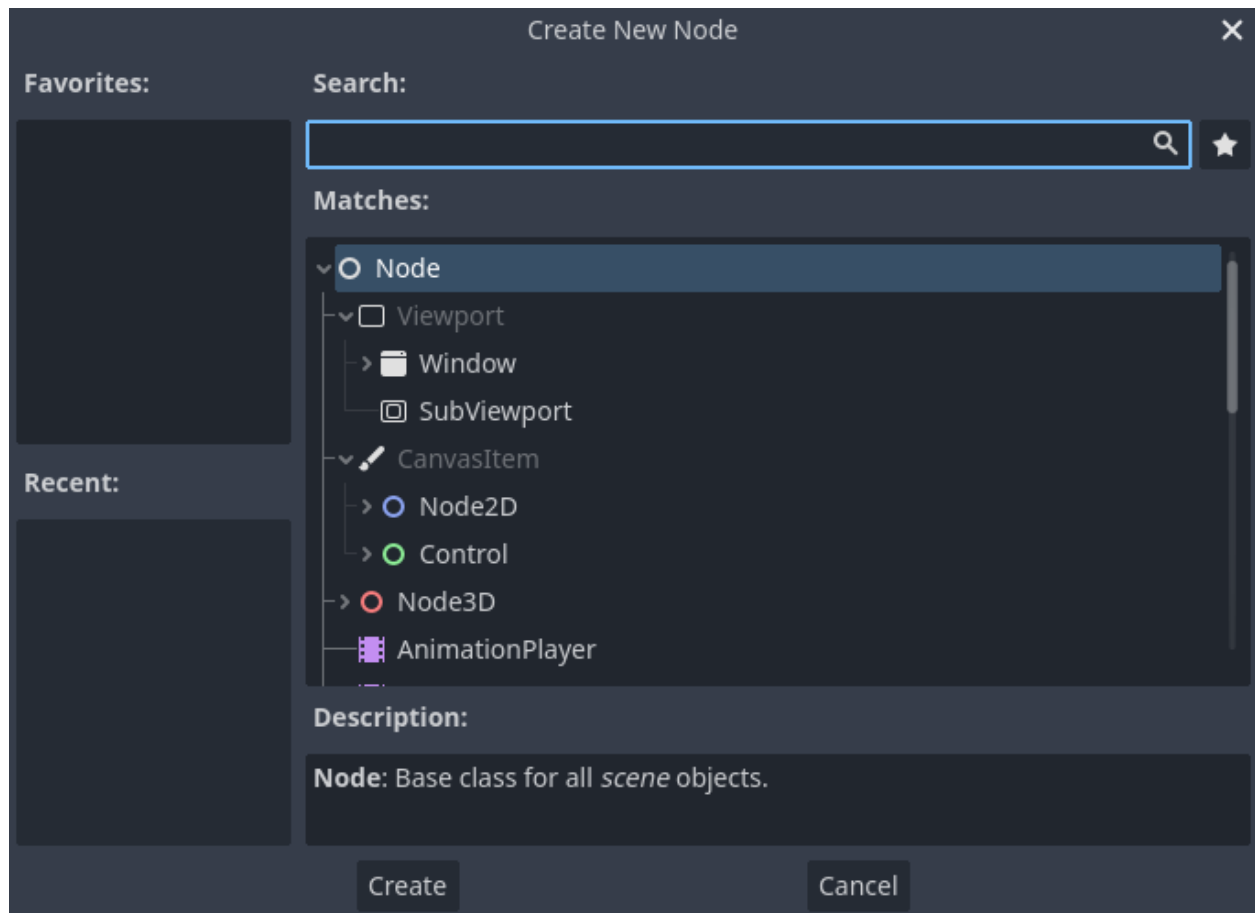
In an empty scene, the Scene dock on the left shows several options to add a root node quickly. "2D Scene" adds a Node2D node, "3D Scene" adds a Node3D node, and "User Interface" adds a Control node. These presets are here for convenience; they are not mandatory. "Other Node" lets you select any node to be the root node. In an empty scene, "Other Node" is equivalent to pressing the "Add Child Node" button at the top-left of the Scene dock, which usually adds a new node as a child of the currently selected node.

We're going to add a single Label node to our scene. Its function is to draw text on the screen.

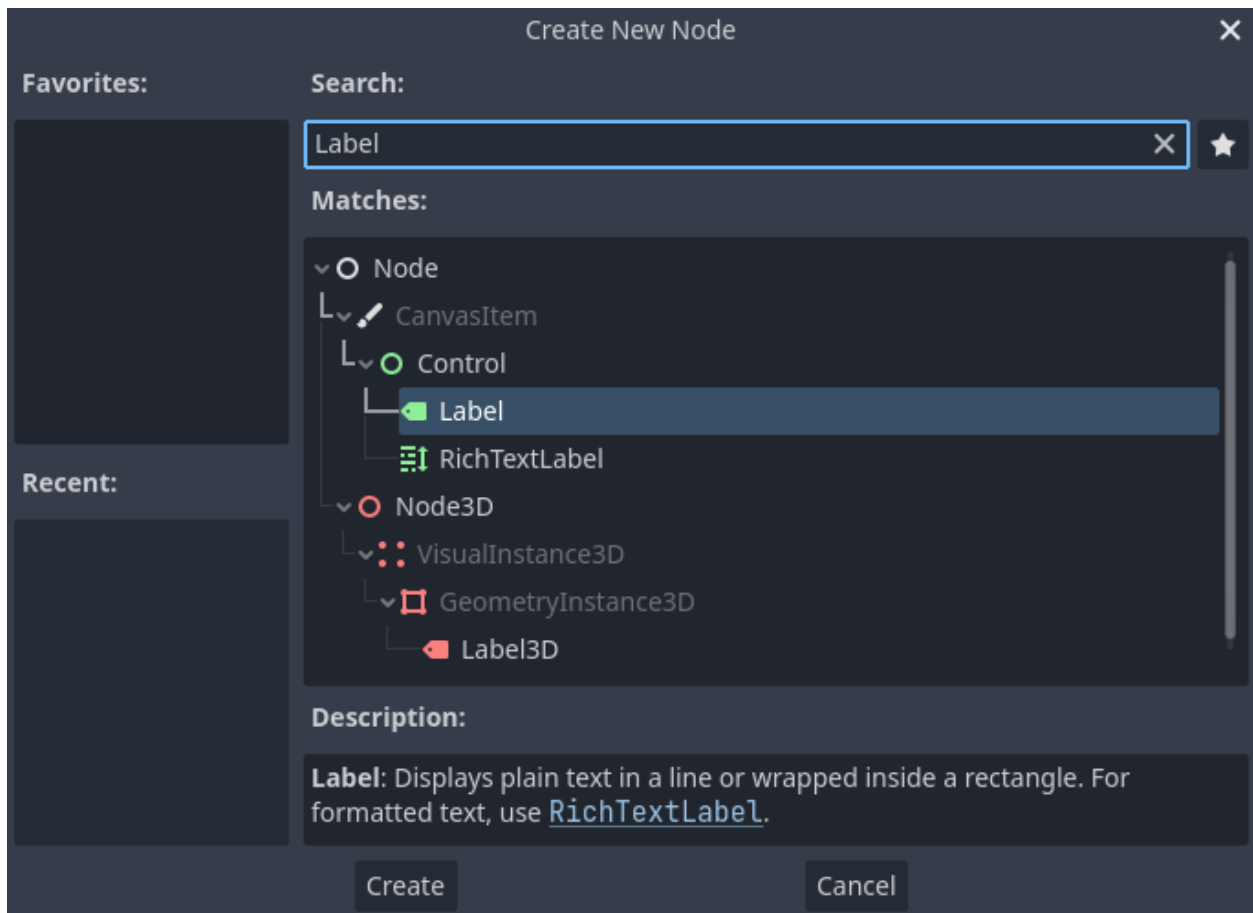
Press the "Add Child Node" button or "Other Node" to create a root node.



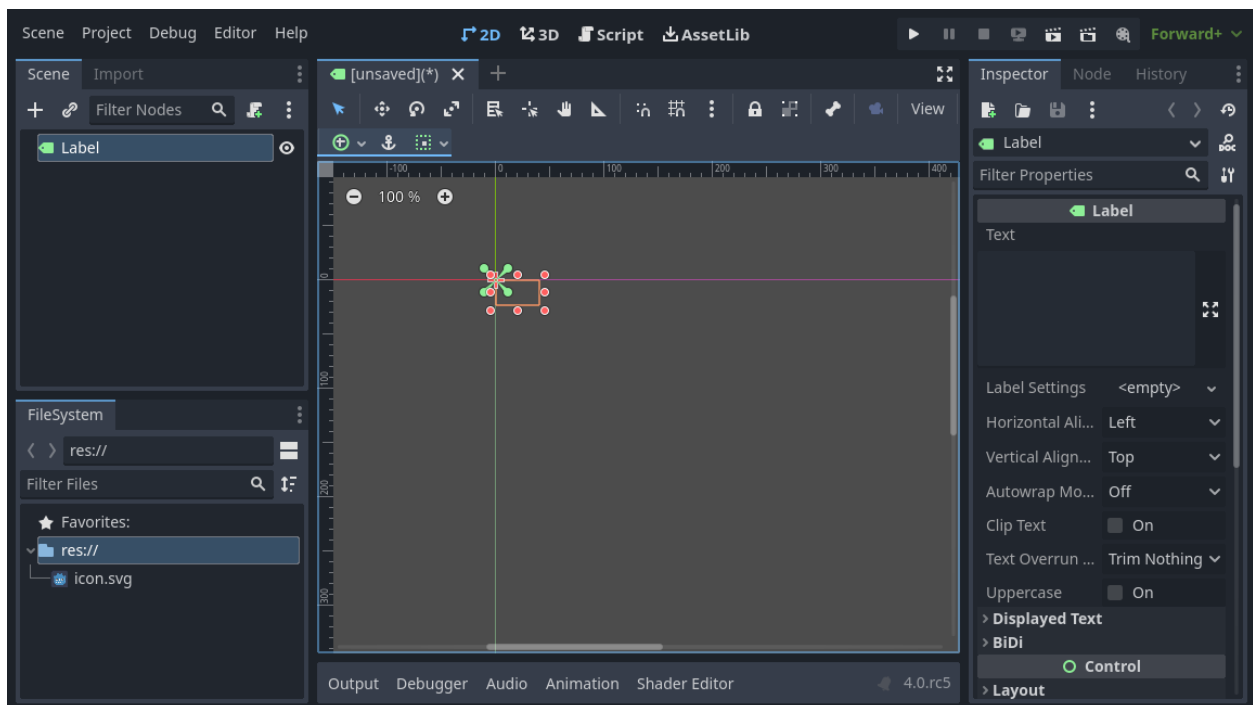
The Create Node dialog opens, showing the long list of available nodes.



Select the Label node. You can type its name to filter down the list.



Click on the Label node to select it and click the Create button at the bottom of the window.



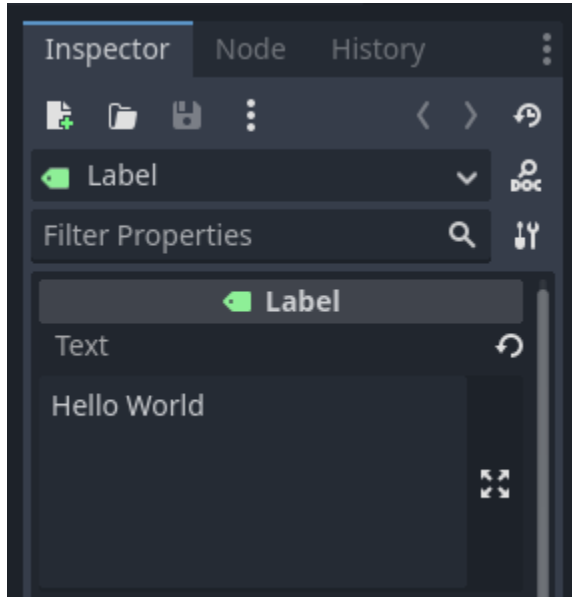
A lot happens when you add a scene's first node. The scene changes to the 2D workspace because Label is

a 2D node type. The Label appears, selected, in the top-left corner of the viewport. The node appears in the Scene dock on the left, and the node's properties appear in the Inspector dock on the right.

Changing a node's properties

The next step is to change the Label's "Text" property. Let's change it to "Hello World".

Head to the Inspector dock on the right of the viewport. Click inside the field below the Text property and type "Hello World".



You will see the text draw in the viewport as you type.

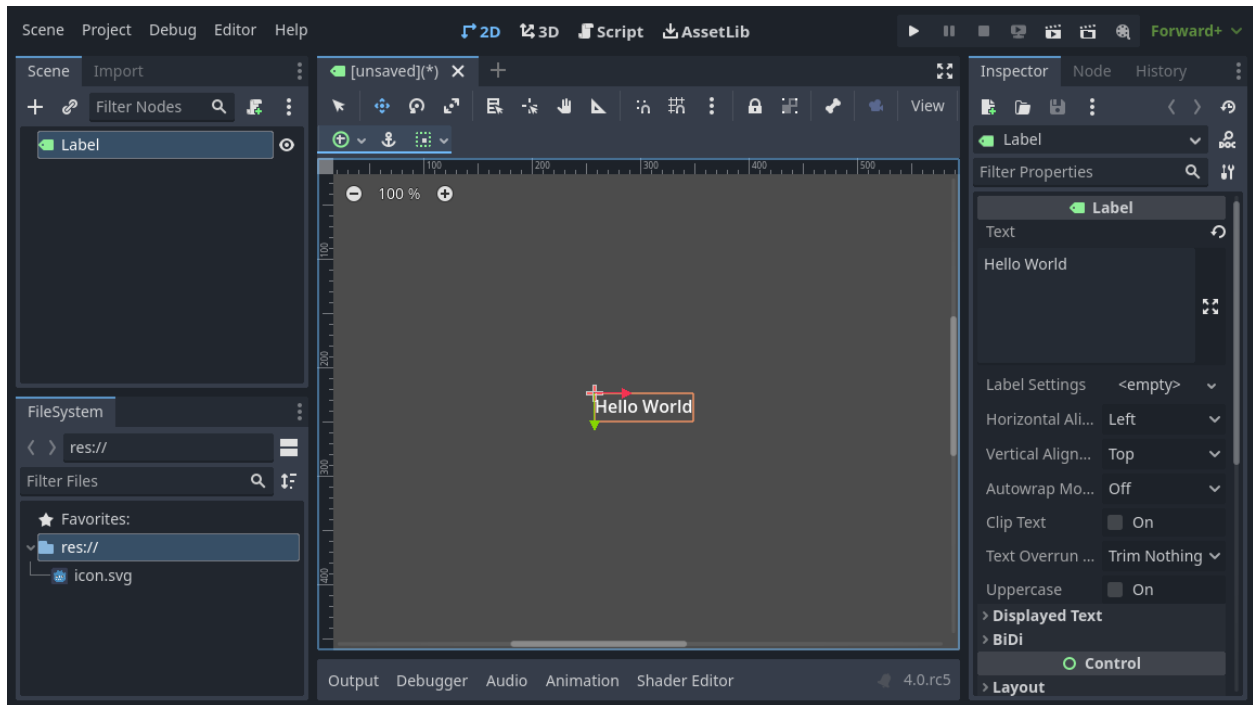
See also:

You can edit any property listed in the Inspector as we did with the Text. For a complete reference of the Inspector dock, see [The Inspector](#).

You can move your Label node in the viewport by selecting the move tool in the toolbar.

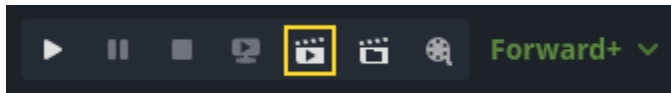


With the Label selected, click and drag anywhere in the viewport to move it to the center of the view delimited by the rectangle.

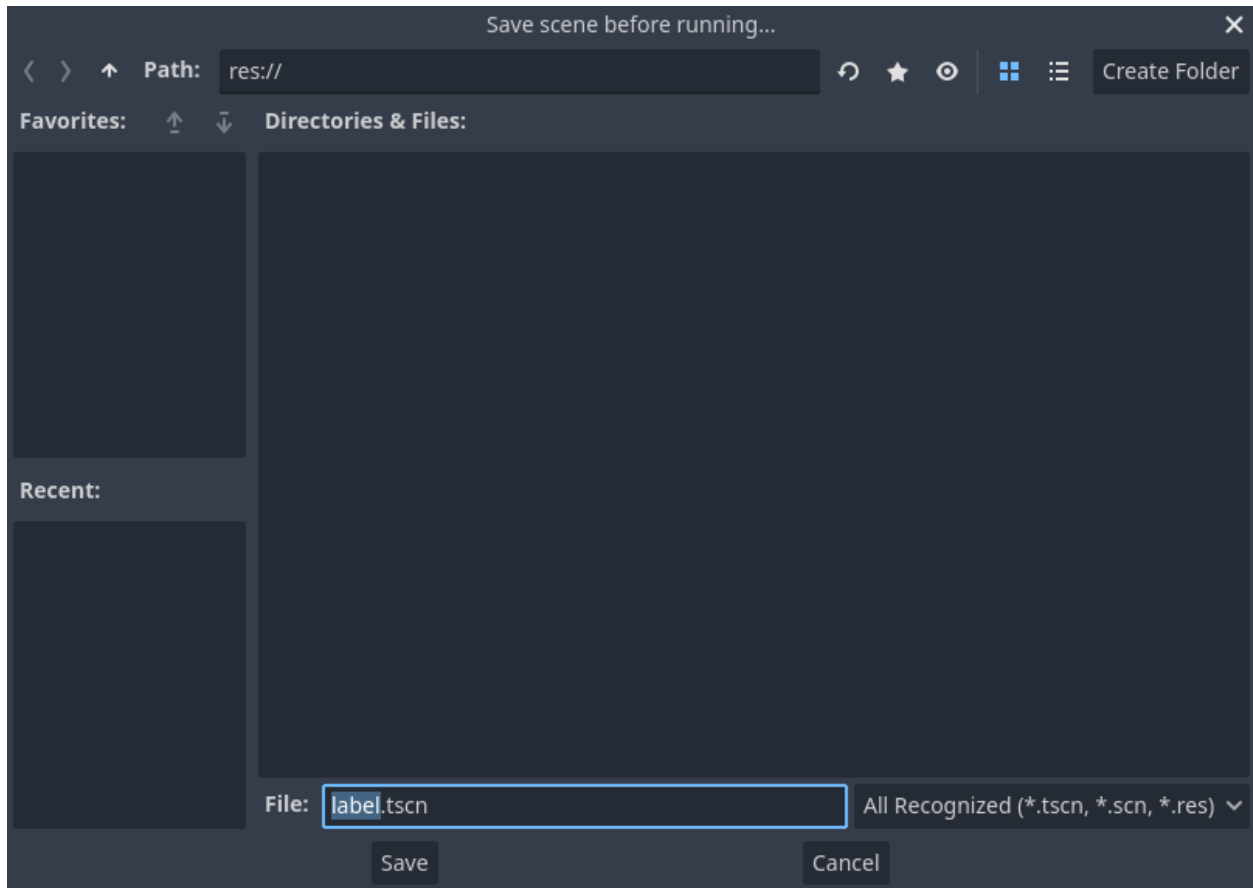


Running the scene

Everything's ready to run the scene! Press the Play Scene button in the top-right of the screen or press F6 (Cmd + R on macOS).



A popup invites you to save the scene, which is required to run it. Click the Save button in the file browser to save it as `label.tscn`.



Note: The Save Scene As dialog, like other file dialogs in the editor, only allows you to save files inside the project. The `res://` path at the top of the window represents the project's root directory and stands for "resource path". For more information about file paths in Godot, see [File system](#).

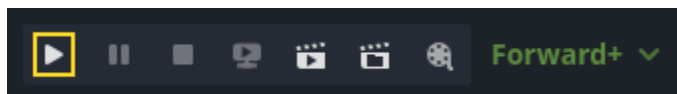
The application should open in a new window and display the text "Hello World".



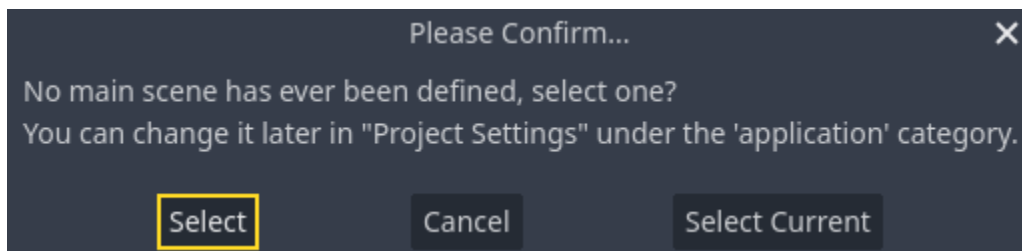
Close the window or press F8 (Cmd + . on macOS) to quit the running scene.

Setting the main scene

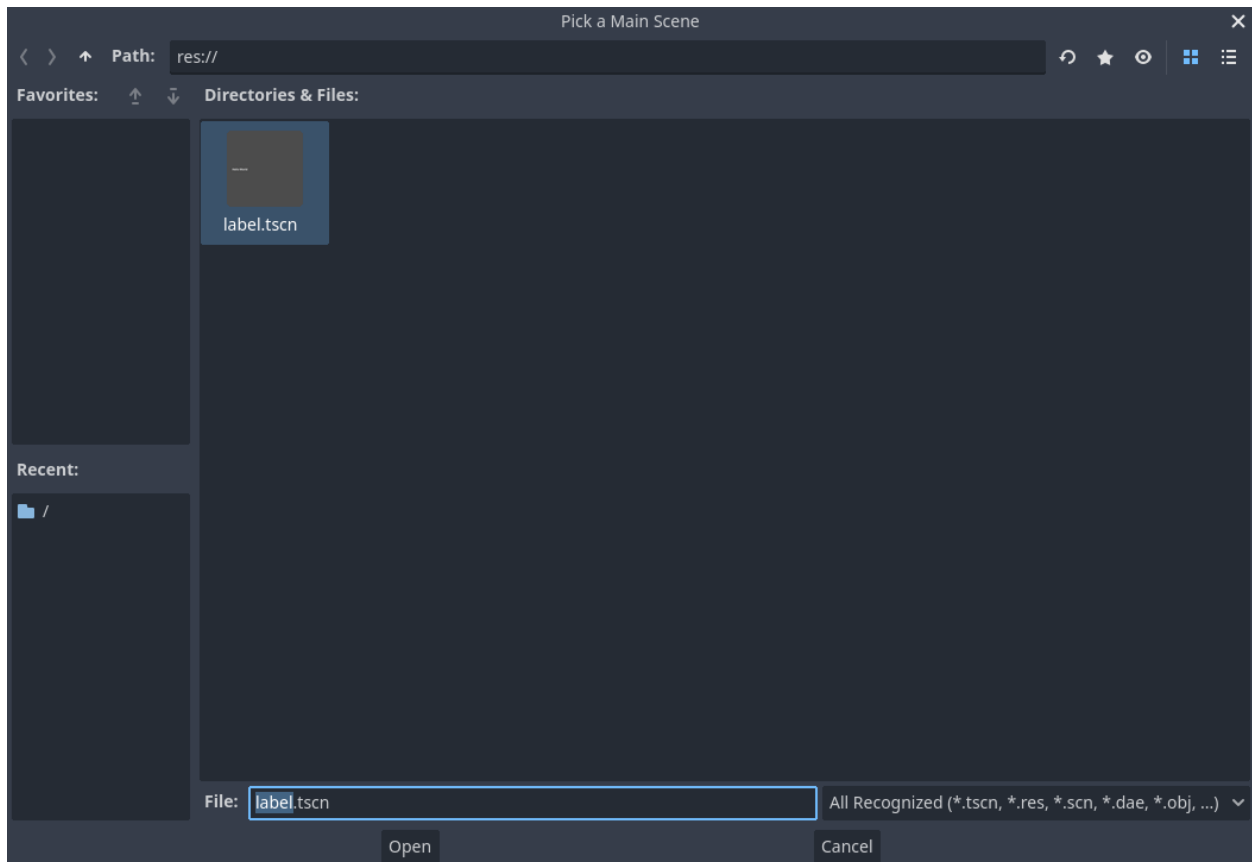
To run our test scene, we used the Play Scene button. Another button next to it allows you to set and run the project's main scene. You can press F5 (Cmd + B on macOS) to do so.



A popup window appears and invites you to select the main scene.



Click the Select button, and in the file dialog that appears, double click on label.tscn.



The demo should run again. Moving forward, every time you run the project, Godot will use this scene as a starting point.

Note: The editor saves the main scene's path in a `project.godot` file in your project's directory. While you can edit this text file directly to change project settings, you can also use the "Project -> Project Settings" window to do so. For more information, see [Project Settings](#).

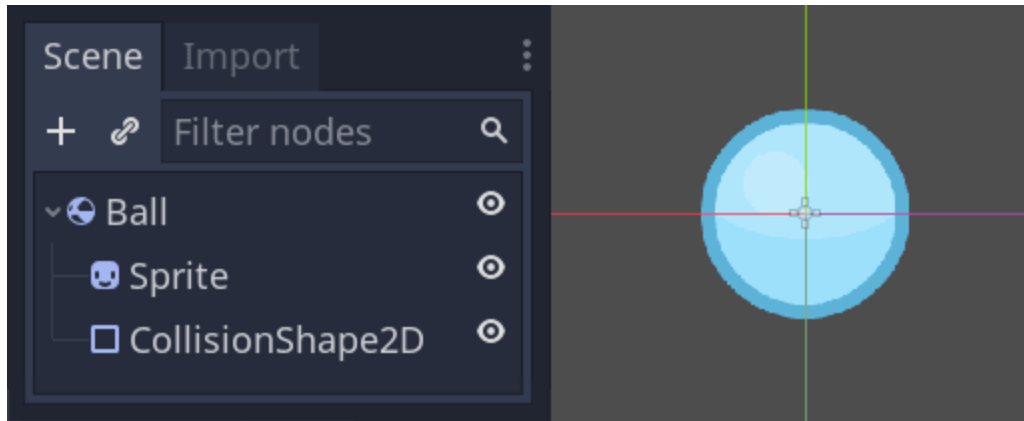
In the next part, we will discuss another key concept in games and in Godot: creating instances of a scene.

2.8.2 Creating instances

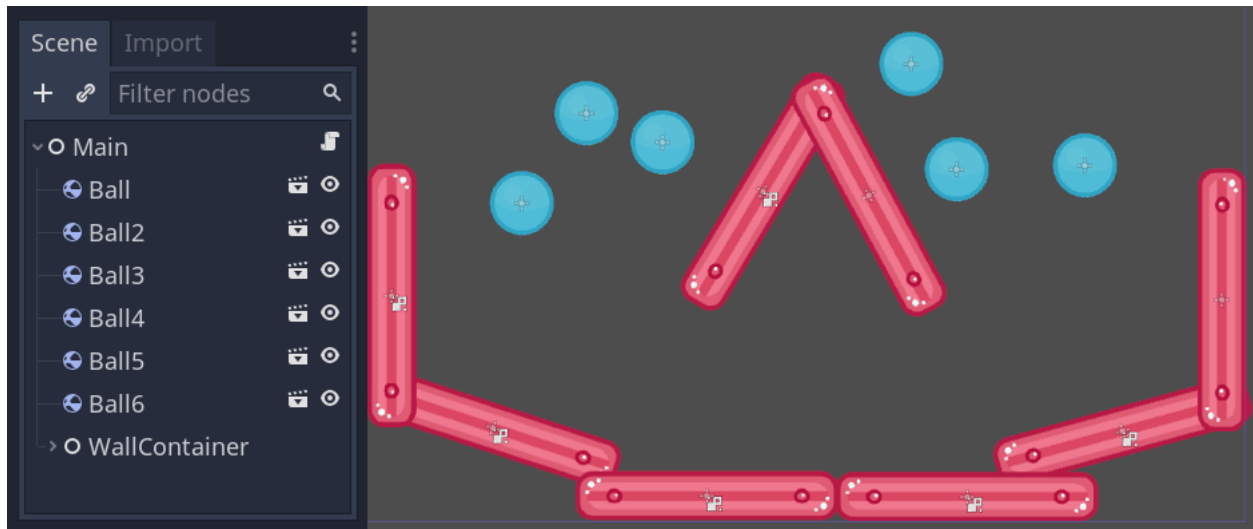
In the previous part, we saw that a scene is a collection of nodes organized in a tree structure, with a single node as its root. You can split your project into any number of scenes. This feature helps you break down and organize your game's different components.

You can create as many scenes as you'd like and save them as files with the `.tscn` extension, which stands for "text scene". The `label.tscn` file from the previous lesson was an example. We call those files "Packed Scenes" as they pack information about your scene's content.

Here's an example of a ball. It's composed of a `RigidBody2D` node as its root named `Ball`, which allows the ball to fall and bounce on walls, a `Sprite2D` node, and a `CollisionShape2D`.



Once you saved a scene, it works as a blueprint: you can reproduce it in other scenes as many times as you'd like. Replicating an object from a template like this is called **instancing**.



As we mentioned in the previous part, instanced scenes behave like a node: the editor hides their content by default. When you instance the Ball, you only see the Ball node. Notice also how each duplicate has a unique name.

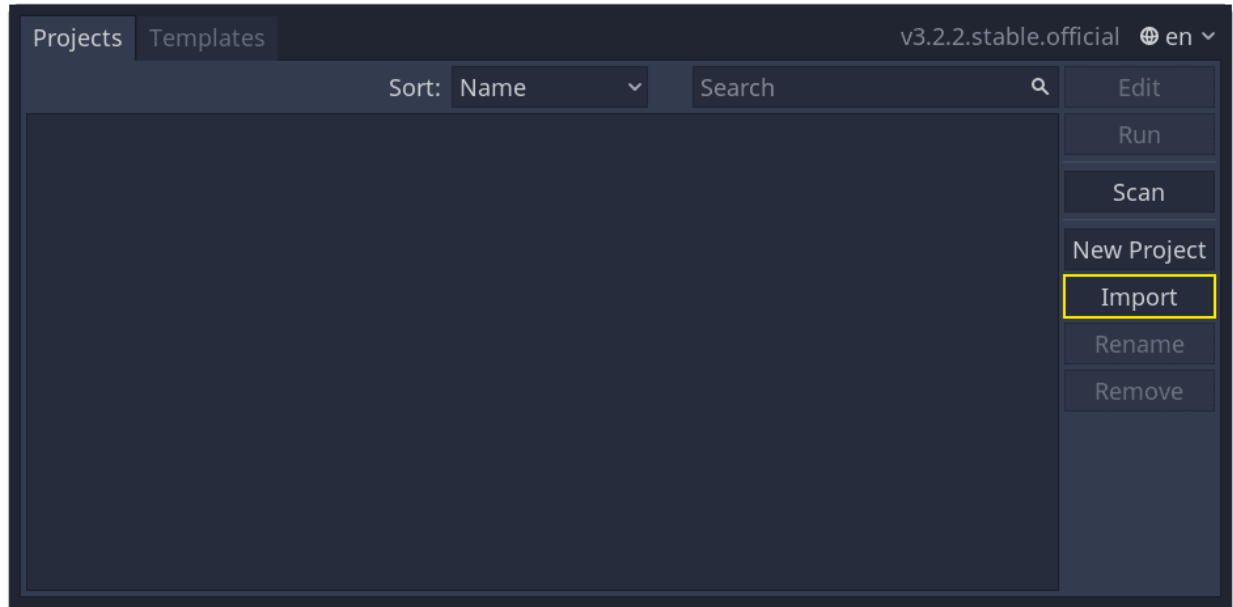
Every instance of the Ball scene starts with the same structure and properties as `ball.tscn`. However, you can modify each independently, such as changing how they bounce, how heavy they are, or any property exposed by the source scene.

In practice

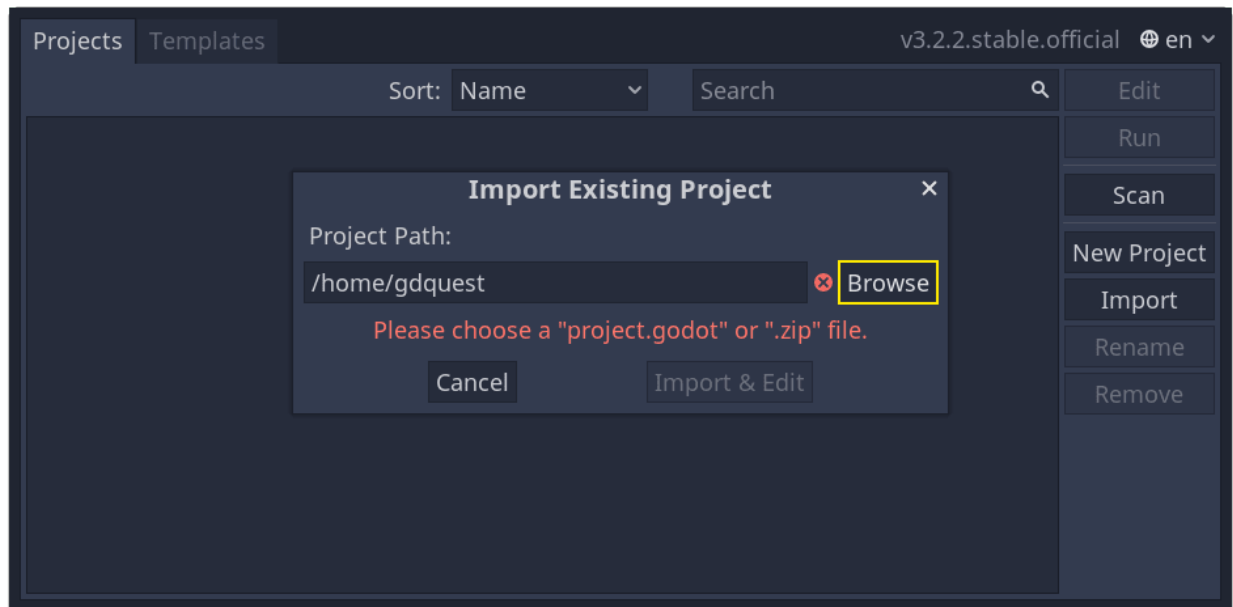
Let's use instancing in practice to see how it works in Godot. We invite you to download the ball's sample project we prepared for you: [instancing_starter.zip](#).

Extract the archive on your computer. To import it, you need the Project Manager. The Project Manager is accessed by opening Godot, or if you already have Godot opened, click on Project -> Quit to Project List (Ctrl + Shift + Q, Ctrl + Option + Cmd + Q on macOS)

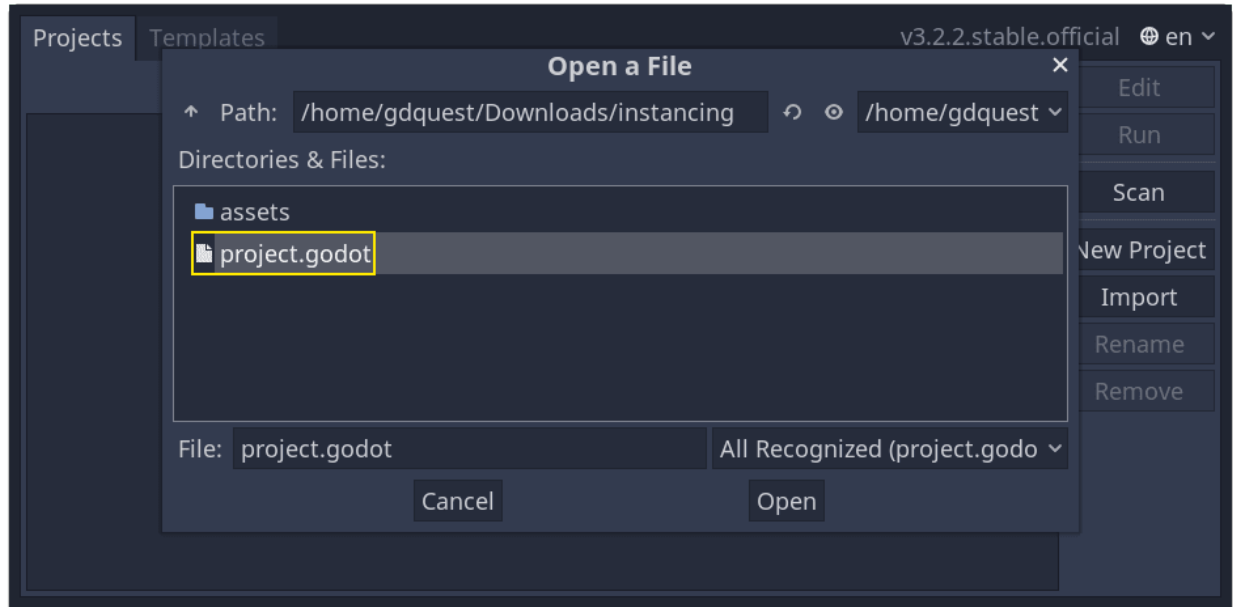
In the Project Manager, click the Import button to import the project.



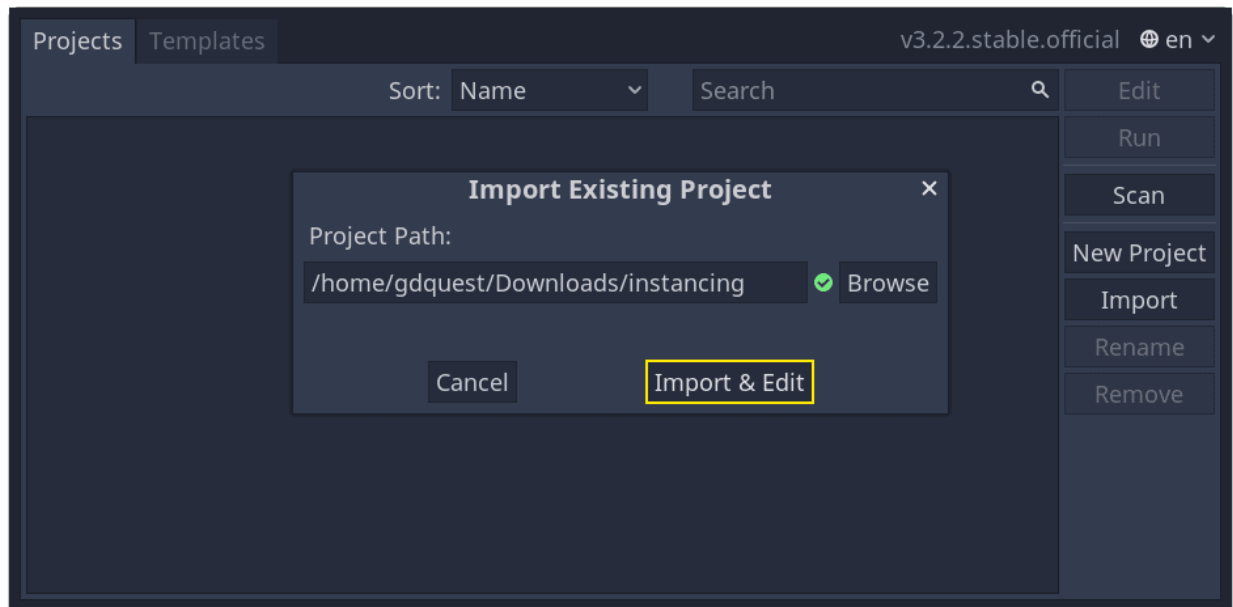
In the pop-up that appears, click the browse button and navigate to the folder you extracted.



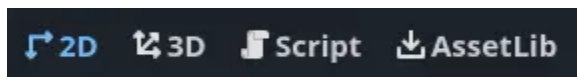
Double-click the project.godot file to open it.

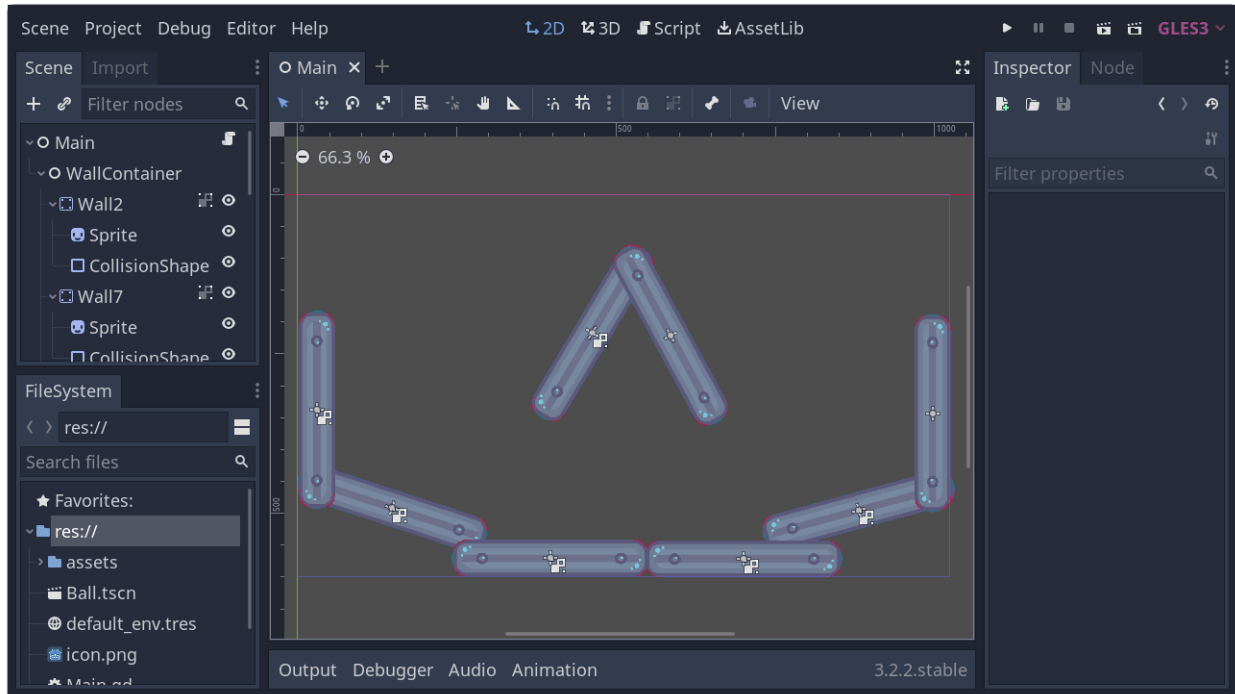


Finally, click the Import & Edit button.

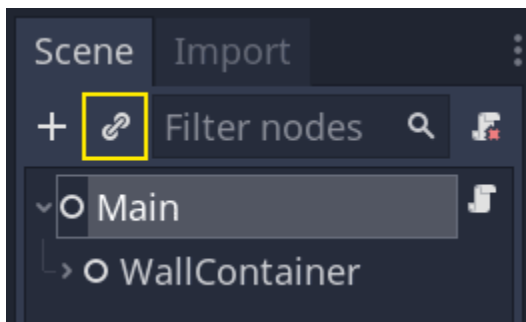


The project contains two packed scenes: `main.tscn`, containing walls against which the ball collides, and `ball.tscn`. The Main scene should open automatically. If you're seeing an empty 3D scene instead of the main scene, click the 2D button at the top of the screen.

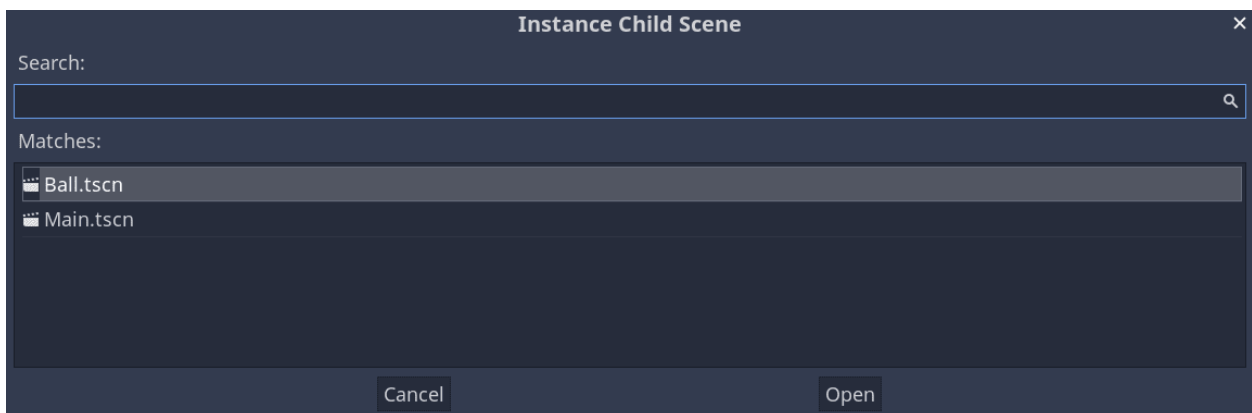




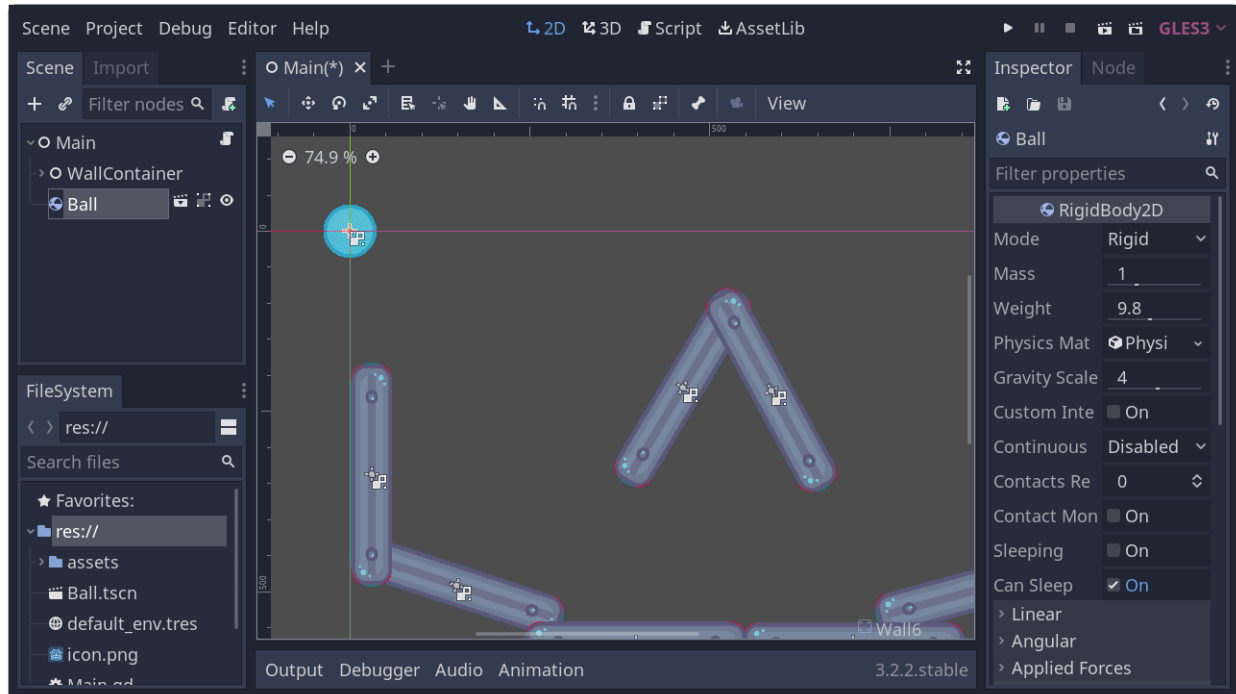
Let's add a ball as a child of the Main node. In the Scene dock, select the Main node. Then, click the link icon at the top of the scene dock. This button allows you to add an instance of a scene as a child of the currently selected node.



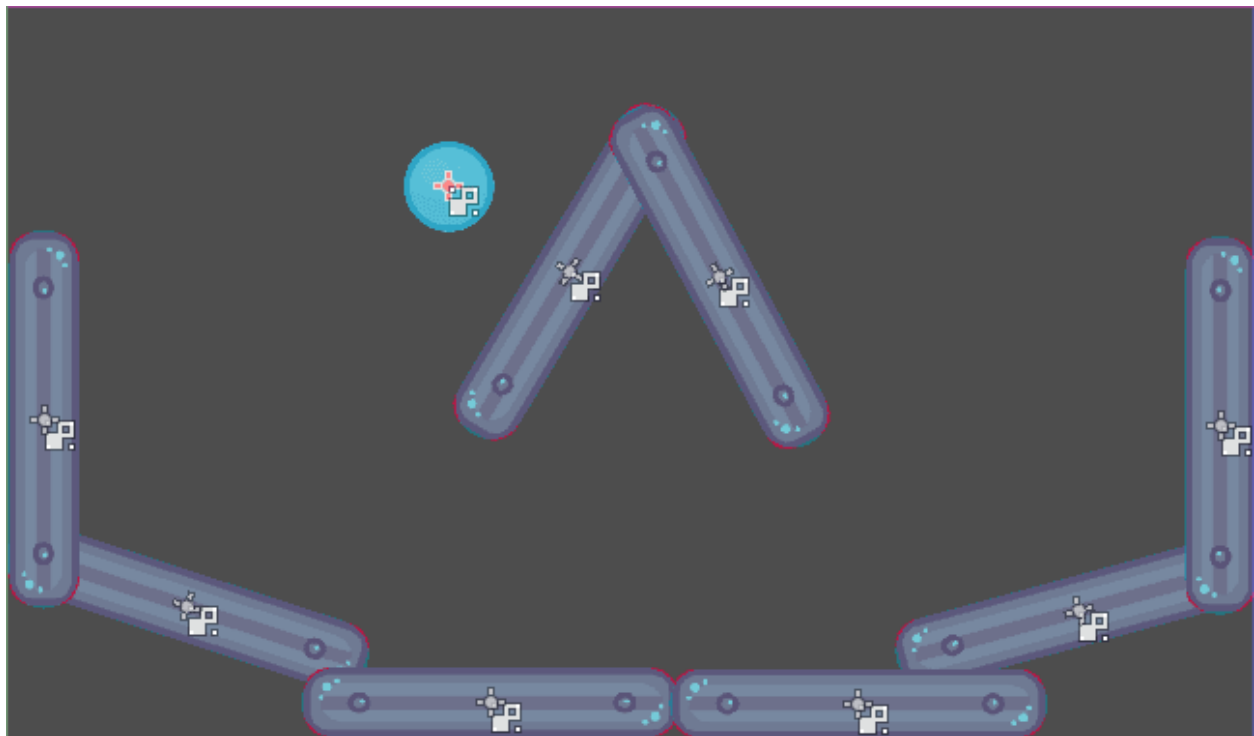
Double-click the ball scene to instance it.



The ball appears in the top-left corner of the viewport.

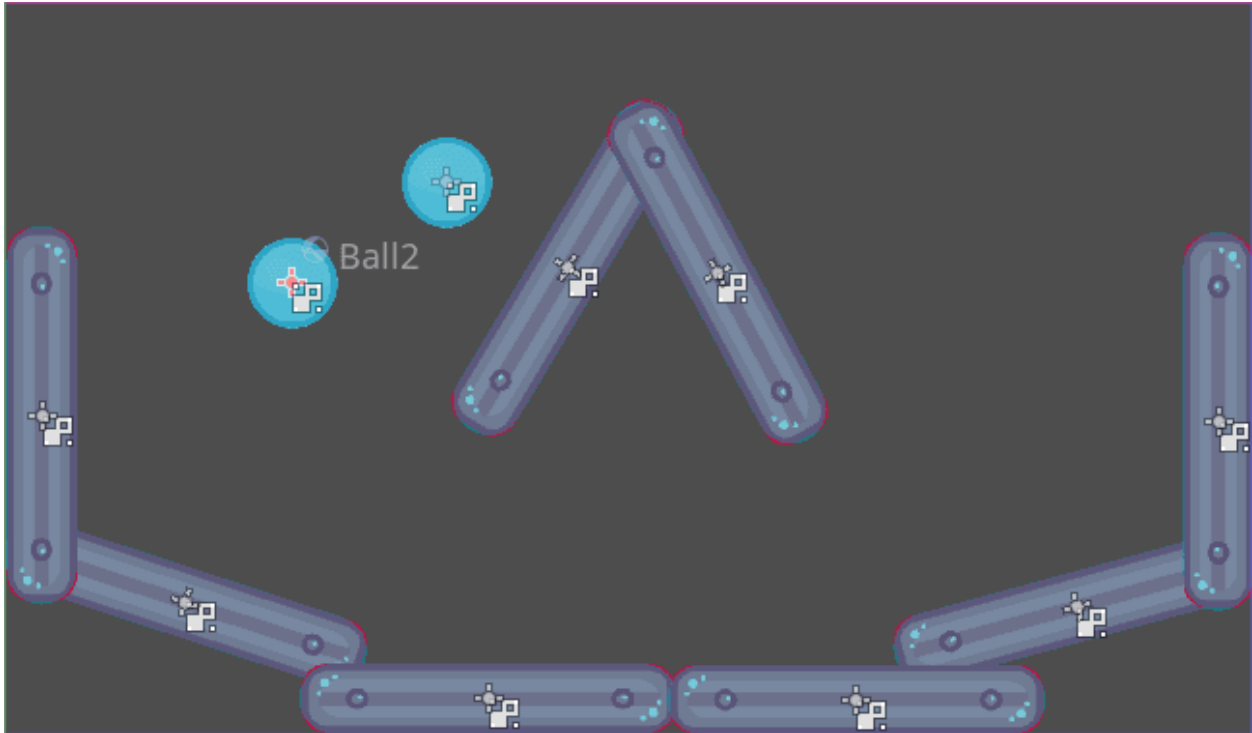


Click on it and drag it towards the center of the view.

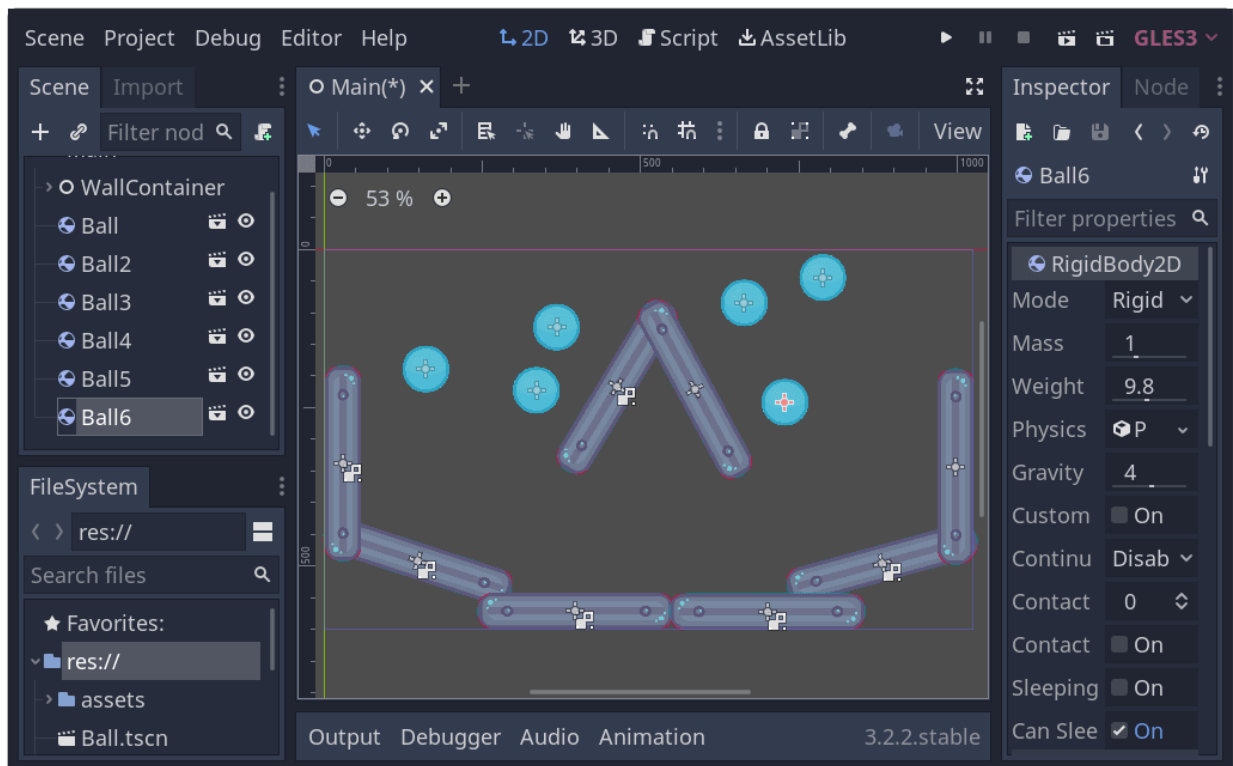


Play the game by pressing F5 (Cmd + B on macOS). You should see it fall.

Now, we want to create more instances of the Ball node. With the ball still selected, press Ctrl + D (Cmd + D on macOS) to call the duplicate command. Click and drag to move the new ball to a different location.



You can repeat this process until you have several in the scene.



Play the game again. You should now see every ball fall independently from one another. This is what instances do. Each is an independent reproduction of a template scene.

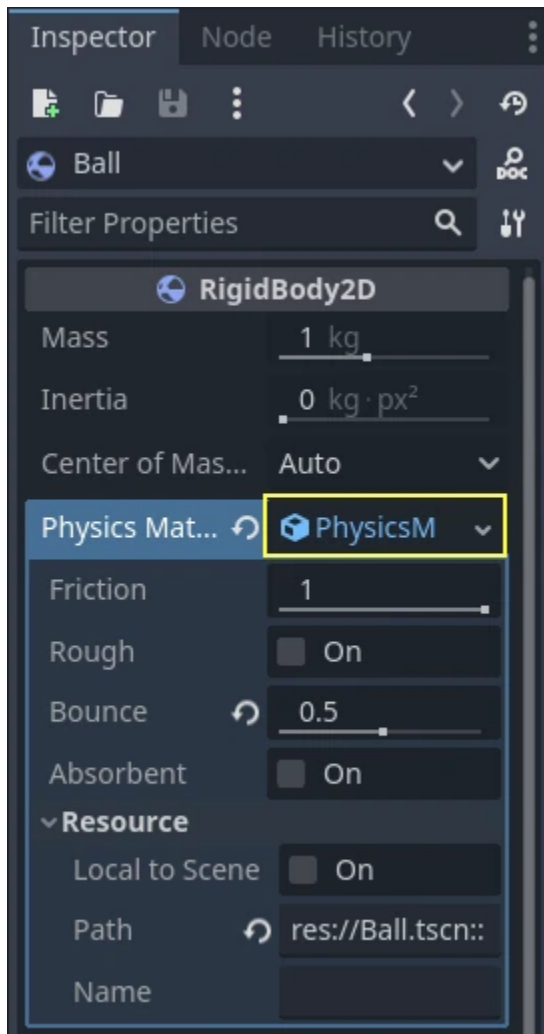
Editing scenes and instances

There is more to instances. With this feature, you can:

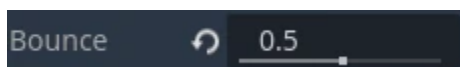
1. Change the properties of one ball without affecting the others using the Inspector.
2. Change the default properties of every Ball by opening the ball.tscn scene and making a change to the Ball node there. Upon saving, all instances of the Ball in the project will see their values update.

Note: Changing a property on an instance always overrides values from the corresponding packed scene.

Let's try this. Open ball.tscn and select the Ball node. In the Inspector on the right, click on the Physics-Material property to expand it.

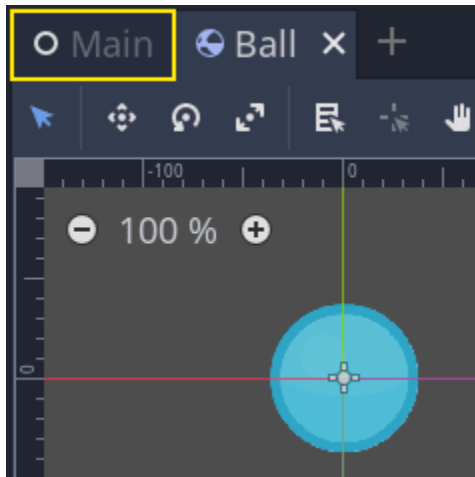


Set its Bounce property to 0.5 by clicking on the number field, typing 0.5, and pressing Enter.



Play the game by pressing F5 and notice how all balls now bounce a lot more. As the Ball scene is a template for all instances, modifying it and saving causes all instances to update accordingly.

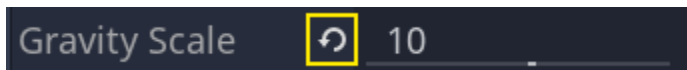
Let's now adjust an individual instance. Head back to the Main scene by clicking on the corresponding tab above the viewport.



Select one of the instanced Ball nodes and, in the Inspector, set its Gravity Scale value to 10.



A grey "revert" button appears next to the adjusted property.



This icon indicates you are overriding a value from the source packed scene. Even if you modify the property in the original scene, the value override will be preserved in the instance. Clicking the revert icon will restore the property to the value in the saved scene.

Rerun the game and notice how this ball now falls much faster than the others.

Note: If you change a value on the `PhysicsMaterial` of one instance, it will affect all the others. This is because `PhysicsMaterial` is a resource, and resources are shared between instances. To make a resource unique for one instance, right-click on it in the Inspector and click `Make Unique` in the contextual menu.

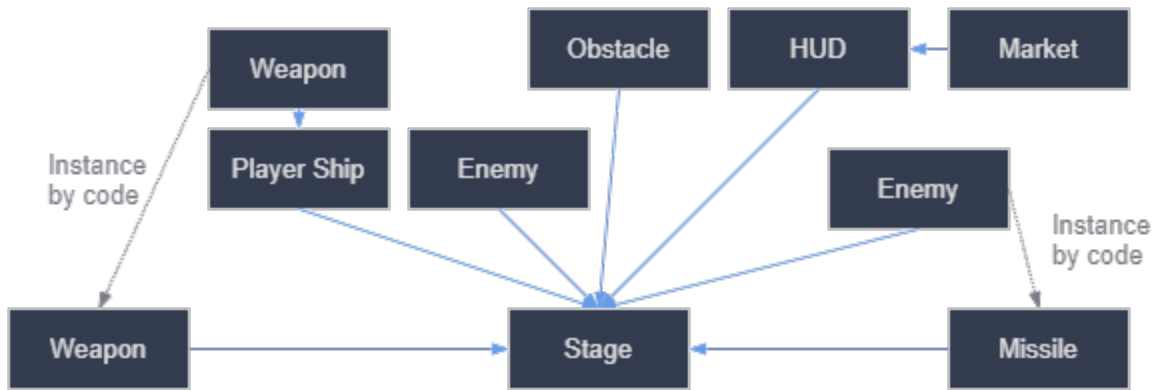
Resources are another essential building block of Godot games we will cover in a later lesson.

Scene instances as a design language

Instances and scenes in Godot offer an excellent design language, setting the engine apart from others out there. We designed Godot around this concept from the ground up.

We recommend dismissing architectural code patterns when making games with Godot, such as Model-View-Controller (MVC) or Entity-Relationship diagrams. Instead, you can start by imagining the elements players will see in your game and structure your code around them.

For example, you could break down a shooter game like so:

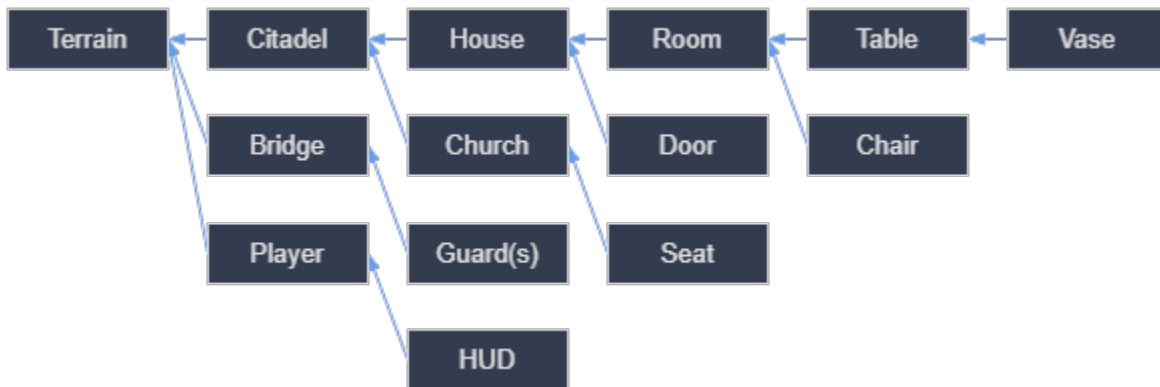


You can come up with a diagram like this for almost any type of game. Each rectangle represents an entity that's visible in the game from the player's perspective. The arrows tell you which scene owns which.

Once you have a diagram, we recommend creating a scene for each element listed in it to develop your game. You'll use instancing, either by code or directly in the editor, to build your tree of scenes.

Programmers tend to spend a lot of time designing abstract architectures and trying to fit components into it. Designing based on scenes makes development faster and more straightforward, allowing you to focus on the game logic itself. Because most game components map directly to a scene, using a design based on scene instantiation means you need little other architectural code.

Here's the example of a scene diagram for an open-world game with tons of assets and nested elements:



Imagine we started by creating the room. We could make a couple of different room scenes, with unique arrangements of furniture in them. Later, we could make a house scene that uses multiple room instances for the interior. We would create a citadel out of many instanced houses and a large terrain on which we would place the citadel. Each of these would be a scene instancing one or more sub-scenes.

Later, we could create scenes representing guards and add them to the citadel. They would be indirectly added to the overall game world.

With Godot, it's easy to iterate on your game like this, as all you need to do is create and instantiate more scenes. We designed the editor to be accessible to programmers, designers, and artists alike. A typical team development process can involve 2D or 3D artists, level designers, game designers, and animators, all working with the Godot editor.

Summary

Instancing, the process of producing an object from a blueprint, has many handy uses. With scenes, it gives you:

- The ability to divide your game into reusable components.
- A tool to structure and encapsulate complex systems.
- A language to think about your game project's structure in a natural way.

2.8.3 Scripting languages

This lesson will give you an overview of the available scripting languages in Godot. You will learn the pros and cons of each option. In the next part, you will write your first script using GDScript.

Scripts attach to a node and extend its behavior. This means that scripts inherit all functions and properties of the node they attach to.

For example, take a game where a Camera2D node follows a ship. The Camera2D node follows its parent by default. Imagine you want the camera to shake when the player takes damage. As this feature is not built into Godot, you would attach a script to the Camera2D node and code the shake.

Available scripting languages

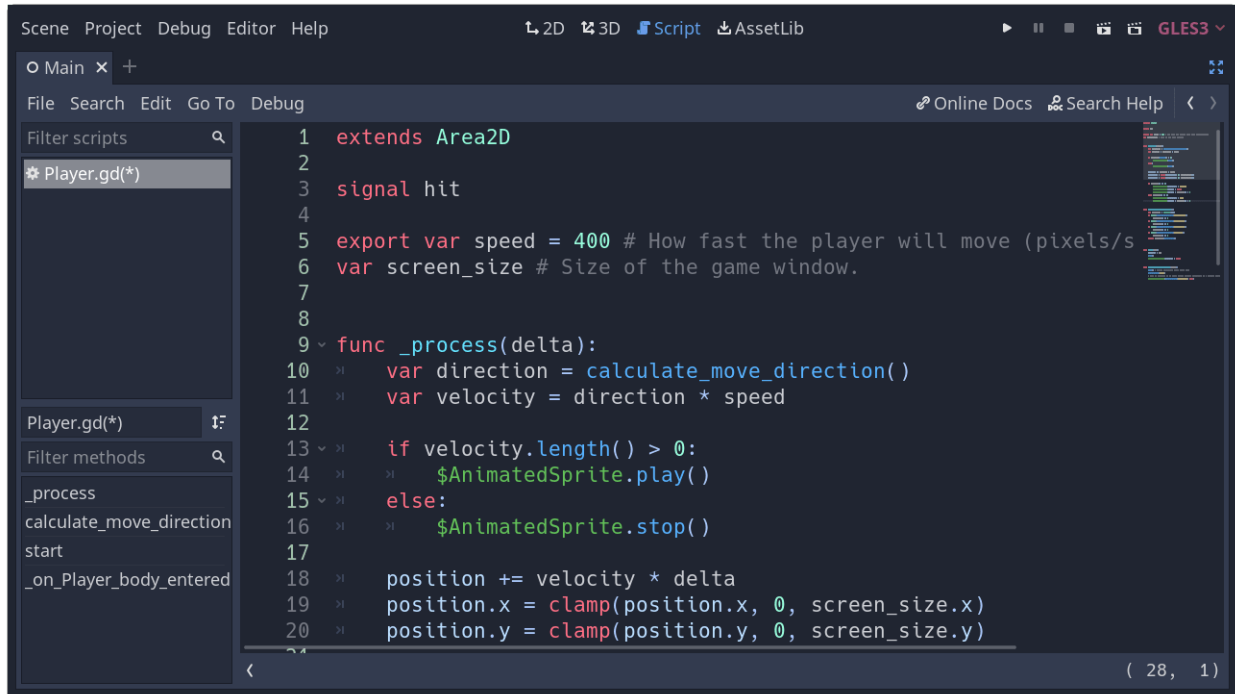
Godot offers four gameplay programming languages: GDScript, C#, and, via its GDEXTension technology, C and C++. There are more community-supported languages, but these are the official ones.

You can use multiple languages in a single project. For instance, in a team, you could code gameplay logic in GDScript as it's fast to write, and use C# or C++ to implement complex algorithms and maximize their performance. Or you can write everything in GDScript or C#. It's your call.

We provide this flexibility to answer the needs of different game projects and developers.

Which language should I use?

If you're a beginner, we recommend to start with GDScript. We made this language specifically for Godot and the needs of game developers. It has a lightweight and straightforward syntax and provides the tightest integration with Godot.



For C#, you will need an external code editor like [VSCode](#) or Visual Studio. While C# support is now mature, you will find fewer learning resources for it compared to GDScript. That's why we recommend C# mainly to users who already have experience with the language.

Let's look at each language's features, as well as its pros and cons.

GDScript

GDScript is an [object-oriented](#) and [imperative](#) programming language built for Godot. It's made by and for game developers to save you time coding games. Its features include:

- A simple syntax that leads to short files.
- Blazing fast compilation and loading times.
- Tight editor integration, with code completion for nodes, signals, and more information from the scene it's attached to.
- Built-in vector and transform types, making it efficient for heavy use of linear algebra, a must for games.
- Supports multiple threads as efficiently as statically typed languages.
- No [garbage collection](#), as this feature eventually gets in the way when creating games. The engine counts references and manages the memory for you in most cases by default, but you can also control memory if you need to.
- [Gradual typing](#). Variables have dynamic types by default, but you also can use type hints for strong type checks.

GDScript looks like Python as you structure your code blocks using indentations, but it doesn't work the same way in practice. It's inspired by multiple languages, including Squirrel, Lua, and Python.

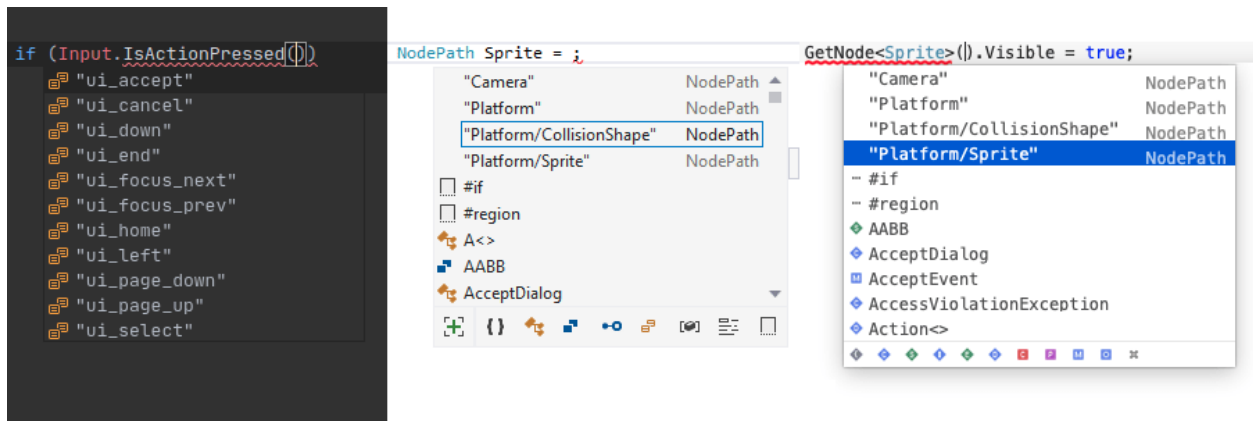
Note: Why don't we use Python or Lua directly?

Years ago, Godot used Python, then Lua. Both languages' integration took a lot of work and had severe limitations. For example, threading support was a big challenge with Python.

Developing a dedicated language doesn't take us more work and we can tailor it to game developers' needs. We're now working on performance optimizations and features that would've been difficult to offer with third-party languages.

.NET / C#

As Microsoft's C# is a favorite amongst game developers, we officially support it. C# is a mature and flexible language with tons of libraries written for it. We were able to add support for it thanks to a generous donation from Microsoft.



C# offers a good tradeoff between performance and ease of use, although you should be aware of its garbage collector.

Note: You must use the .NET edition of the Godot editor to script in C#. You can download it on the Godot website's [download](#) page.

Since Godot uses .NET 6, in theory, you can use any third-party .NET library or framework in Godot, as well as any Common Language Infrastructure-compliant programming language, such as F#, Boo, or ClojureCLR. However, C# is the only officially supported .NET option.

Note: GDScript code itself doesn't execute as fast as compiled C# or C++. However, most script code calls functions written with fast algorithms in C++ code inside the engine. In many cases, writing gameplay logic in GDScript, C#, or C++ won't have a significant impact on performance.

Attention: Projects written in C# using Godot 4 currently cannot be exported to the web platform. To use C# on that platform, consider Godot 3 instead. Android and iOS platform support is available as of Godot 4.2, but is experimental and some limitations apply.

C++ via GDEXTENSION

GDEXTENSION allows you to write game code in C++ without needing to recompile Godot.



```
Heatmap.cpp x
24 #include "Heatmap.h"
23 #include <TileMap.hpp>
22 #include <SceneTree.hpp>
21 #include <Viewport.hpp>
20 #include <deque>
19 #include <TileSet.hpp>
18 #include <algorithm>
17
16 namespace godot {
15     inline float lerp(const float& a, const float& b, const float& t) {
14         return a + t * (b - a);
13     }
12
11     void Heatmap::_register_methods() {
10         //public
9         register_method("_ready", &Heatmap::_ready);
8         register_method("_draw", &Heatmap::_draw);
7         register_method("_process", &Heatmap::_process);
6         register_method("best_direction_for", &Heatmap::best_direction_for);
5         register_method("calculate_point_index", &Heatmap::calculate_point_index);
4         register_method("calculate_point_index_for_world_position",
3             &Heatmap::calculate_point_index_for_world_position);
2
1         //semi-private
0         register_method("_on_Events_player_moved", &Heatmap::on_Events_player_moved);
25
1         //properties
2         register_property<Heatmap, NodePath>("pathfinding_tilemap", &Heatmap::m_pathfinding_tilemap,
NodePath());
10k godot-platformer-2d/game/src/Native/Heatmap/Heatmap.cpp 25:0 Top7:35AM 0.97 LF UTF-8 C++//l p master
```

You can use any version of the language or mix compiler brands and versions for the generated shared libraries, thanks to our use of an internal C API Bridge.

GDEXTENSION is the best choice for performance. You don't need to use it throughout an entire game, as you can write other parts in GDScript or C#.

When working with GDEXTENSION, the available types, functions, and properties closely resemble Godot's actual C++ API.

Summary

Scripts are files containing code that you attach to a node to extend its functionality.

Godot supports four official scripting languages, offering you flexibility between performance and ease of use.

You can mix languages, for instance, to implement demanding algorithms with C or C++ and write most of the game logic with GDScript or C#.

2.8.4 Creating your first script

In this lesson, you will code your first script to make the Godot icon turn in circles using GDScript. As we mentioned in the introduction, we assume you have programming foundations. The equivalent C# code has been included in another tab for convenience.

See also:

To learn more about GDScript, its keywords, and its syntax, head to the GDScript reference.

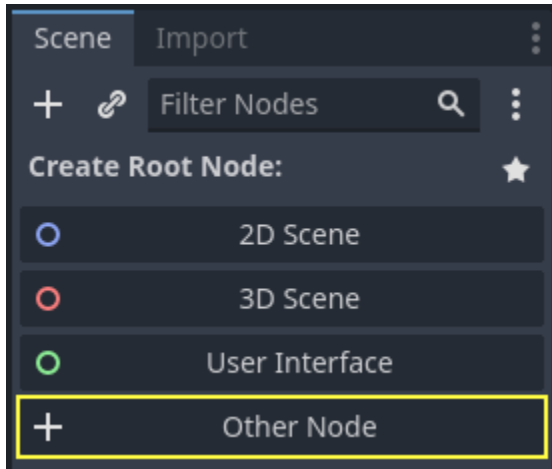
See also:

To learn more about C#, head to the C# basics page.

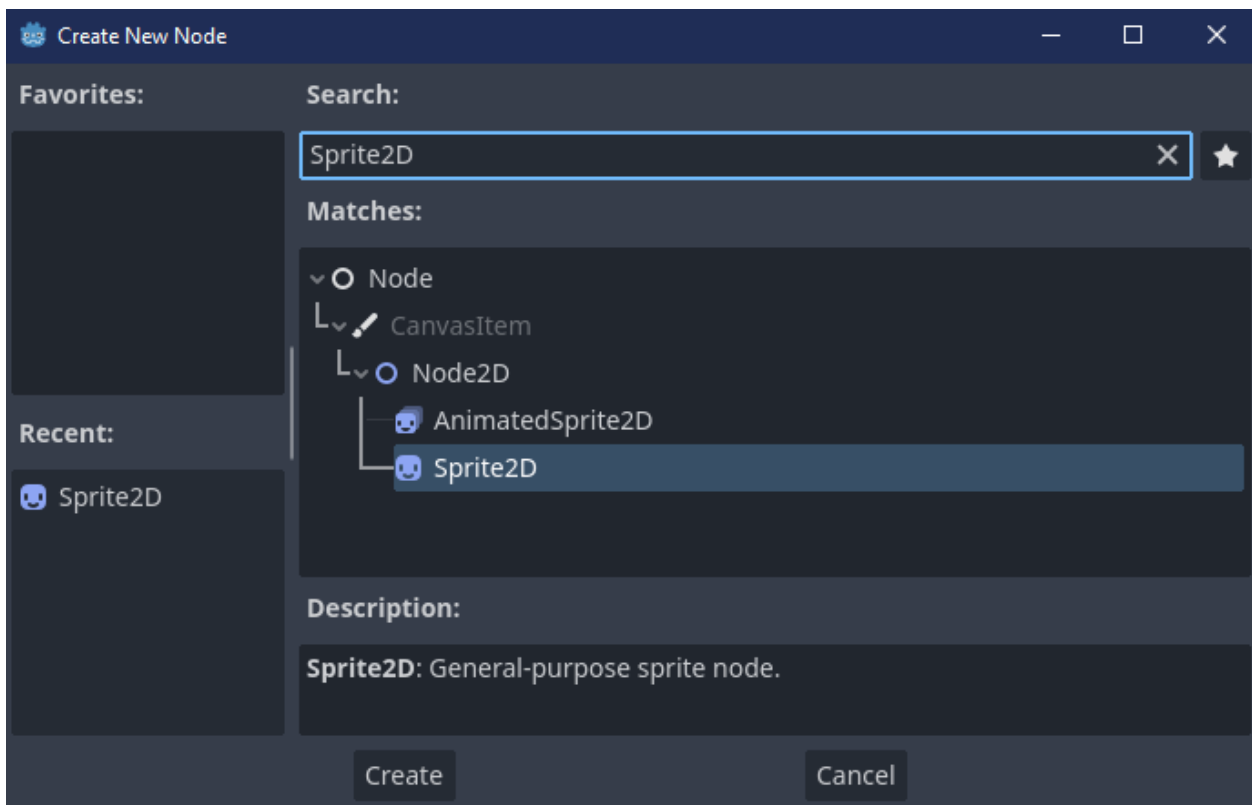
Project setup

Please create a new project to start with a clean slate. Your project should contain one picture: the Godot icon, which we often use for prototyping in the community.

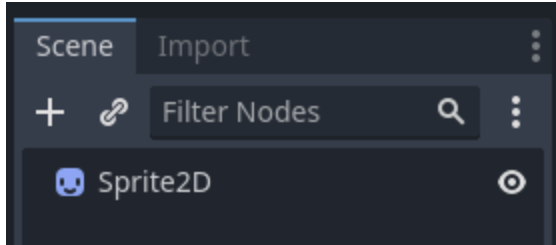
We need to create a `Sprite2D` node to display it in the game. In the Scene dock, click the Other Node button.



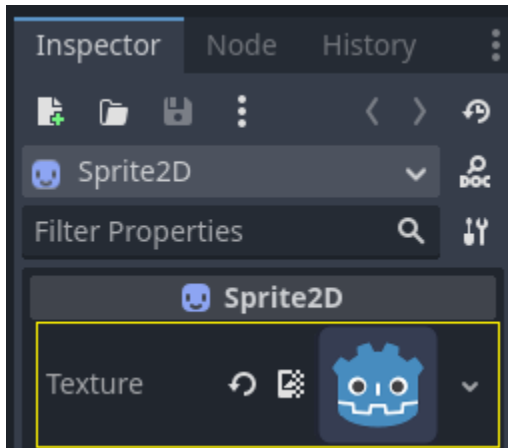
Type "Sprite2D" in the search bar to filter nodes and double-click on `Sprite2D` to create the node.



Your Scene tab should now only have a `Sprite2D` node.

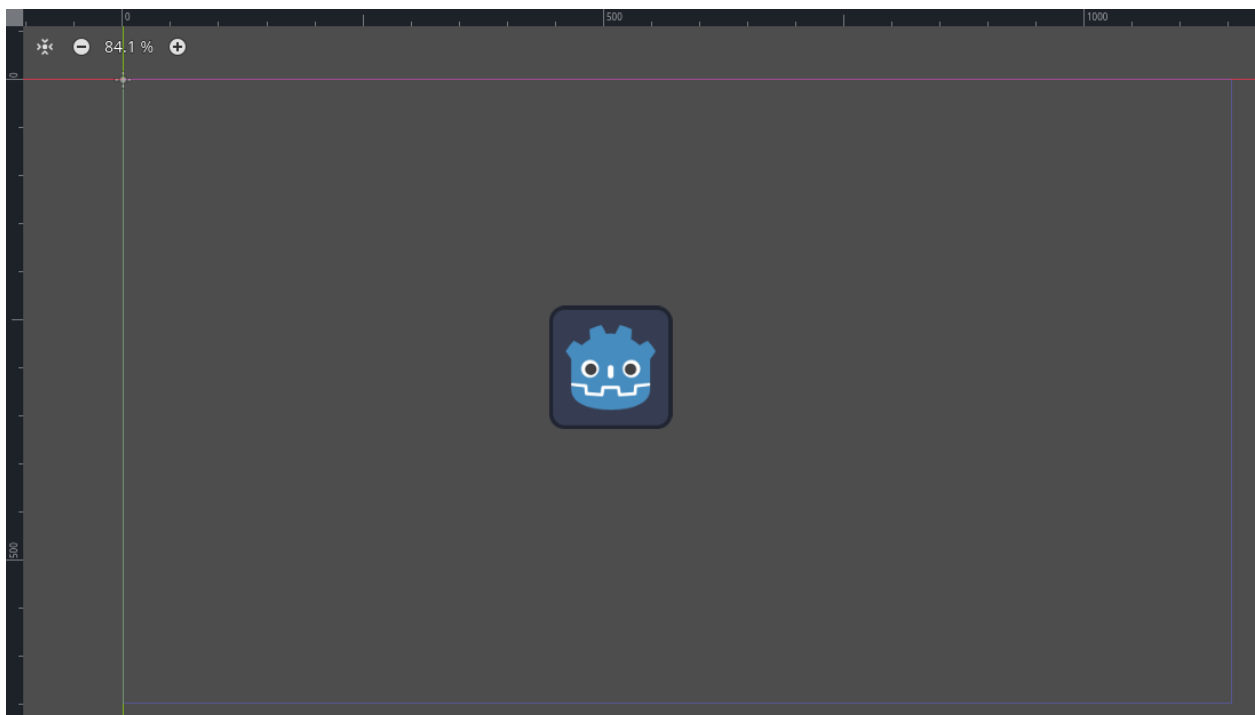


A Sprite2D node needs a texture to display. In the Inspector on the right, you can see that the Texture property says "[empty]". To display the Godot icon, click and drag the file icon.svg from the FileSystem dock onto the Texture slot.



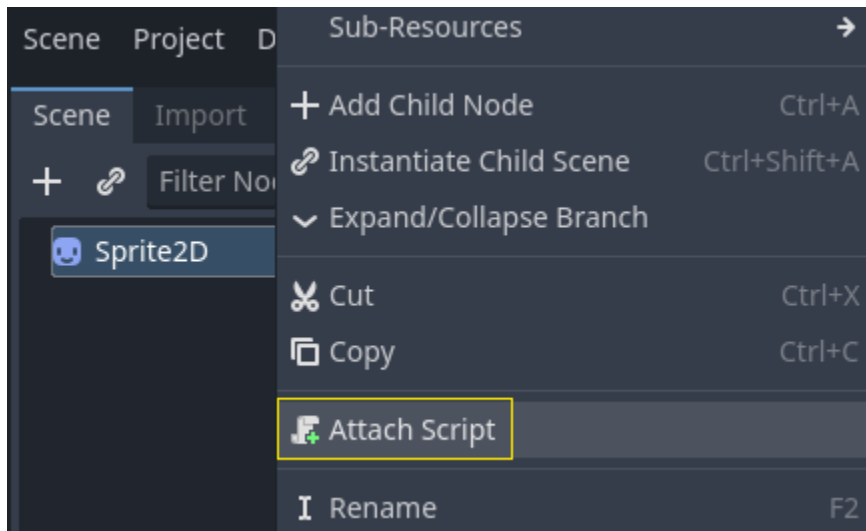
Note: You can create Sprite2D nodes automatically by dragging and dropping images on the viewport.

Then, click and drag the icon in the viewport to center it in the game view.



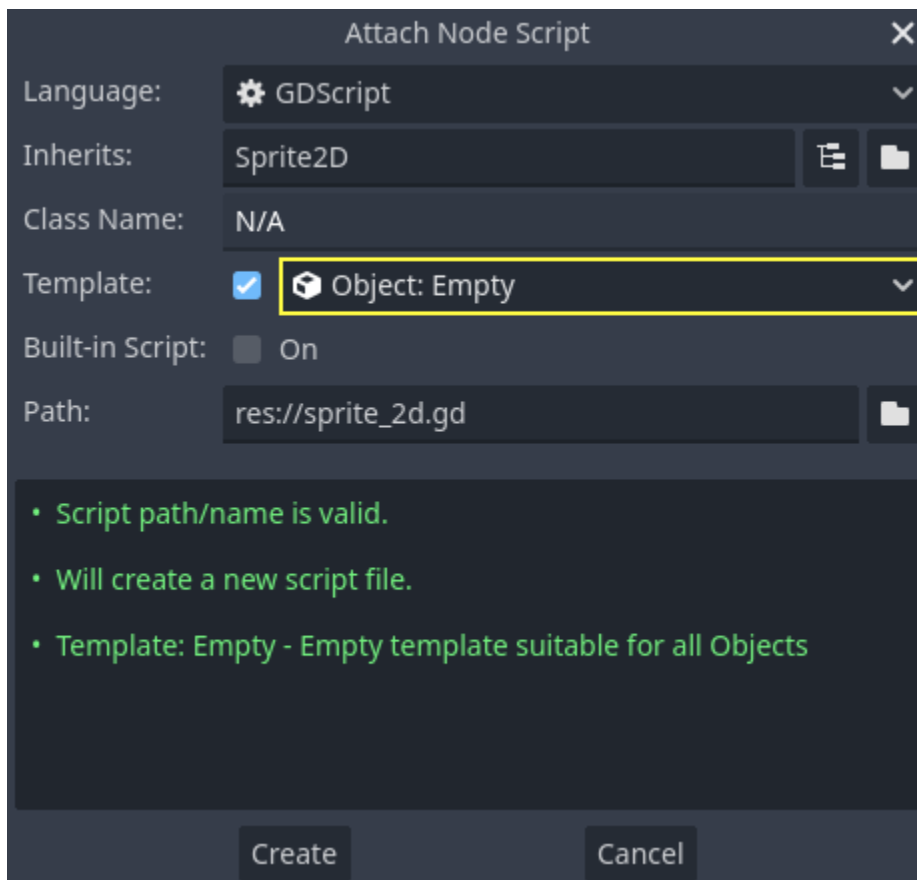
Creating a new script

To create and attach a new script to our node, right-click on Sprite2D in the scene dock and select "Attach Script".



The Attach Node Script window appears. It allows you to select the script's language and file path, among other options.

Change the Template field from "Node: Default" to "Object: Empty" to start with a clean file. Leave the other options set to their default values and click the Create button to create the script.



The Script workspace should appear with your new `sprite_2d.gd` file open and the following line of code:

GDScript

```
extends Sprite2D
```

C#

```
using Godot;

public partial class MySprite2D : Sprite2D
{
}
```

Every GDScript file is implicitly a class. The `extends` keyword defines the class this script inherits or extends. In this case, it's `Sprite2D`, meaning our script will get access to all the properties and functions of the `Sprite2D` node, including classes it extends, like `Node2D`, `CanvasItem`, and `Node`.

Note: In GDScript, if you omit the line with the `extends` keyword, your class will implicitly extend `RefCounted`, which Godot uses to manage your application's memory.

Inherited properties include the ones you can see in the Inspector dock, like our node's texture.

Note: By default, the Inspector displays a node's properties in "Title Case", with capitalized words separated by a space. In GDScript code, these properties are in "snake_case", which is lowercase with words separated by an underscore.

You can hover over any property's name in the Inspector to see a description and its identifier in code.

Hello, world!

Our script currently doesn't do anything. Let's make it print the text "Hello, world!" to the Output bottom panel to get started.

Add the following code to your script:

GDScript

```
func _init():
    print("Hello, world!")
```

C#

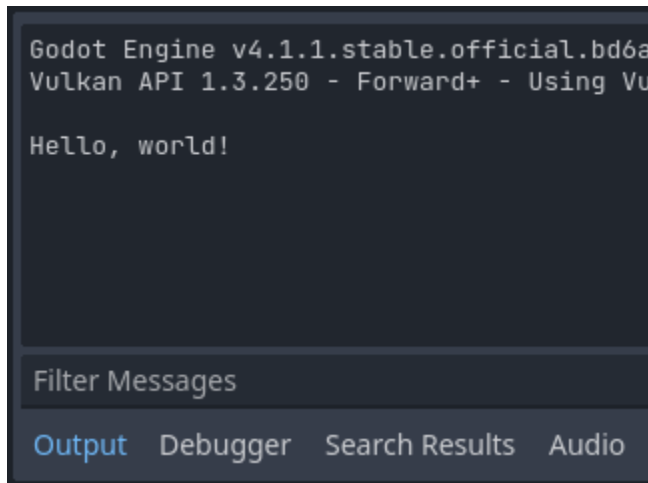
```
public MySprite2D()
{
    GD.Print("Hello, world!");
}
```

Let's break it down. The `func` keyword defines a new function named `_init`. This is a special name for our class's constructor. The engine calls `_init()` on every object or node upon creating it in memory, if you define this function.

Note: GDScript is an indent-based language. The tab at the start of the line that says `print()` is necessary for the code to work. If you omit it or don't indent a line correctly, the editor will highlight it in red and

display the following error message: "Indented block expected".

Save the scene as `sprite_2d.tscn` if you haven't already, then press F6 (Cmd + R on macOS) to run it. Look at the Output bottom panel that expands. It should display "Hello, world!".



Delete the `_init()` function, so you're only left with the line `extends Sprite2D`.

Turning around

It's time to make our node move and rotate. To do so, we're going to add two member variables to our script: the movement speed in pixels per second and the angular speed in radians per second. Add the following after the `extends Sprite2D` line.

GDScript

```
var speed = 400
var angular_speed = PI
```

C#

```
private int _speed = 400;
private float _angularSpeed = Mathf.Pi;
```

Member variables sit near the top of the script, after any "extends" lines, but before functions. Every node instance with this script attached to it will have its own copy of the `speed` and `angular_speed` properties.

Note: Angles in Godot work in radians by default, but you have built-in functions and properties available if you prefer to calculate angles in degrees instead.

To move our icon, we need to update its position and rotation every frame in the game loop. We can use the `_process()` virtual function of the Node class. If you define it in any class that extends the Node class, like `Sprite2D`, Godot will call the function every frame and pass it an argument named `delta`, the time elapsed since the last frame.

Note: Games work by rendering many images per second, each called a frame, and they do so in a loop. We measure the rate at which a game produces images in Frames Per Second (FPS). Most games aim for 60 FPS, although you might find figures like 30 FPS on slower mobile devices or 90 to 240 for virtual reality games.

The engine and game developers do their best to update the game world and render images at a constant time interval, but there are always small variations in frame render times. That's why the engine provides us with this delta time value, making our motion independent of our framerate.

At the bottom of the script, define the function:

GScript

```
func _process(delta):  
    rotation += angular_speed * delta
```

C#

```
public override void _Process(double delta)  
{  
    Rotation += _angularSpeed * (float)delta;  
}
```

The `func` keyword defines a new function. After it, we have to write the function's name and arguments it takes in parentheses. A colon ends the definition, and the indented blocks that follow are the function's content or instructions.

Note: Notice how `_process()`, like `_init()`, starts with a leading underscore. By convention, Godot's virtual functions, that is to say, built-in functions you can override to communicate with the engine, start with an underscore.

The line inside the function, `rotation += angular_speed * delta`, increments our sprite's rotation every frame. Here, `rotation` is a property inherited from the class `Node2D`, which `Sprite2D` extends. It controls the rotation of our node and works with radians.

Tip: In the code editor, you can ctrl-click on any built-in property or function like `position`, `rotation`, or `_process` to open the corresponding documentation in a new tab.

Run the scene to see the Godot icon turn in-place.

Note: In C#, notice how the `delta` argument taken by `_Process()` is a double. We therefore need to convert it to float when we apply it to the rotation.

Moving forward

Let's now make the node move. Add the following two lines inside of the `_process()` function, ensuring the new lines are indented the same way as the `rotation += angular_speed * delta` line before them.

GScript

```
var velocity = Vector2.UP.rotated(rotation) * speed  
  
position += velocity * delta
```

C#

```
var velocity = Vector2.Up.Rotated(Rotation) * _speed;

Position += velocity * (float)delta;
```

As we already saw, the `var` keyword defines a new variable. If you put it at the top of the script, it defines a property of the class. Inside a function, it defines a local variable: it only exists within the function's scope.

We define a local variable named `velocity`, a 2D vector representing both a direction and a speed. To make the node move forward, we start from the `Vector2` class's constant `Vector2.UP`, a vector pointing up, and rotate it by calling the `Vector2` method `rotated()`. This expression, `Vector2.UP.rotated(rotation)`, is a vector pointing forward relative to our icon. Multiplied by our speed property, it gives us a velocity we can use to move the node forward.

We add `velocity * delta` to the node's position to move it. The position itself is of type `Vector2`, a built-in type in Godot representing a 2D vector.

Run the scene to see the Godot head run in circles.

Note: Moving a node like that does not take into account colliding with walls or the floor. In Your first 2D game, you will learn another approach to moving objects while detecting collisions.

Our node currently moves by itself. In the next part, Listening to player input, we'll use player input to control it.

Complete script

Here is the complete `sprite_2d.gd` file for reference.

GDScript

```
extends Sprite2D

var speed = 400
var angular_speed = PI

func _process(delta):
    rotation += angular_speed * delta

    var velocity = Vector2.UP.rotated(rotation) * speed

    position += velocity * delta
```

C#

```
using Godot;

public partial class MySprite2D : Sprite2D
{
    private int _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Process(double delta)
    {
```

(continues on next page)

(continued from previous page)

```
Rotation += _angularSpeed * (float)delta;
var velocity = Vector2.Up.Rotated(Rotation) * _speed;

Position += velocity * (float)delta;
}
}
```

2.8.5 Listening to player input

Building upon the previous lesson, Creating your first script, let's look at another important feature of any game: giving control to the player. To add this, we need to modify our `sprite_2d.gd` code.

You have two main tools to process the player's input in Godot:

1. The built-in input callbacks, mainly `_unhandled_input()`. Like `_process()`, it's a built-in virtual function that Godot calls every time the player presses a key. It's the tool you want to use to react to events that don't happen every frame, like pressing Space to jump. To learn more about input callbacks, see [Using InputEvent](#).
2. The Input singleton. A singleton is a globally accessible object. Godot provides access to several in scripts. It's the right tool to check for input every frame.

We're going to use the Input singleton here as we need to know if the player wants to turn or move every frame.

For turning, we should use a new variable: `direction`. In our `_process()` function, replace the `rotation += angular_speed * delta` line with the code below.

GDScript

```
var direction = 0
if Input.is_action_pressed("ui_left"):
    direction = -1
if Input.is_action_pressed("ui_right"):
    direction = 1

rotation += angular_speed * direction * delta
```

C#

```
var direction = 0;
if (Input.IsActionPressed("ui_left"))
{
    direction = -1;
}
if (Input.IsActionPressed("ui_right"))
{
    direction = 1;
}

Rotation += _angularSpeed * direction * (float)delta;
```

Our `direction` local variable is a multiplier representing the direction in which the player wants to turn. A value of 0 means the player isn't pressing the left or the right arrow key. A value of 1 means the player wants to turn right, and -1 means they want to turn left.

To produce these values, we introduce conditions and the use of Input. A condition starts with the if keyword in GDScript and ends with a colon. The condition is the expression between the keyword and the end of the line.

To check if a key was pressed this frame, we call `Input.is_action_pressed()`. The method takes a text string representing an input action and returns true if the action is pressed, false otherwise.

The two actions we use above, "ui_left" and "ui_right", are predefined in every Godot project. They respectively trigger when the player presses the left and right arrows on the keyboard or left and right on a gamepad's D-pad.

Note: You can see and edit input actions in your project by going to Project -> Project Settings and clicking on the Input Map tab.

Finally, we use the direction as a multiplier when we update the node's rotation: `rotation += angular_speed * direction * delta`.

If you run the scene with this code, the icon should rotate when you press Left and Right.

Moving when pressing "up"

To only move when pressing a key, we need to modify the code that calculates the velocity. Replace the line starting with `var velocity` with the code below.

GDScript

```
var velocity = Vector2.ZERO
if Input.is_action_pressed("ui_up"):
    velocity = Vector2.UP.rotated(rotation) * speed
```

C#

```
var velocity = Vector2.Zero;
if (Input.IsActionPressed("ui_up"))
{
    velocity = Vector2.Up.Rotated(Rotation) * _speed;
}
```

We initialize the velocity with a value of `Vector2.ZERO`, another constant of the built-in `Vector` type representing a 2D vector of length 0.

If the player presses the "ui_up" action, we then update the velocity's value, causing the sprite to move forward.

Complete script

Here is the complete `sprite_2d.gd` file for reference.

GDScript

```
extends Sprite2D

var speed = 400
var angular_speed = PI
```

(continues on next page)

(continued from previous page)

```
func _process(delta):
    var direction = 0
    if Input.is_action_pressed("ui_left"):
        direction = -1
    if Input.is_action_pressed("ui_right"):
        direction = 1

    rotation += angular_speed * direction * delta

    var velocity = Vector2.ZERO
    if Input.is_action_pressed("ui_up"):
        velocity = Vector2.UP.rotated(rotation) * speed

    position += velocity * delta
```

C#

```
using Godot;

public partial class MySprite2D : Sprite2D
{
    private float _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Process(double delta)
    {
        var direction = 0;
        if (Input.IsActionPressed("ui_left"))
        {
            direction = -1;
        }
        if (Input.IsActionPressed("ui_right"))
        {
            direction = 1;
        }

        Rotation += _angularSpeed * direction * (float)delta;

        var velocity = Vector2.Zero;
        if (Input.IsActionPressed("ui_up"))
        {
            velocity = Vector2.Up.Rotated(Rotation) * _speed;
        }

        Position += velocity * (float)delta;
    }
}
```

If you run the scene, you should now be able to rotate with the left and right arrow keys and move forward by pressing Up.

Summary

In summary, every script in Godot represents a class and extends one of the engine's built-in classes. The node types your classes inherit from give you access to properties, such as rotation and position in our sprite's case. You also inherit many functions, which we didn't get to use in this example.

In GDScript, the variables you put at the top of the file are your class's properties, also called member variables. Besides variables, you can define functions, which, for the most part, will be your classes' methods.

Godot provides several virtual functions you can define to connect your class with the engine. These include `_process()`, to apply changes to the node every frame, and `_unhandled_input()`, to receive input events like key and button presses from the users. There are quite a few more.

The Input singleton allows you to react to the players' input anywhere in your code. In particular, you'll get to use it in the `_process()` loop.

In the next lesson, Using signals, we'll build upon the relationship between scripts and nodes by having our nodes trigger code in scripts.

2.8.6 Using signals

In this lesson, we will look at signals. They are messages that nodes emit when something specific happens to them, like a button being pressed. Other nodes can connect to that signal and call a function when the event occurs.

Signals are a delegation mechanism built into Godot that allows one game object to react to a change in another without them referencing one another. Using signals limits [coupling](#) and keeps your code flexible.

For example, you might have a life bar on the screen that represents the player's health. When the player takes damage or uses a healing potion, you want the bar to reflect the change. To do so, in Godot, you would use signals.

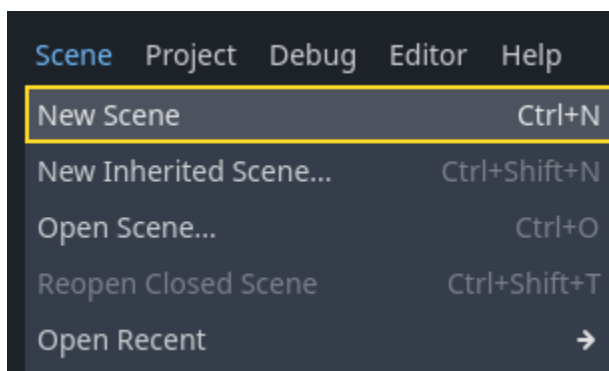
Note: As mentioned in the introduction, signals are Godot's version of the observer pattern. You can learn more about it here: <https://gameprogrammingpatterns.com/observer.html>

We will now use a signal to make our Godot icon from the previous lesson (Listening to player input) move and stop by pressing a button.

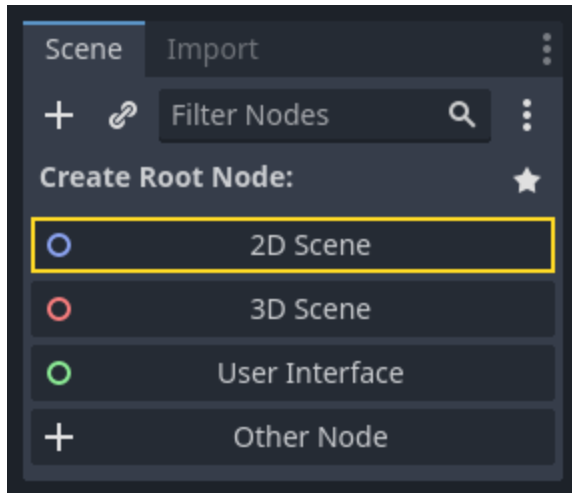
Scene setup

To add a button to our game, we will create a new main scene which will include both a Button and the `sprite_2d.tscn` scene we created in the Creating your first script lesson.

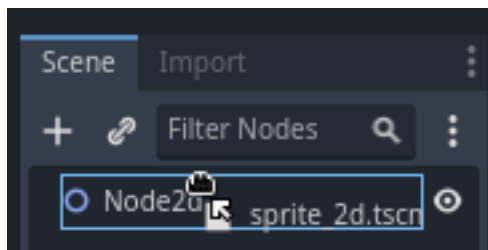
Create a new scene by going to the menu Scene -> New Scene.



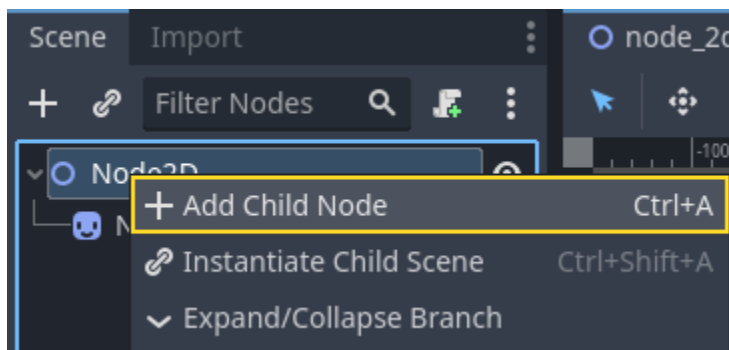
In the Scene dock, click the 2D Scene button. This will add a Node2D as our root.



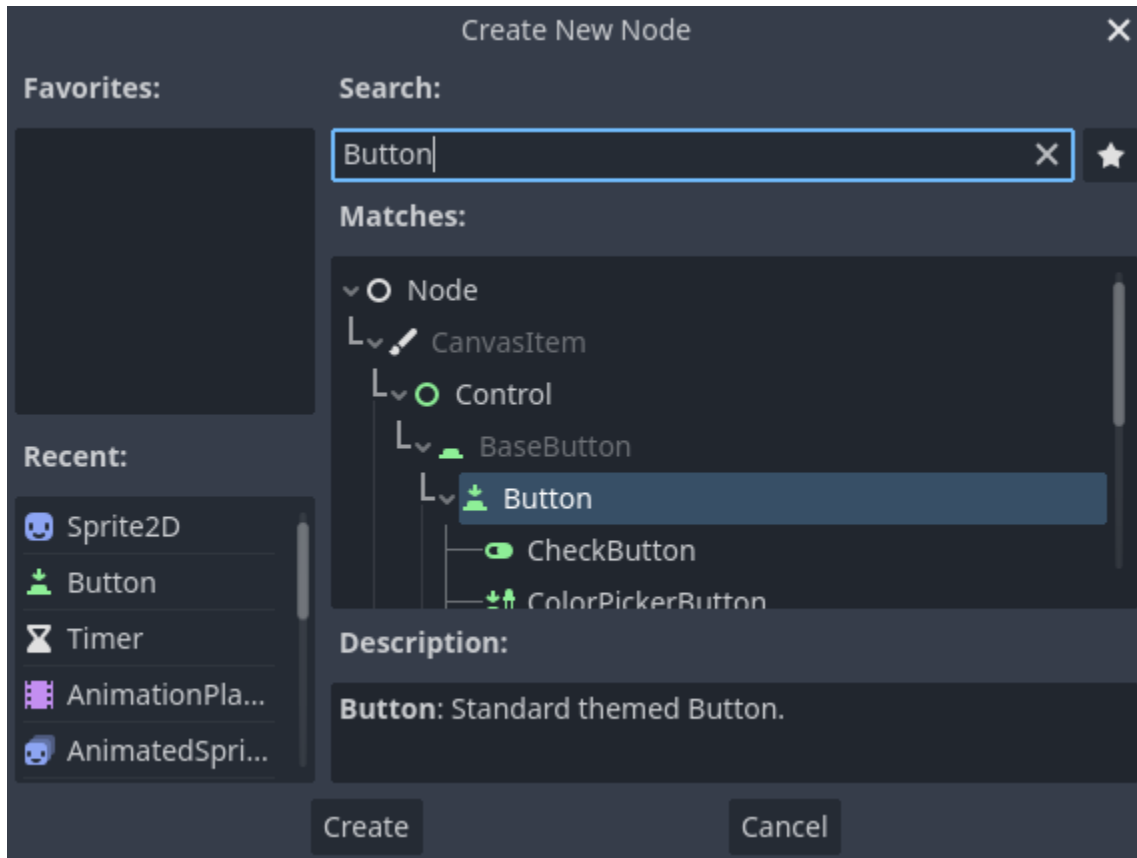
In the FileSystem dock, click and drag the `sprite_2d.tscn` file you saved previously onto the Node2D to instantiate it.



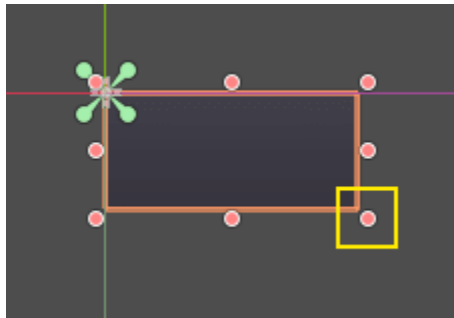
We want to add another node as a sibling of the Sprite2D. To do so, right-click on Node2D and select Add Child Node.



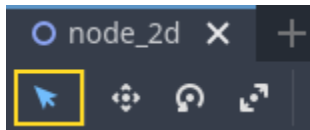
Search for the Button node and add it.



The node is small by default. Click and drag on the bottom-right handle of the Button in the viewport to resize it.

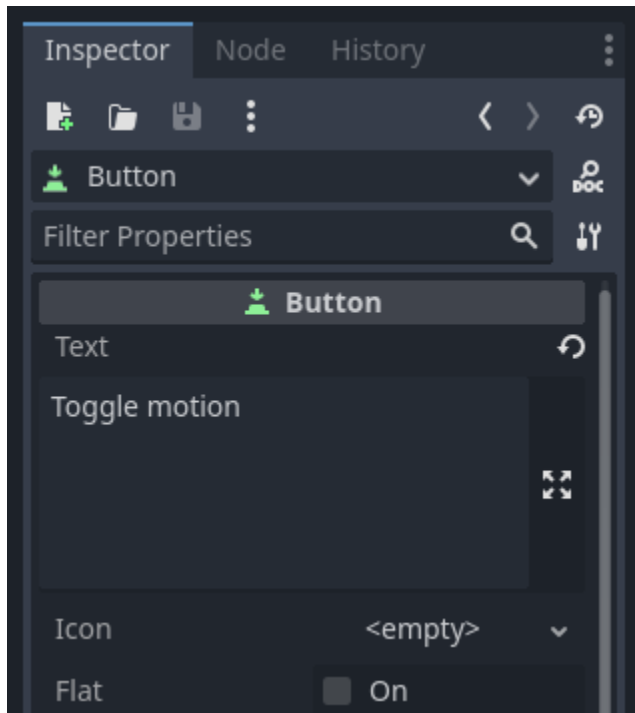


If you don't see the handles, ensure the select tool is active in the toolbar.

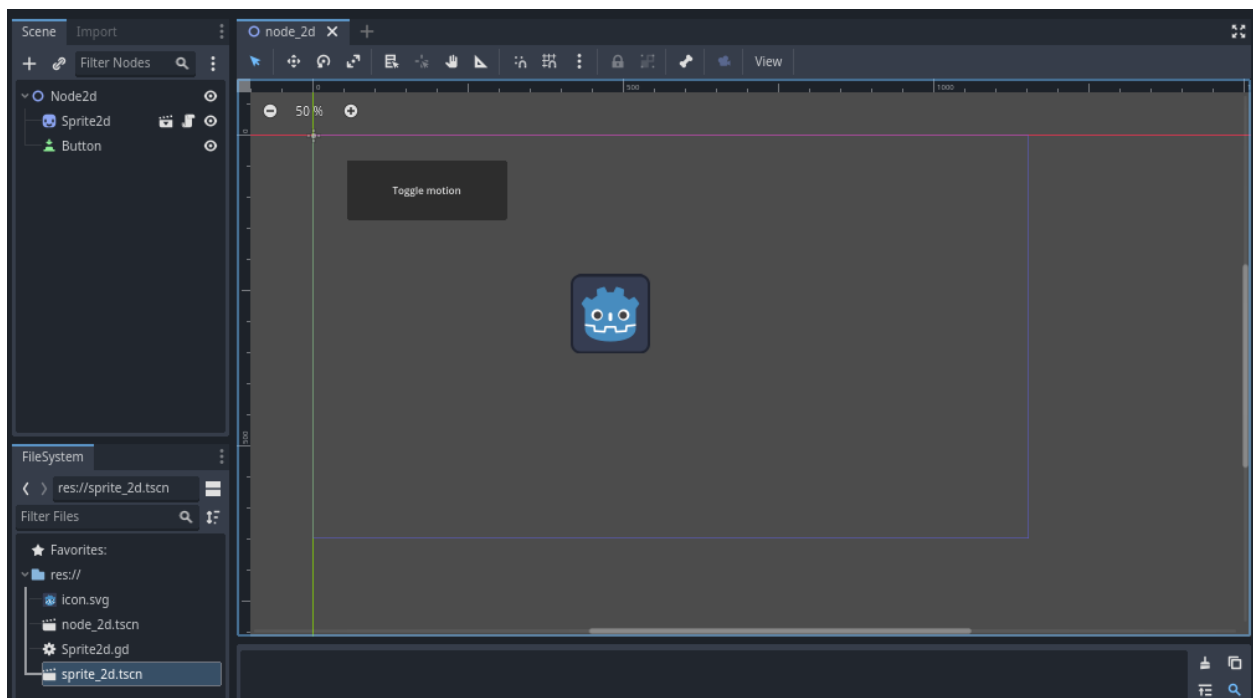


Click and drag on the button itself to move it closer to the sprite.

You can also write a label on the Button by editing its Text property in the Inspector. Enter Toggle motion.



Your scene tree and viewport should look like this.

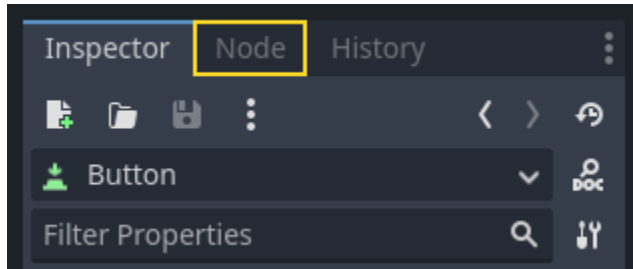


Save your newly created scene as `node_2d.tscn`, if you haven't already. You can then run it with F6 (Cmd + R on macOS). At the moment, the button will be visible, but nothing will happen if you press it.

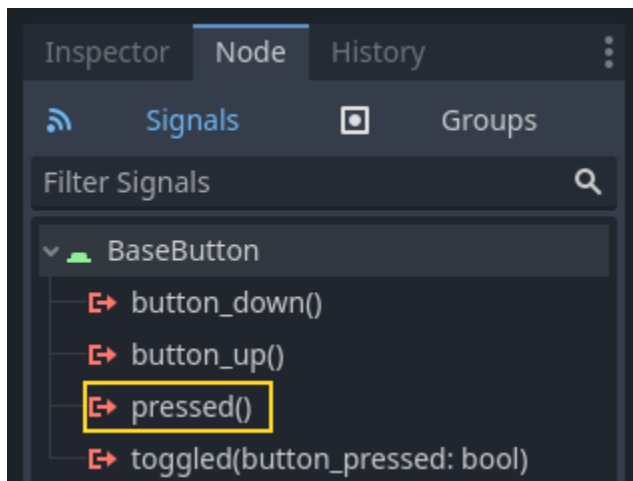
Connecting a signal in the editor

Here, we want to connect the Button's "pressed" signal to our Sprite2D, and we want to call a new function that will toggle its motion on and off. We need to have a script attached to the Sprite2D node, which we do from the previous lesson.

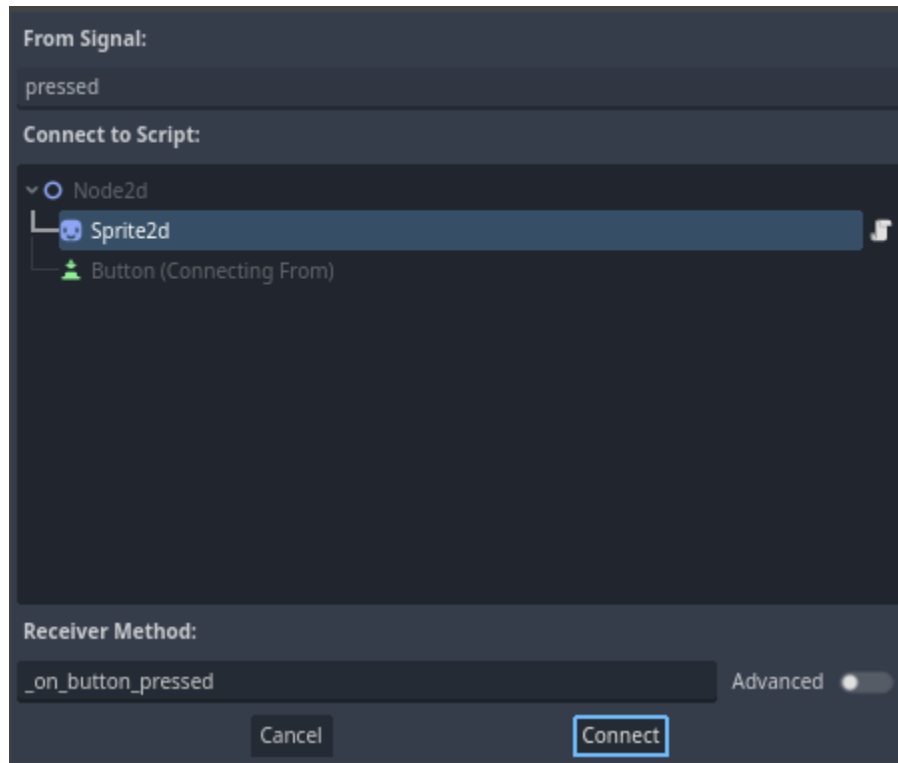
You can connect signals in the Node dock. Select the Button node and, on the right side of the editor, click on the tab named "Node" next to the Inspector.



The dock displays a list of signals available on the selected node.

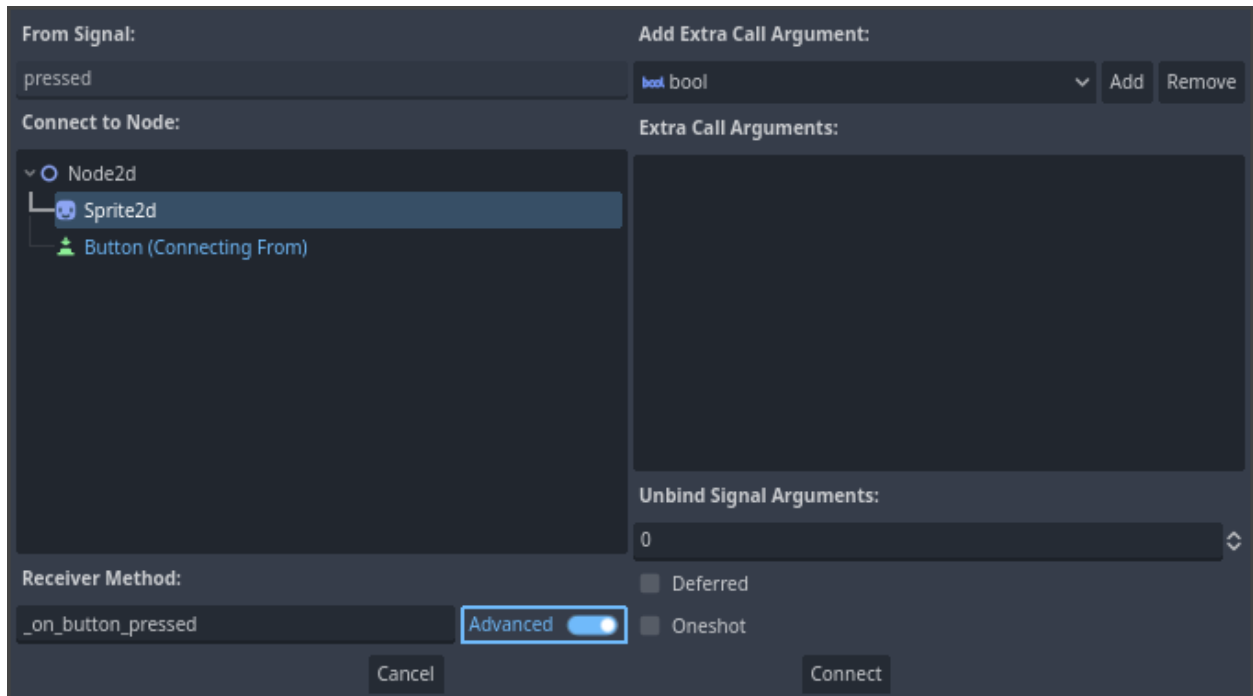


Double-click the "pressed" signal to open the node connection window.



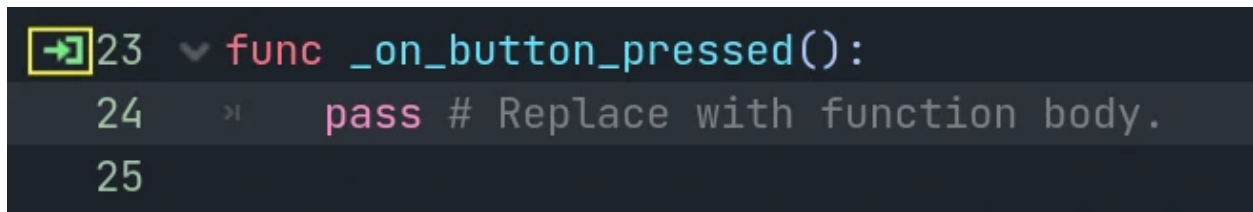
There, you can connect the signal to the `Sprite2D` node. The node needs a receiver method, a function that Godot will call when the `Button` emits the signal. The editor generates one for you. By convention, we name these callback methods `"_on_node_name_signal_name"`. Here, it'll be `"_on_button_pressed"`.

Note: When connecting signals via the editor's Node dock, you can use two modes. The simple one only allows you to connect to nodes that have a script attached to them and creates a new callback function on them.

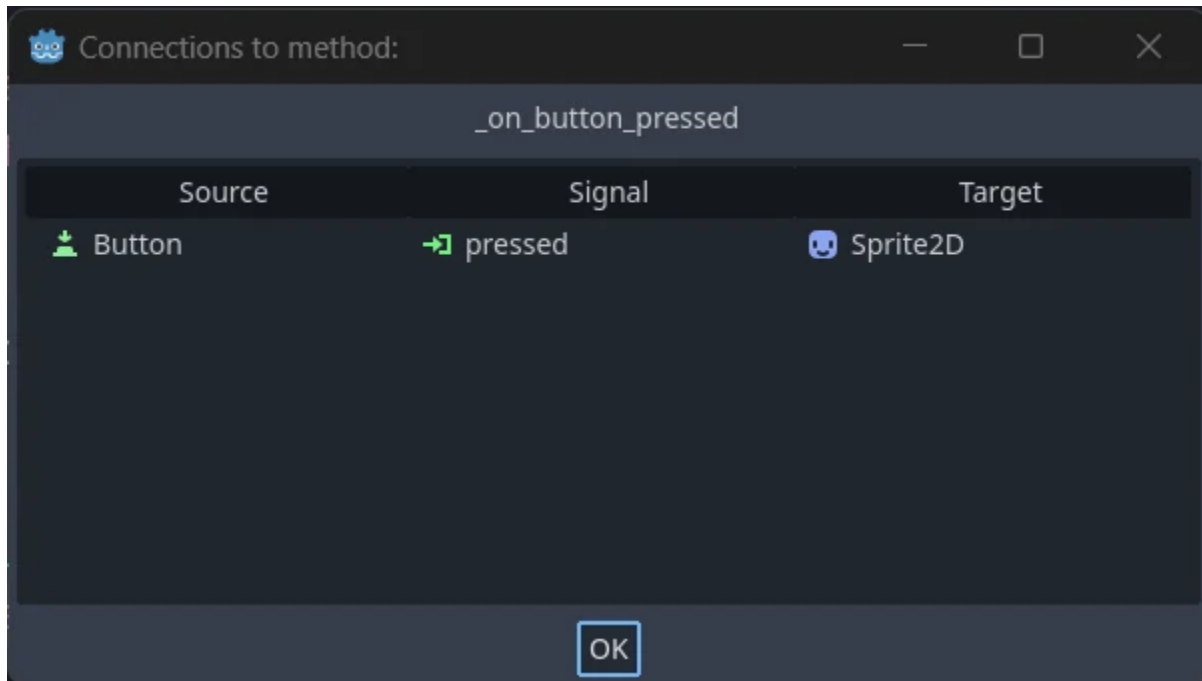


The advanced view lets you connect to any node and any built-in function, add arguments to the callback, and set options. You can toggle the mode in the window's bottom-right by clicking the Advanced button.

Click the Connect button to complete the signal connection and jump to the Script workspace. You should see the new method with a connection icon in the left margin.



If you click the icon, a window pops up and displays information about the connection. This feature is only available when connecting nodes in the editor.



Let's replace the line with the pass keyword with code that'll toggle the node's motion.

Our Sprite2D moves thanks to code in the `_process()` function. Godot provides a method to toggle processing on and off: `Node.set_process()`. Another method of the Node class, `is_processing()`, returns true if idle processing is active. We can use the `not` keyword to invert the value.

GDScript

```
func _on_button_pressed():
    set_process(not is_processing())
```

C#

```
private void OnButtonPressed()
{
    SetProcess(!IsProcessing());
}
```

This function will toggle processing and, in turn, the icon's motion on and off upon pressing the button.

Before trying the game, we need to simplify our `_process()` function to move the node automatically and not wait for user input. Replace it with the following code, which we saw two lessons ago:

GDScript

```
func _process(delta):
    rotation += angular_speed * delta
    var velocity = Vector2.UP.rotated(rotation) * speed
    position += velocity * delta
```

C#

```
public override void _Process(double delta)
{
```

(continues on next page)

(continued from previous page)

```

    Rotation += _angularSpeed * (float)delta;
    var velocity = Vector2.Up.Rotated(Rotation) * _speed;
    Position += velocity * (float)delta;
}

```

Your complete `sprite_2d.gd` code should look like the following.

GDScript

```

extends Sprite2D

var speed = 400
var angular_speed = PI

func _process(delta):
    rotation += angular_speed * delta
    var velocity = Vector2.UP.rotated(rotation) * speed
    position += velocity * delta

func _on_button_pressed():
    set_process(not is_processing())

```

C#

```

using Godot;

public partial class MySprite2D : Sprite2D
{
    private float _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Process(double delta)
    {
        Rotation += _angularSpeed * (float)delta;
        var velocity = Vector2.Up.Rotated(Rotation) * _speed;
        Position += velocity * (float)delta;
    }

    private void OnButtonPressed()
    {
        SetProcess(!IsProcessing());
    }
}

```

Run the scene now and click the button to see the sprite start and stop.

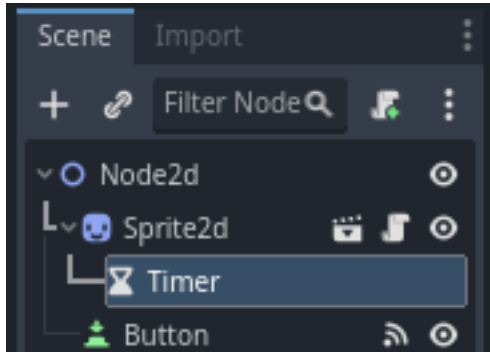
Connecting a signal via code

You can connect signals via code instead of using the editor. This is necessary when you create nodes or instantiate scenes inside of a script.

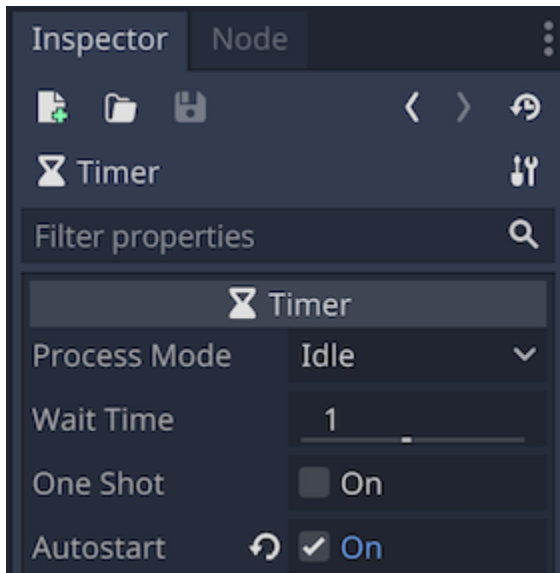
Let's use a different node here. Godot has a Timer node that's useful to implement skill cooldown times, weapon reloading, and more.

Head back to the 2D workspace. You can either click the "2D" text at the top of the window or press Ctrl + F1 (Ctrl + Cmd + 1 on macOS).

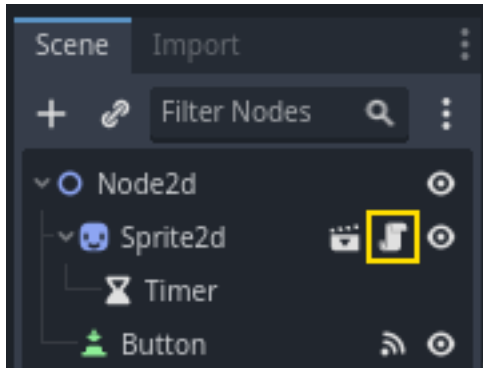
In the Scene dock, right-click on the Sprite2D node and add a new child node. Search for Timer and add the corresponding node. Your scene should now look like this.



With the Timer node selected, go to the Inspector and enable the Autostart property.



Click the script icon next to Sprite2D to jump back to the scripting workspace.



We need to do two operations to connect the nodes via code:

1. Get a reference to the Timer from the Sprite2D.
2. Call the `connect()` method on the Timer's "timeout" signal.

Note: To connect to a signal via code, you need to call the `connect()` method of the signal you want to listen to. In this case, we want to listen to the Timer's "timeout" signal.

We want to connect the signal when the scene is instantiated, and we can do that using the `Node._ready()` built-in function, which is called automatically by the engine when a node is fully instantiated.

To get a reference to a node relative to the current one, we use the method `Node.get_node()`. We can store the reference in a variable.

GDScript

```
func _ready():
    var timer = get_node("Timer")
```

C#

```
public override void _Ready()
{
    var timer = GetNode<Timer>("Timer");
}
```

The function `get_node()` looks at the Sprite2D's children and gets nodes by their name. For example, if you renamed the Timer node to "BlinkingTimer" in the editor, you would have to change the call to `get_node("BlinkingTimer")`.

We can now connect the Timer to the Sprite2D in the `_ready()` function.

GDScript

```
func _ready():
    var timer = get_node("Timer")
    timer.timeout.connect(_on_timer_timeout)
```

C#

```
public override void _Ready()
{
    var timer = GetNode<Timer>("Timer");
```

(continues on next page)

(continued from previous page)

```
timer.Timeout += OnTimerTimeout;
}
```

The line reads like so: we connect the Timer's "timeout" signal to the node to which the script is attached. When the Timer emits timeout, we want to call the function `_on_timer_timeout()`, that we need to define. Let's add it at the bottom of our script and use it to toggle our sprite's visibility.

Note: By convention, we name these callback methods in GDScript as `"_on_node_name_signal_name"` and in C# as `"OnNodeNameSignalName"`. Here, it'll be `"_on_timer_timeout"` for GDScript and `OnTimerTimeout()` for C#.

GDScript

```
func _on_timer_timeout():
    visible = not visible
```

C#

```
private void OnTimerTimeout()
{
    Visible = !Visible;
}
```

The visible property is a boolean that controls the visibility of our node. The line `visible = not visible` toggles the value. If visible is true, it becomes false, and vice-versa.

If you run the scene now, you will see that the sprite blinks on and off, at one second intervals.

Complete script

That's it for our little moving and blinking Godot icon demo! Here is the complete `sprite_2d.gd` file for reference.

GDScript

```
extends Sprite2D

var speed = 400
var angular_speed = PI

func _ready():
    var timer = get_node("Timer")
    timer.timeout.connect(_on_timer_timeout)

func _process(delta):
    rotation += angular_speed * delta
    var velocity = Vector2.UP.rotated(rotation) * speed
    position += velocity * delta

func _on_button_pressed():
```

(continues on next page)

(continued from previous page)

```

set_process(not is_processing())

func _on_timer_timeout():
    visible = not visible

```

C#

```

using Godot;

public partial class MySprite2D : Sprite2D
{
    private float _speed = 400;
    private float _angularSpeed = Mathf.Pi;

    public override void _Ready()
    {
        var timer = GetNode<Timer>("Timer");
        timer.Timeout += OnTimerTimeout;
    }

    public override void _Process(double delta)
    {
        Rotation += _angularSpeed * (float)delta;
        var velocity = Vector2.Up.Rotated(Rotation) * _speed;
        Position += velocity * (float)delta;
    }

    private void OnButtonPressed()
    {
        SetProcess(!IsProcessing());
    }

    private void OnTimerTimeout()
    {
        Visible = !Visible;
    }
}

```

Custom signals

Note: This section is a reference on how to define and use your own signals, and does not build upon the project created in previous lessons.

You can define custom signals in a script. Say, for example, that you want to show a game over screen when the player's health reaches zero. To do so, you could define a signal named "died" or "health_depleted" when their health reaches 0.

GDScript

```
extends Node2D

signal health_depleted

var health = 10
```

C#

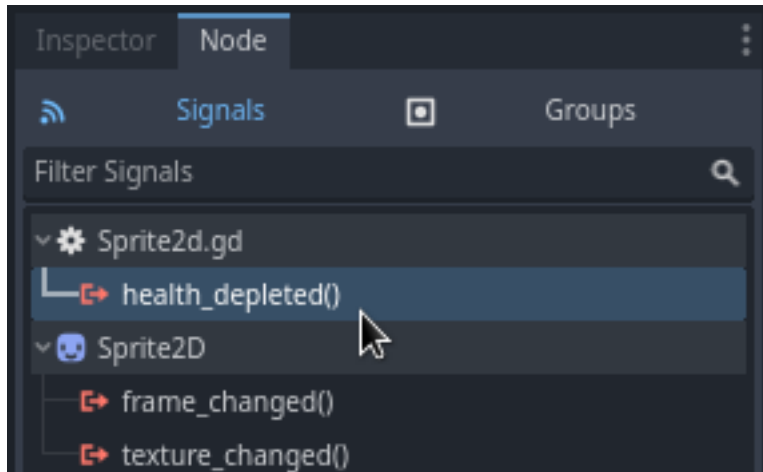
```
using Godot;

public partial class MyNode2D : Node2D
{
    [Signal]
    public delegate void HealthDepletedEventHandler();

    private int _health = 10;
}
```

Note: As signals represent events that just occurred, we generally use an action verb in the past tense in their names.

Your signals work the same way as built-in ones: they appear in the Node tab and you can connect to them like any other.



To emit a signal in your scripts, call `emit()` on the signal.

GDScript

```
func take_damage(amount):
    health -= amount
    if health <= 0:
        health_depleted.emit()
```

C#

```
public void TakeDamage(int amount)
{
```

(continues on next page)

(continued from previous page)

```

    _health -= amount;

    if (_health <= 0)
    {
        EmitSignal(SignalName.HealthDepleted);
    }
}

```

A signal can optionally declare one or more arguments. Specify the argument names between parentheses:

GDScript

```

extends Node

signal health_changed(old_value, new_value)

var health = 10

```

C#

```

using Godot;

public partial class MyNode : Node
{
    [Signal]
    public delegate void HealthChangedEventHandler(int oldValue, int newValue);

    private int _health = 10;
}

```

Note: The signal arguments show up in the editor's node dock, and Godot can use them to generate callback functions for you. However, you can still emit any number of arguments when you emit signals. So it's up to you to emit the correct values.

To emit values along with the signal, add them as extra arguments to the `emit()` function:

GDScript

```

func take_damage(amount):
    var old_health = health
    health -= amount
    health_changed.emit(old_health, health)

```

C#

```

public void TakeDamage(int amount)
{
    int oldHealth = _health;
    _health -= amount;
    EmitSignal(SignalName.HealthChanged, oldHealth, _health);
}

```

Summary

Any node in Godot emits signals when something specific happens to them, like a button being pressed. Other nodes can connect to individual signals and react to selected events.

Signals have many uses. With them, you can react to a node entering or exiting the game world, to a collision, to a character entering or leaving an area, to an element of the interface changing size, and much more.

For example, an `Area2D` representing a coin emits a `body_entered` signal whenever the player's physics body enters its collision shape, allowing you to know when the player collected it.

In the next section, Your first 2D game, you'll create a complete 2D game and put everything you learned so far into practice.

2.9 Your first 2D game

In this step-by-step tutorial series, you will create your first complete 2D game with Godot. By the end of the series, you will have a simple yet complete game of your own, like the image below.

You will learn how the Godot editor works, how to structure a project, and build a 2D game.

Note: This project is an introduction to the Godot engine. It assumes that you have some programming experience already. If you're new to programming entirely, you should start here: Scripting languages.

The game is called "Dodge the Creeps!". Your character must move and avoid the enemies for as long as possible.

You will learn to:

- Create a complete 2D game with the Godot editor.
- Structure a simple game project.
- Move the player character and change its sprite.
- Spawn random enemies.
- Count the score.

And more.

You'll find another series where you'll create a similar game but in 3D. We recommend you to start with this one, though.

Why start with 2D?

If you are new to game development or unfamiliar with Godot, we recommend starting with 2D games. This will allow you to become comfortable with both before tackling 3D games, which tend to be more complicated.

You can find a completed version of this project at this location:

- https://github.com/godotengine/godot-demo-projects/tree/master/2d/dodge_the_creeps

2.9.1 Prerequisites

This step-by-step tutorial is intended for beginners who followed the complete Getting Started.

If you're an experienced programmer, you can find the complete demo's source code here: [Dodge the Creeps source code](#).

We prepared some game assets you'll need to download so we can jump straight to the code.

You can download them by clicking the link below.

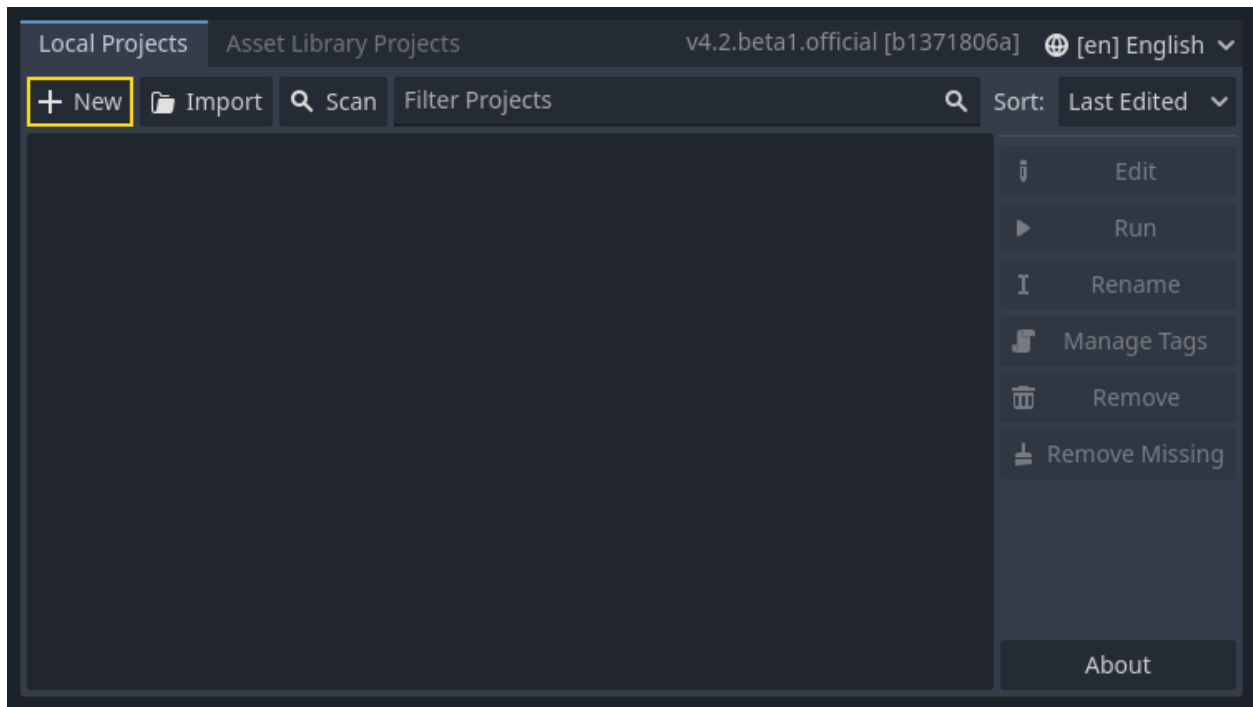
[dodge_the_creeps_2d_assets.zip](#).

2.9.2 Contents

Setting up the project

In this short first part, we'll set up and organize the project.

Launch Godot and create a new project.



When creating the new project, you only need to choose a valid Project Path. You can leave the other default settings alone.

GDScript

Download [dodge_the_creeps_2d_assets.zip](#). The archive contains the images and sounds you'll be using to make the game. Extract the archive and move the art/ and fonts/ directories to your project's directory.

C#

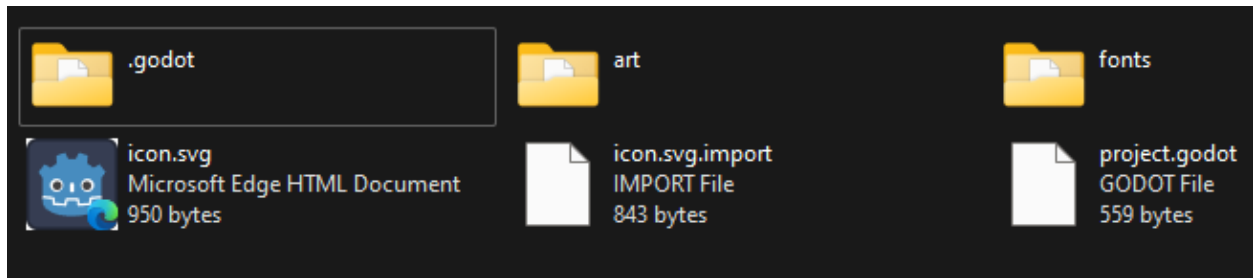
Download [dodge_the_creeps_2d_assets.zip](#). The archive contains the images and sounds you'll be using to make the game. Extract the archive and move the art/ and fonts/ directories to your project's directory.

Ensure that you have the required dependencies to use C# in Godot. You need the latest stable .NET SDK, and an editor such as VS Code. See Prerequisites.

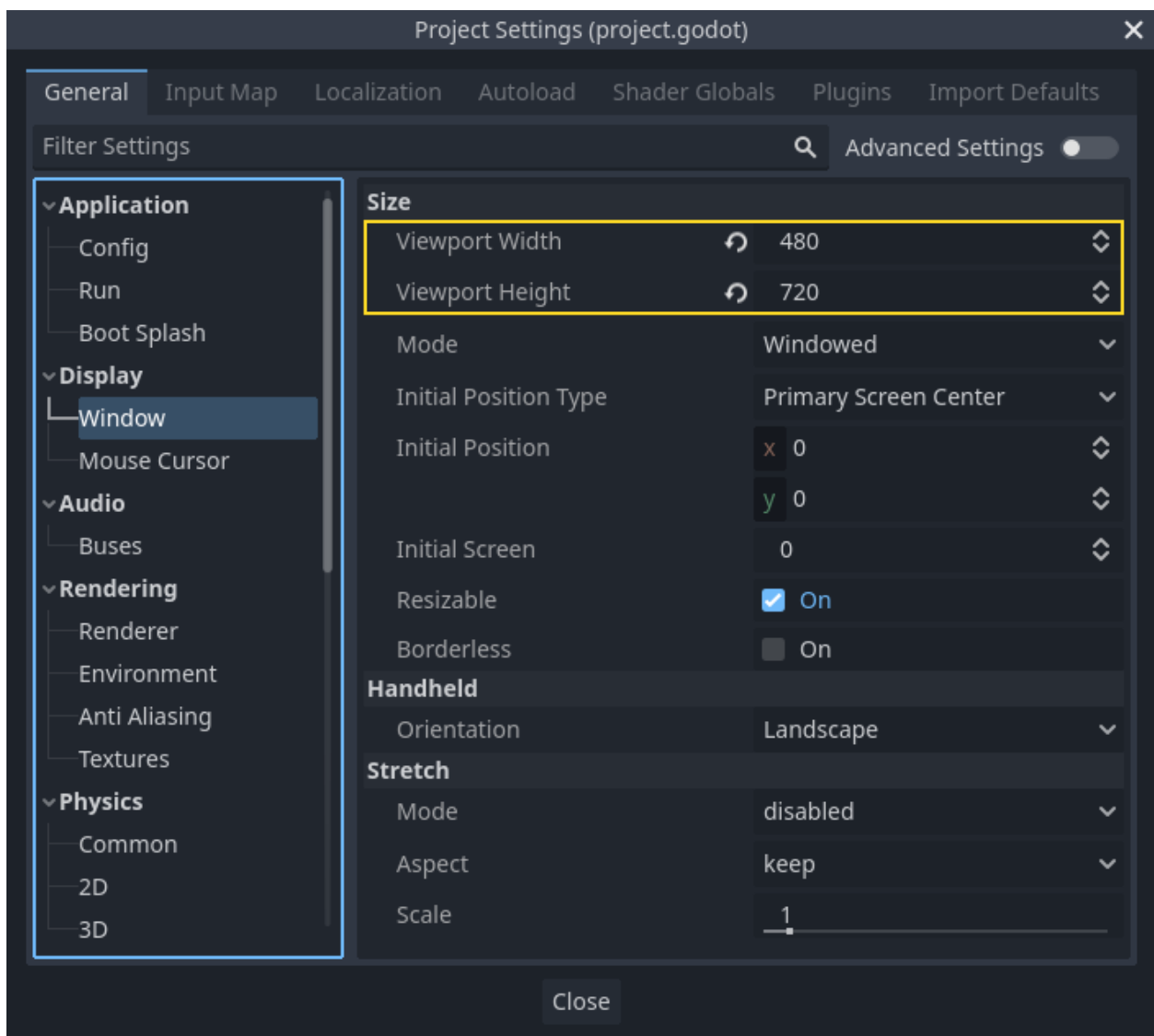
C++

The C++ part of this tutorial wasn't rewritten for the new GDExtension system yet.

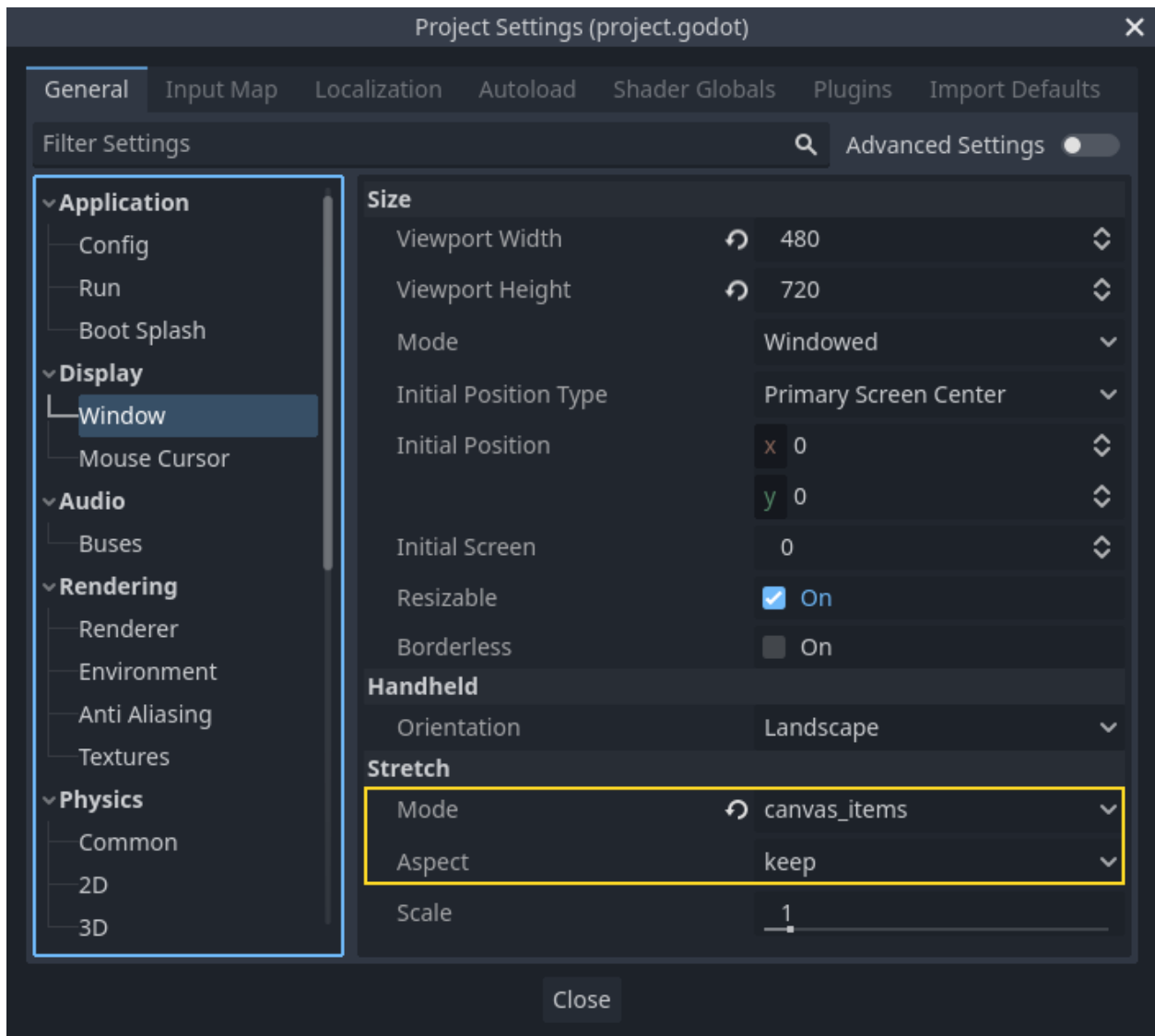
Your project folder should look like this.



This game is designed for portrait mode, so we need to adjust the size of the game window. Click on Project -> Project Settings to open the project settings window, in the left column open the Display -> Window tab. There, set "Viewport Width" to 480 and "Viewport Height" to 720.



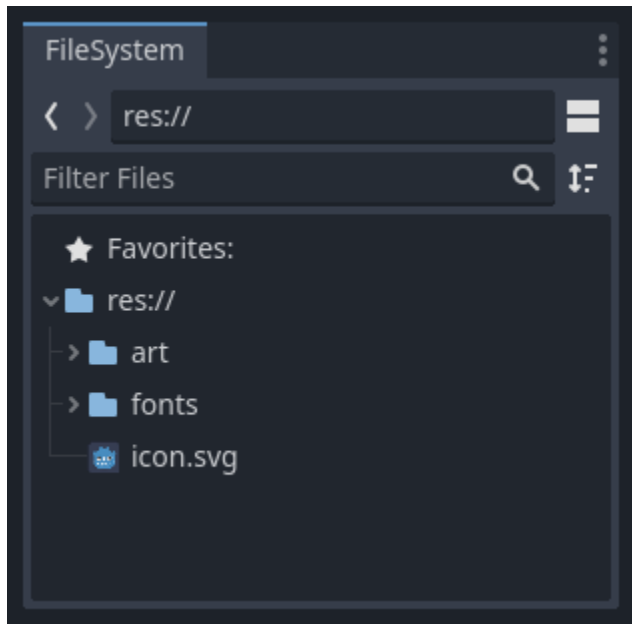
Also, under the Stretch options, set Mode to `canvas_items` and Aspect to `keep`. This ensures that the game scales consistently on different sized screens.



Organizing the project

In this project, we will make 3 independent scenes: Player, Mob, and HUD, which we will combine into the game's Main scene.

In a larger project, it might be useful to create folders to hold the various scenes and their scripts, but for this relatively small game, you can save your scenes and scripts in the project's root folder, identified by `res://`. You can see your project folders in the FileSystem dock in the lower left corner:



With the project in place, we're ready to design the player scene in the next lesson.

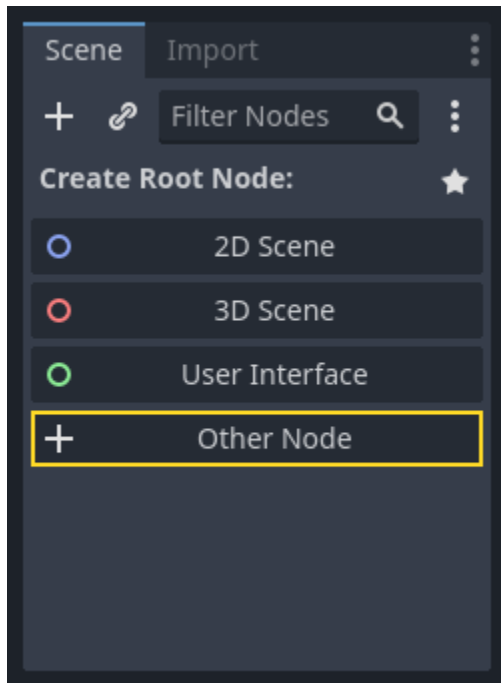
Creating the player scene

With the project settings in place, we can start working on the player-controlled character.

The first scene will define the Player object. One of the benefits of creating a separate Player scene is that we can test it separately, even before we've created other parts of the game.

Node structure

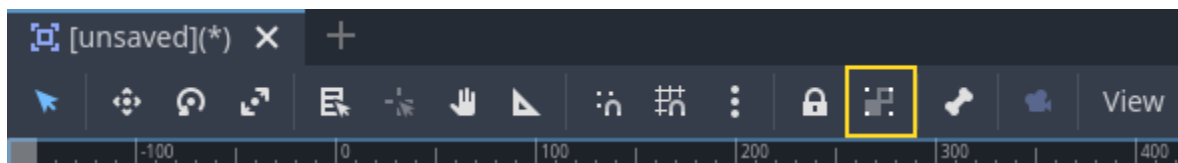
To begin, we need to choose a root node for the player object. As a general rule, a scene's root node should reflect the object's desired functionality - what the object is. Click the "Other Node" button and add an Area2D node to the scene.



Godot will display a warning icon next to the node in the scene tree. You can ignore it for now. We will address it later.

With `Area2D` we can detect objects that overlap or run into the player. Change the node's name to `Player` by double-clicking on it. Now that we've set the scene's root node, we can add additional nodes to give it more functionality.

Before we add any children to the `Player` node, we want to make sure we don't accidentally move or resize them by clicking on them. Select the node and click the icon to the right of the lock. Its tooltip says "Make selected node's children not selectable."



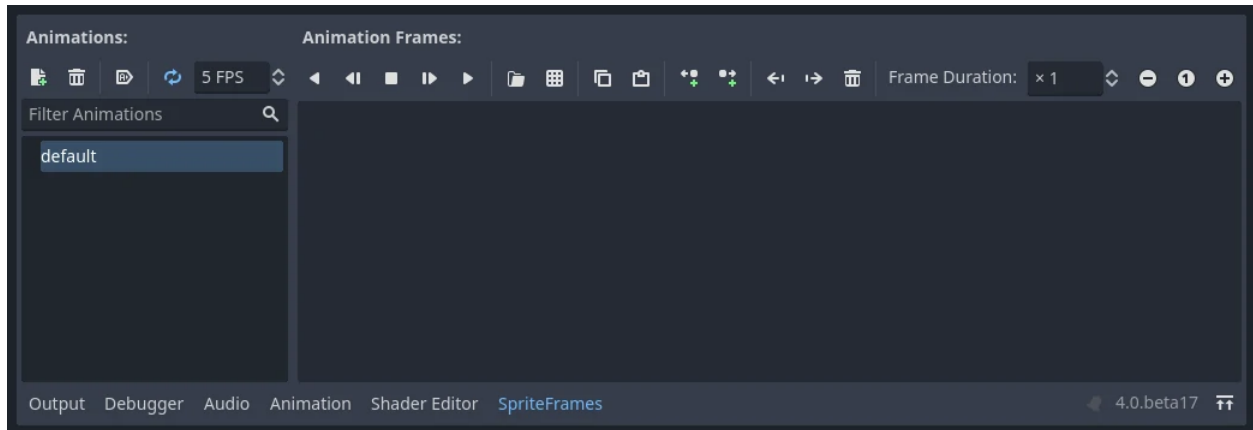
Save the scene. Click `Scene -> Save`, or press `Ctrl + S` on Windows/Linux or `Cmd + S` on macOS.

Note: For this project, we will be following the Godot naming conventions.

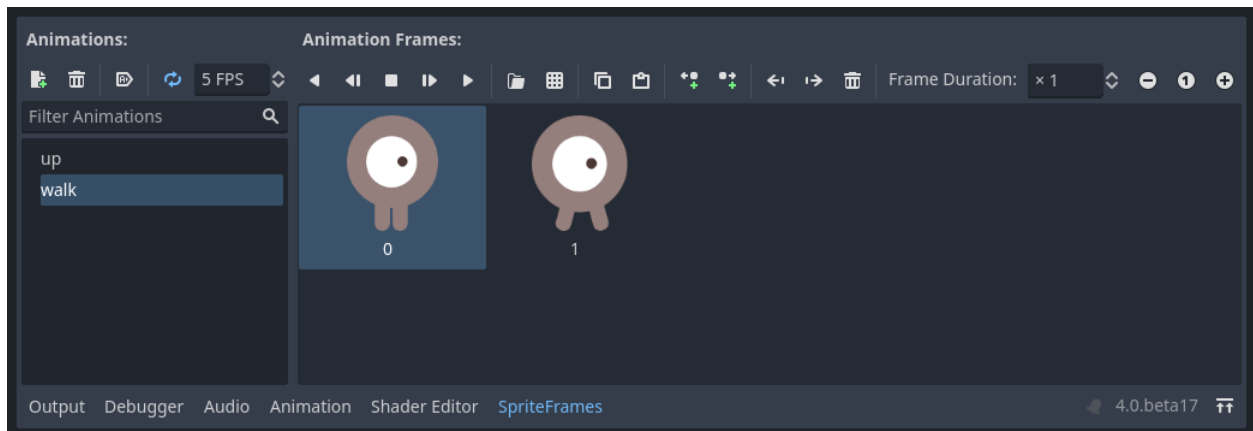
- **GScript:** Classes (nodes) use `PascalCase`, variables and functions use `snake_case`, and constants use `ALL_CAPS` (See `GScript` style guide).
- **C#:** Classes, export variables and methods use `PascalCase`, private fields use `_camelCase`, local variables and parameters use `camelCase` (See `C#` style guide). Be careful to type the method names precisely when connecting signals.

Sprite animation

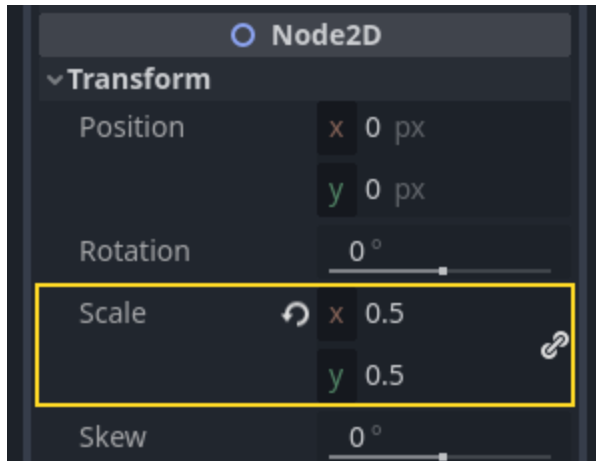
Click on the Player node and add (Ctrl + A on Windows/Linux or Cmd + A on macOS) a child node AnimatedSprite2D. The AnimatedSprite2D will handle the appearance and animations for our player. Notice that there is a warning symbol next to the node. An AnimatedSprite2D requires a SpriteFrames resource, which is a list of the animations it can display. To create one, find the Sprite Frames property under the Animation tab in the Inspector and click "[empty]" -> "New SpriteFrames". Click again to open the "SpriteFrames" panel:



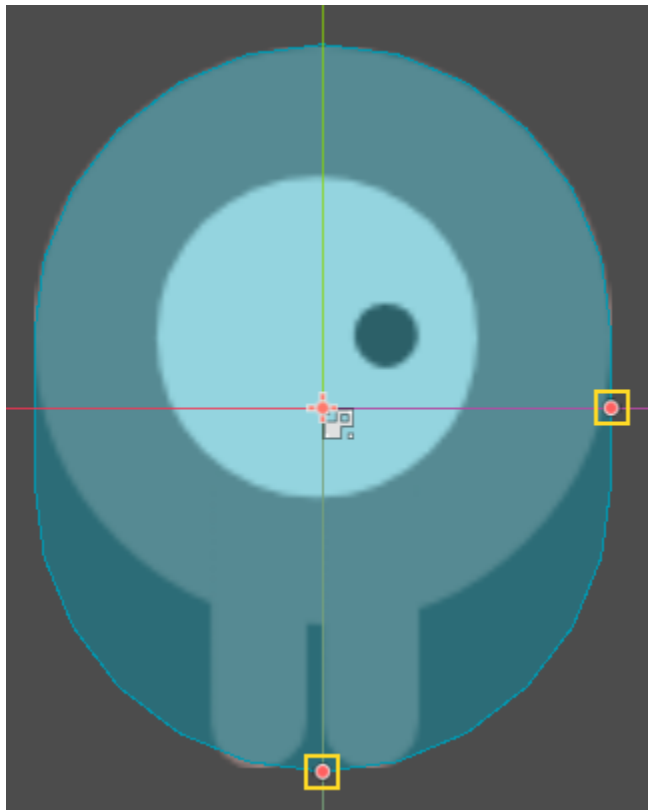
On the left is a list of animations. Click the "default" one and rename it to "walk". Then click the "Add Animation" button to create a second animation named "up". Find the player images in the "FileSystem" tab - they're in the art folder you unzipped earlier. Drag the two images for each animation, named playerGrey_up[1/2] and playerGrey_walk[1/2], into the "Animation Frames" side of the panel for the corresponding animation:



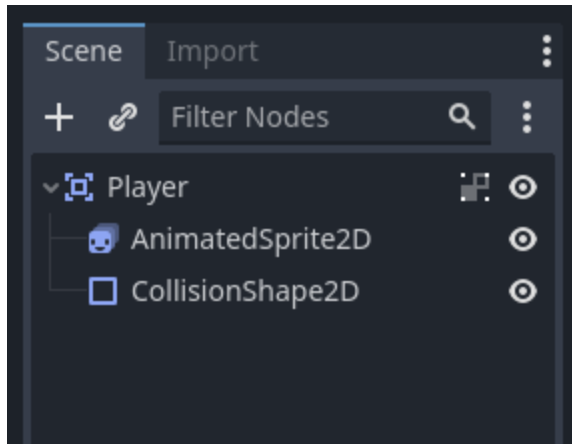
The player images are a bit too large for the game window, so we need to scale them down. Click on the AnimatedSprite2D node and set the Scale property to (0.5, 0.5). You can find it in the Inspector under the Node2D heading.



Finally, add a `CollisionShape2D` as a child of `Player`. This will determine the player's "hit box", or the bounds of its collision area. For this character, a `CapsuleShape2D` node gives the best fit, so next to "Shape" in the Inspector, click "[empty]" -> "New CapsuleShape2D". Using the two size handles, resize the shape to cover the sprite:



When you're finished, your `Player` scene should look like this:



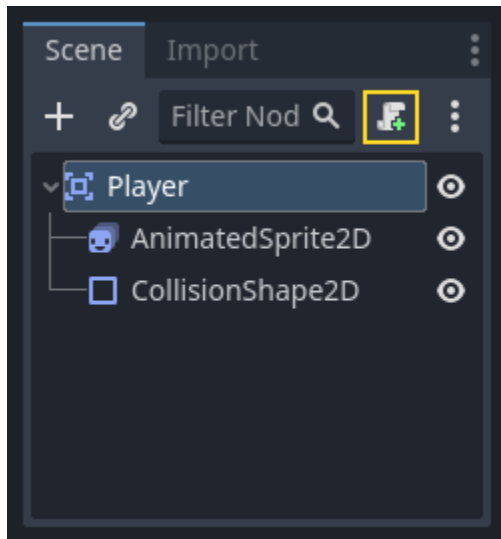
Make sure to save the scene again after these changes.

In the next part, we'll add a script to the player node to move and animate it. Then, we'll set up collision detection to know when the player got hit by something.

Coding the player

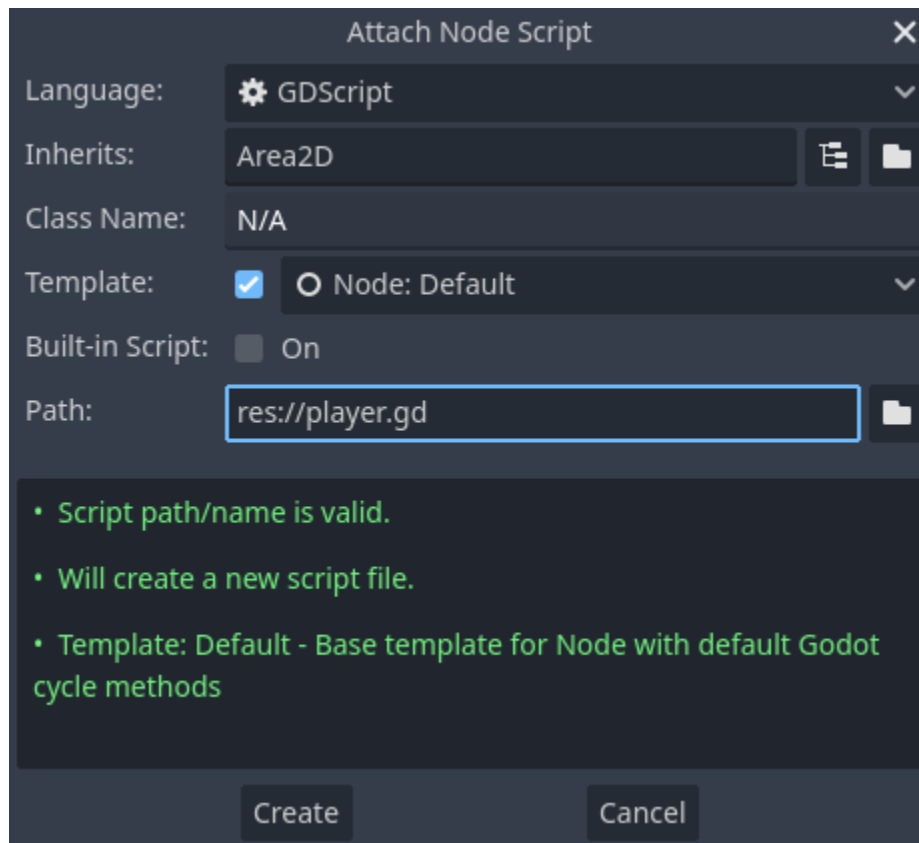
In this lesson, we'll add player movement, animation, and set it up to detect collisions.

To do so, we need to add some functionality that we can't get from a built-in node, so we'll add a script. Click the Player node and click the "Attach Script" button:



In the script settings window, you can leave the default settings alone. Just click "Create":

Note: If you're creating a `C#` script or other languages, select the language from the language drop down menu before hitting create.



Note: If this is your first time encountering GDScript, please read [Scripting languages](#) before continuing.

Start by declaring the member variables this object will need:

GDScript

```
extends Area2D

@export var speed = 400 # How fast the player will move (pixels/sec).
var screen_size # Size of the game window.
```

C#

```
using Godot;

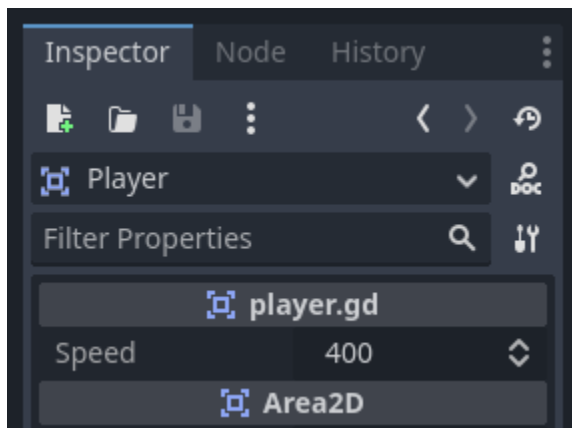
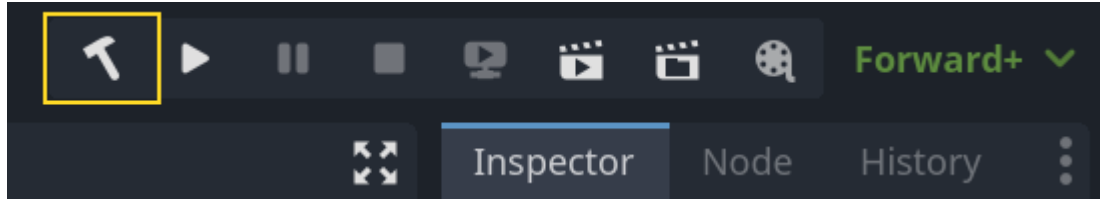
public partial class Player : Area2D
{
    [Export]
    public int Speed { get; set; } = 400; // How fast the player will move (pixels/sec).

    public Vector2 ScreenSize; // Size of the game window.
}
```

Using the export keyword on the first variable speed allows us to set its value in the Inspector. This can be handy for values that you want to be able to adjust just like a node's built-in properties. Click on the Player node and you'll see the property now appears in the "Script Variables" section of the Inspector. Remember,

if you change the value here, it will override the value written in the script.

Warning: If you're using C#, you need to (re)build the project assemblies whenever you want to see new export variables or signals. This build can be manually triggered by clicking the Build button at the top right of the editor.



Your `player.gd` script should already contain a `_ready()` and a `_process()` function. If you didn't select the default template shown above, create these functions while following the lesson.

The `_ready()` function is called when a node enters the scene tree, which is a good time to find the size of the game window:

GDScript

```
func _ready():
    screen_size = get_viewport_rect().size
```

C#

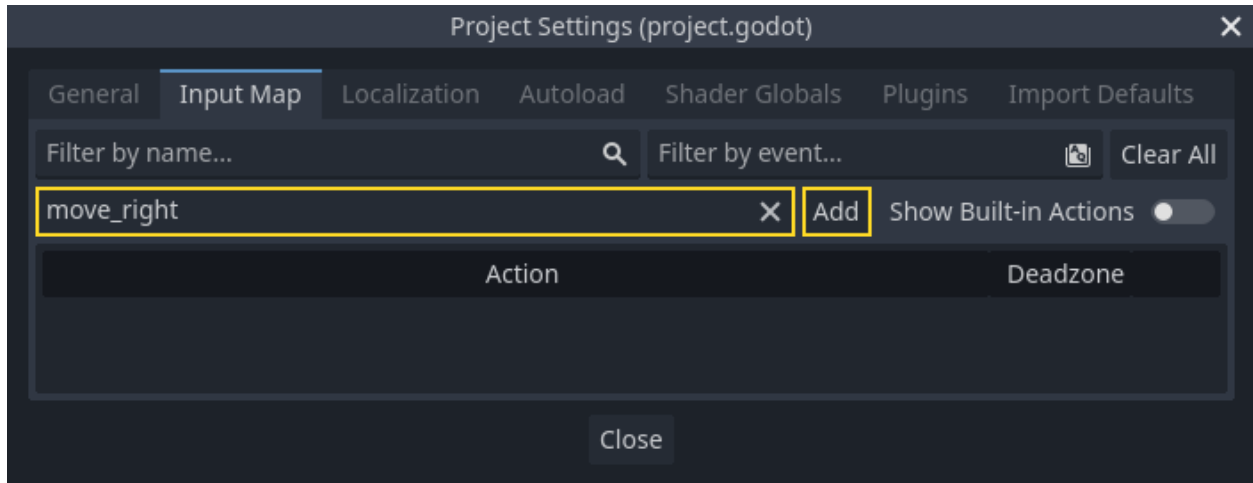
```
public override void _Ready()
{
    ScreenSize = GetViewportRect().Size;
}
```

Now we can use the `_process()` function to define what the player will do. `_process()` is called every frame, so we'll use it to update elements of our game, which we expect will change often. For the player, we need to do the following:

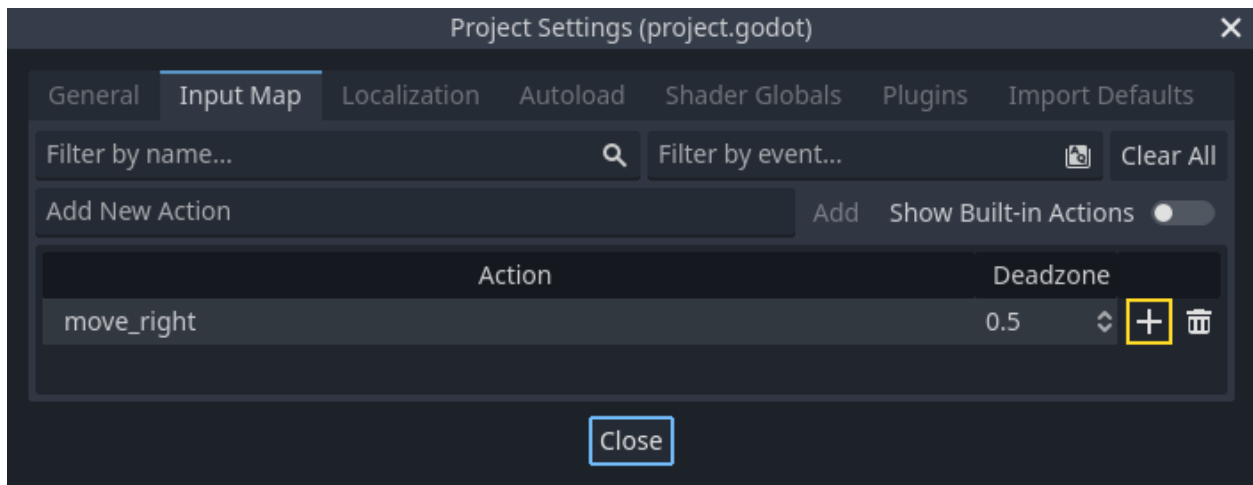
- Check for input.
- Move in the given direction.
- Play the appropriate animation.

First, we need to check for input - is the player pressing a key? For this game, we have 4 direction inputs to check. Input actions are defined in the Project Settings under "Input Map". Here, you can define custom events and assign different keys, mouse events, or other inputs to them. For this game, we will map the arrow keys to the four directions.

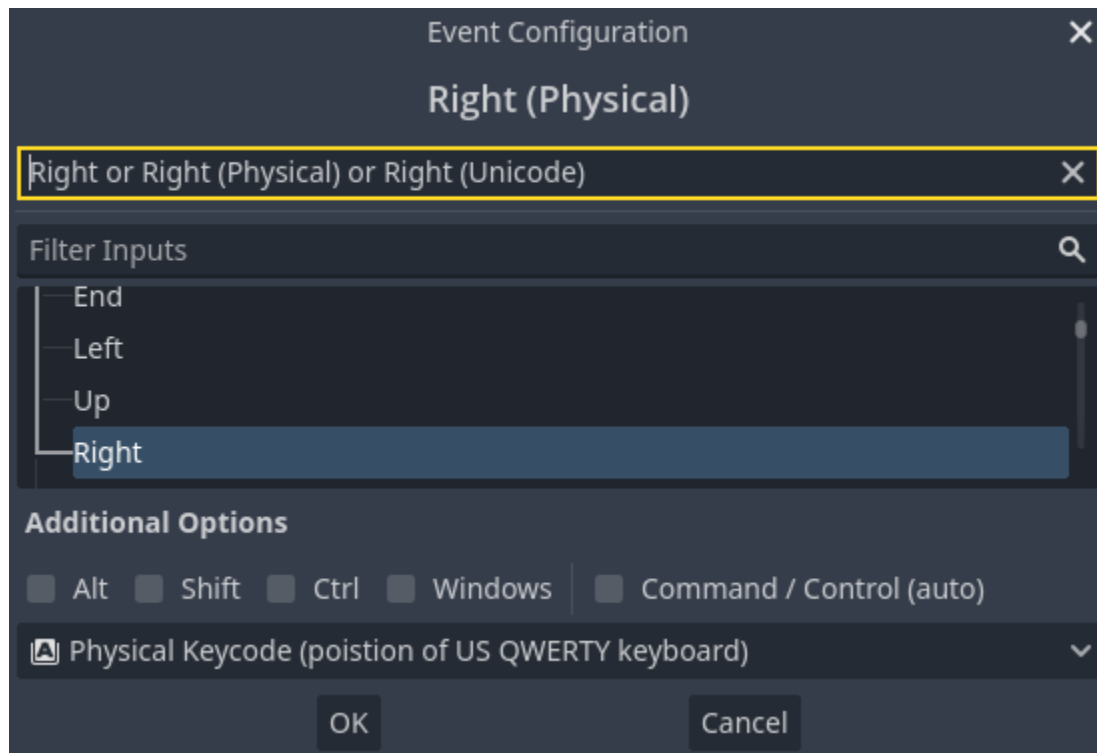
Click on Project -> Project Settings to open the project settings window and click on the Input Map tab at the top. Type "move_right" in the top bar and click the "Add" button to add the move_right action.



We need to assign a key to this action. Click the "+" icon on the right, to open the event manager window.



The "Listening for Input..." field should automatically be selected. Press the "right" key on your keyboard, and the menu should look like this now.

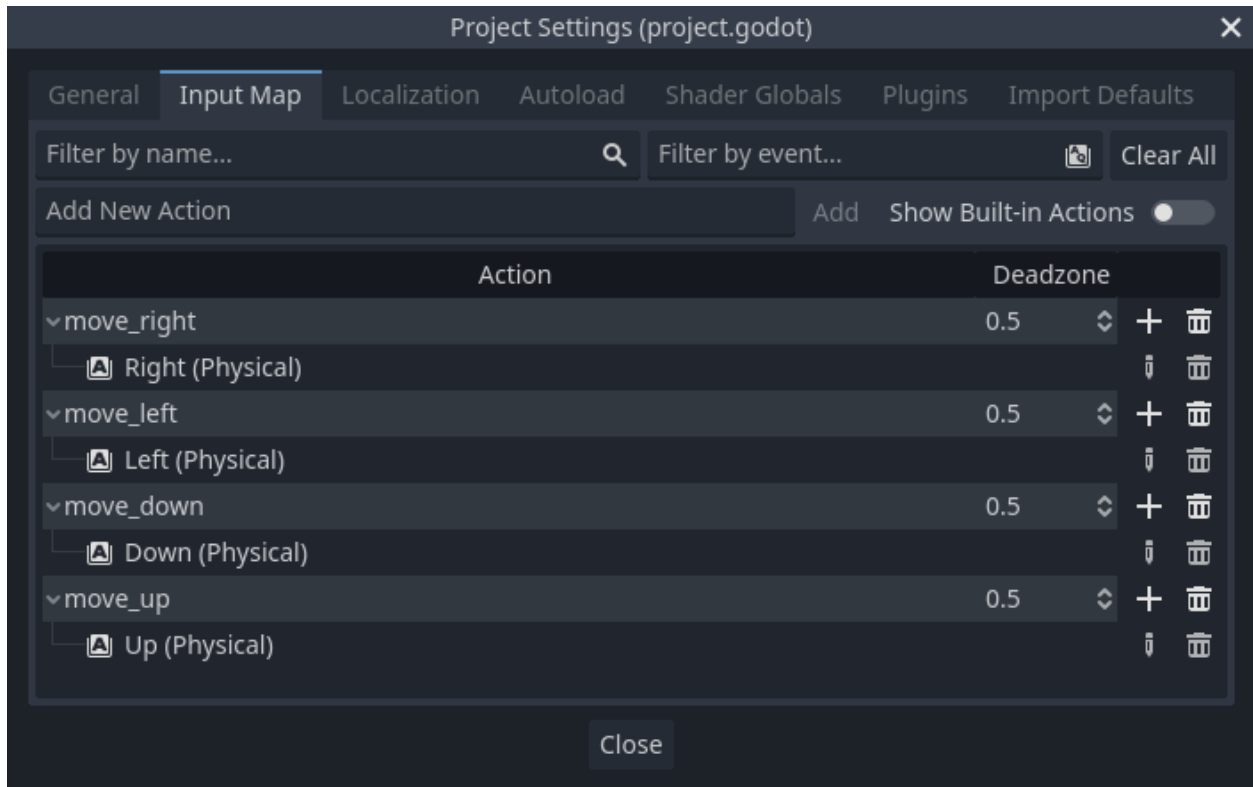


Select the "ok" button. The "right" key is now associated with the `move_right` action.

Repeat these steps to add three more mappings:

1. `move_left` mapped to the left arrow key.
2. `move_up` mapped to the up arrow key.
3. And `move_down` mapped to the down arrow key.

Your input map tab should look like this:



Click the "Close" button to close the project settings.

Note: We only mapped one key to each input action, but you can map multiple keys, joystick buttons, or mouse buttons to the same input action.

You can detect whether a key is pressed using `Input.is_action_pressed()`, which returns true if it's pressed or false if it isn't.

GDScript

```
func _process(delta):
    var velocity = Vector2.ZERO # The player's movement vector.
    if Input.is_action_pressed("move_right"):
        velocity.x += 1
    if Input.is_action_pressed("move_left"):
        velocity.x -= 1
    if Input.is_action_pressed("move_down"):
        velocity.y += 1
    if Input.is_action_pressed("move_up"):
        velocity.y -= 1

    if velocity.length() > 0:
        velocity = velocity.normalized() * speed
        $AnimatedSprite2D.play()
    else:
        $AnimatedSprite2D.stop()
```

C#

```
public override void _Process(double delta)
{
    var velocity = Vector2.Zero; // The player's movement vector.

    if (Input.IsActionPressed("move_right"))
    {
        velocity.X += 1;
    }

    if (Input.IsActionPressed("move_left"))
    {
        velocity.X -= 1;
    }

    if (Input.IsActionPressed("move_down"))
    {
        velocity.Y += 1;
    }

    if (Input.IsActionPressed("move_up"))
    {
        velocity.Y -= 1;
    }

    var animatedSprite2D = GetNode<AnimatedSprite2D>("AnimatedSprite2D");

    if (velocity.Length() > 0)
    {
        velocity = velocity.Normalized() * Speed;
        animatedSprite2D.Play();
    }
    else
    {
        animatedSprite2D.Stop();
    }
}
```

We start by setting the velocity to (0, 0) - by default, the player should not be moving. Then we check each input and add/subtract from the velocity to obtain a total direction. For example, if you hold right and down at the same time, the resulting velocity vector will be (1, 1). In this case, since we're adding a horizontal and a vertical movement, the player would move faster diagonally than if it just moved horizontally.

We can prevent that if we normalize the velocity, which means we set its length to 1, then multiply by the desired speed. This means no more fast diagonal movement.

Tip: If you've never used vector math before, or need a refresher, you can see an explanation of vector usage in Godot at [Vector math](#). It's good to know but won't be necessary for the rest of this tutorial.

We also check whether the player is moving so we can call `play()` or `stop()` on the `AnimatedSprite2D`.

Tip: `$` is shorthand for `get_node()`. So in the code above, `$AnimatedSprite2D.play()` is the same as `get_node("AnimatedSprite2D").play()`.

In GDScript, `$` returns the node at the relative path from the current node, or returns null if the node is not found. Since `AnimatedSprite2D` is a child of the current node, we can use `$AnimatedSprite2D`.

Now that we have a movement direction, we can update the player's position. We can also use `clamp()` to prevent it from leaving the screen. Clamping a value means restricting it to a given range. Add the following to the bottom of the `_process` function (make sure it's not indented under the `else`):

GDScript

```
position += velocity * delta
position = position.clamp(Vector2.ZERO, screen_size)
```

C#

```
Position += velocity * (float)delta;
Position = new Vector2(
    Mathf.Clamp(Position.X, 0, ScreenSize.X),
    Mathf.Clamp(Position.Y, 0, ScreenSize.Y)
);
```

Tip: The `delta` parameter in the `_process()` function refers to the frame length - the amount of time that the previous frame took to complete. Using this value ensures that your movement will remain consistent even if the frame rate changes.

Click "Play Scene" (F6, Cmd + R on macOS) and confirm you can move the player around the screen in all directions.

Warning: If you get an error in the "Debugger" panel that says
Attempt to call function 'play' in base 'null instance' on a null instance
this likely means you spelled the name of the `AnimatedSprite2D` node wrong. Node names are case-sensitive and `$NodeName` must match the name you see in the scene tree.

Choosing animations

Now that the player can move, we need to change which animation the `AnimatedSprite2D` is playing based on its direction. We have the "walk" animation, which shows the player walking to the right. This animation should be flipped horizontally using the `flip_h` property for left movement. We also have the "up" animation, which should be flipped vertically with `flip_v` for downward movement. Let's place this code at the end of the `_process()` function:

GDScript

```
if velocity.x != 0:
    $AnimatedSprite2D.animation = "walk"
    $AnimatedSprite2D.flip_v = false
    # See the note below about boolean assignment.
    $AnimatedSprite2D.flip_h = velocity.x < 0
elif velocity.y != 0:
    $AnimatedSprite2D.animation = "up"
    $AnimatedSprite2D.flip_v = velocity.y > 0
```

C#

```
if (velocity.X != 0)
{
    animatedSprite2D.Animation = "walk";
    animatedSprite2D.FlipV = false;
    // See the note below about boolean assignment.
    animatedSprite2D.FlipH = velocity.X < 0;
}
else if (velocity.Y != 0)
{
    animatedSprite2D.Animation = "up";
    animatedSprite2D.FlipV = velocity.Y > 0;
}
```

Note: The boolean assignments in the code above are a common shorthand for programmers. Since we're doing a comparison test (boolean) and also assigning a boolean value, we can do both at the same time. Consider this code versus the one-line boolean assignment above:

GDScript

```
if velocity.x < 0:
    $AnimatedSprite2D.flip_h = true
else:
    $AnimatedSprite2D.flip_h = false
```

C#

```
if (velocity.X < 0)
{
    animatedSprite2D.FlipH = true;
}
else
{
    animatedSprite2D.FlipH = false;
}
```

Play the scene again and check that the animations are correct in each of the directions.

Tip: A common mistake here is to type the names of the animations wrong. The animation names in the SpriteFrames panel must match what you type in the code. If you named the animation "Walk", you must also use a capital "W" in the code.

When you're sure the movement is working correctly, add this line to `_ready()`, so the player will be hidden when the game starts:

GDScript

```
hide()
```

C#


```
Hide();
```

Preparing for collisions

We want Player to detect when it's hit by an enemy, but we haven't made any enemies yet! That's OK, because we're going to use Godot's signal functionality to make it work.

Add the following at the top of the script. If you're using GDScript, add it after `extends Area2D`. If you're using C#, add it after `public partial class Player : Area2D`:

GDScript

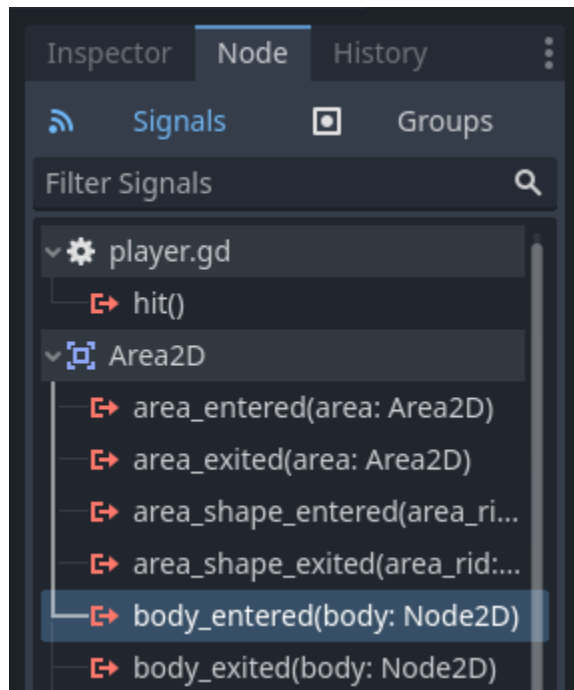
```
signal hit
```

C#

```
// Don't forget to rebuild the project so the editor knows about the new signal.
```

```
[Signal]
public delegate void HitEventHandler();
```

This defines a custom signal called "hit" that we will have our player emit (send out) when it collides with an enemy. We will use `Area2D` to detect the collision. Select the Player node and click the "Node" tab next to the Inspector tab to see the list of signals the player can emit:



Notice our custom "hit" signal is there as well! Since our enemies are going to be `RigidBody2D` nodes, we want the `body_entered(body: Node2D)` signal. This signal will be emitted when a body contacts the player. Click "Connect.." and the "Connect a Signal" window appears.

Godot will create a function with that exact name directly in script for you. You don't need to change the default settings right now.

Warning: If you're using an external text editor (for example, Visual Studio Code), a bug currently prevents Godot from doing so. You'll be sent to your external editor, but the new function won't be there.

In this case, you'll need to write the function yourself into the Player's script file.

```
→ 8 func _on_body_entered(body):  
  9     >| pass # Replace with function body.  
 10
```

Note the green icon indicating that a signal is connected to this function; this does not mean the function exists, only that the signal will attempt to connect to a function with that name, so double-check that the spelling of the function matches exactly!

Next, add this code to the function:

GDScript

```
func _on_body_entered(body):  
    hide() # Player disappears after being hit.  
    hit.emit()  
    # Must be deferred as we can't change physics properties on a physics callback.  
    $CollisionShape2D.set_deferred("disabled", true)
```

C#

```
private void OnBodyEntered(Node2D body)  
{  
    Hide(); // Player disappears after being hit.  
    EmitSignal(SignalName.Hit);  
    // Must be deferred as we can't change physics properties on a physics callback.  
    GetNode<CollisionShape2D>("CollisionShape2D").SetDeferred(CollisionShape2D.PropertyName.  
    Disabled, true);  
}
```

Each time an enemy hits the player, the signal is going to be emitted. We need to disable the player's collision so that we don't trigger the hit signal more than once.

Note: Disabling the area's collision shape can cause an error if it happens in the middle of the engine's collision processing. Using `set_deferred()` tells Godot to wait to disable the shape until it's safe to do so.

The last piece is to add a function we can call to reset the player when starting a new game.

GDScript

```
func start(pos):  
    position = pos  
    show()  
    $CollisionShape2D.disabled = false
```

C#

```

public void Start(Vector2 position)
{
    Position = position;
    Show();
    GetNode<CollisionShape2D>("CollisionShape2D").Disabled = false;
}

```

With the player working, we'll work on the enemy in the next lesson.

Creating the enemy

Now it's time to make the enemies our player will have to dodge. Their behavior will not be very complex: mobs will spawn randomly at the edges of the screen, choose a random direction, and move in a straight line.

We'll create a Mob scene, which we can then instance to create any number of independent mobs in the game.

Node setup

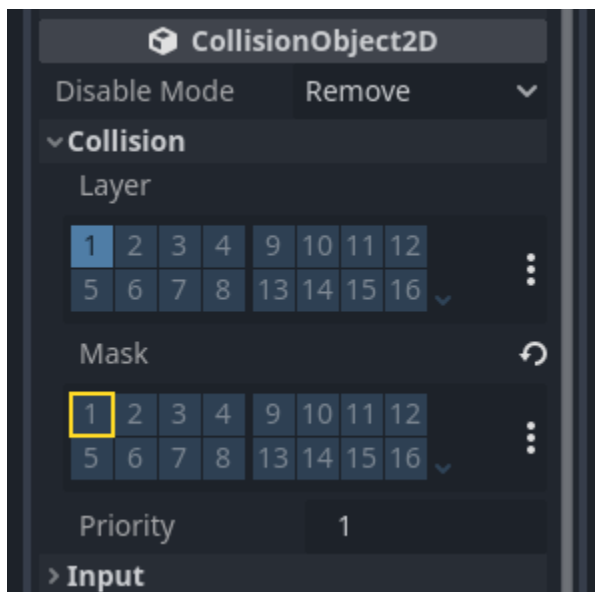
Click Scene -> New Scene from the top menu and add the following nodes:

- RigidBody2D (named Mob)
 - AnimatedSprite2D
 - CollisionShape2D
 - VisibleOnScreenNotifier2D

Don't forget to set the children so they can't be selected, like you did with the Player scene.

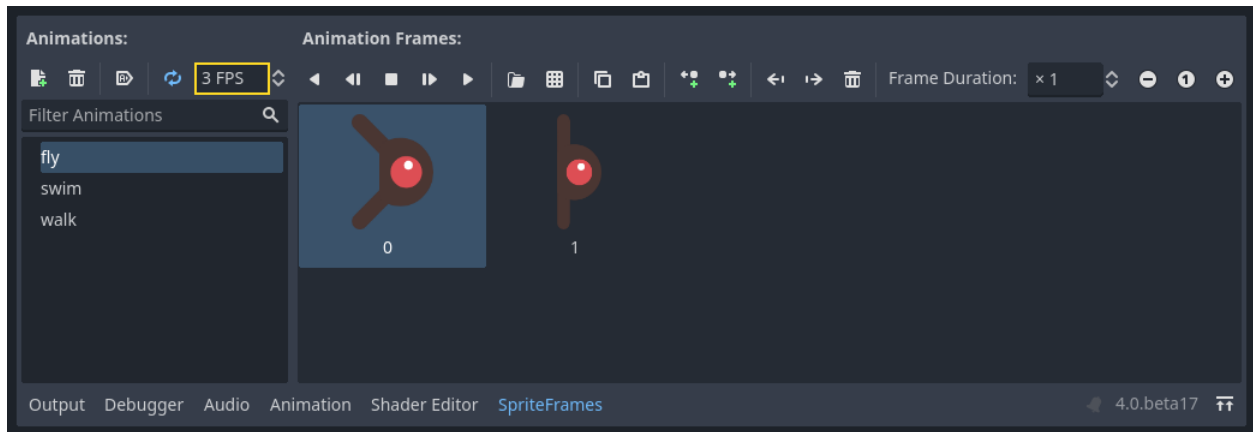
Select the Mob node and set it's Gravity Scale property in the RigidBody2D section of the inspector to 0. This will prevent the mob from falling downwards.

In addition, under the CollisionObject2D section just beneath the RigidBody2D section, expand the Collision group and uncheck the 1 inside the Mask property. This will ensure the mobs do not collide with each other.



Set up the AnimatedSprite2D like you did for the player. This time, we have 3 animations: fly, swim, and walk. There are two images for each animation in the art folder.

The Animation Speed property has to be set for each individual animation. Adjust it to 3 for all 3 animations.



You can use the "Play Animation" buttons on the right of the Animation Speed input field to preview your animations.

We'll select one of these animations randomly so that the mobs will have some variety.

Like the player images, these mob images need to be scaled down. Set the AnimatedSprite2D's Scale property to (0.75, 0.75).

As in the Player scene, add a CapsuleShape2D for the collision. To align the shape with the image, you'll need to set the Rotation property to 90 (under "Transform" in the Inspector).

Save the scene.

Enemy script

Add a script to the Mob like this:

GDScript

```
extends RigidBody2D
```

C#

```
using Godot;

public partial class Mob : RigidBody2D
{
    // Don't forget to rebuild the project.
}
```

Now let's look at the rest of the script. In `_ready()` we play the animation and randomly choose one of the three animation types:

GDScript

```
func _ready():
    var mob_types = $AnimatedSprite2D.sprite_frames.get_animation_names()
    $AnimatedSprite2D.play(mob_types[randi() % mob_types.size()])
```

C#

```
public override void _Ready()
{
    var animatedSprite2D = GetNode<AnimatedSprite2D>("AnimatedSprite2D");
    string[] mobTypes = animatedSprite2D.SpriteFrames.GetAnimationNames();
    animatedSprite2D.Play(mobTypes[GD.Randi() % mobTypes.Length]);
}
```

First, we get the list of animation names from the AnimatedSprite2D's `sprite_frames` property. This returns an Array containing all three animation names: ["walk", "swim", "fly"].

We then need to pick a random number between 0 and 2 to select one of these names from the list (array indices start at 0). `randi() % n` selects a random integer between 0 and n-1.

The last piece is to make the mobs delete themselves when they leave the screen. Connect the `screen_exited()` signal of the VisibleOnScreenNotifier2D node to the Mob and add this code:

GDScript

```
func _on_visible_on_screen_notifier_2d_screen_exited():
    queue_free()
```

C#

```
private void OnVisibleOnScreenNotifier2DScreenExited()
{
    QueueFree();
}
```

This completes the Mob scene.

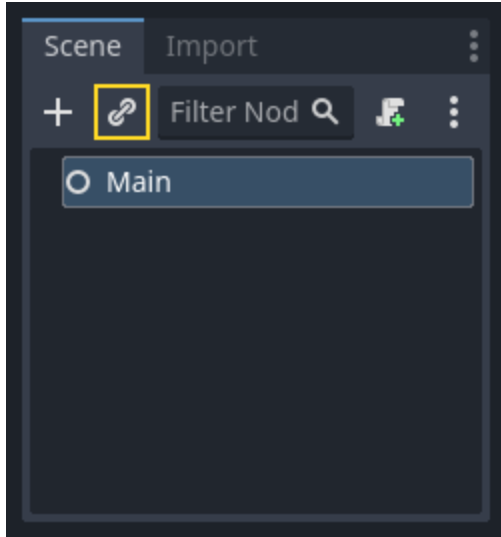
With the player and enemies ready, in the next part, we'll bring them together in a new scene. We'll make enemies spawn randomly around the game board and move forward, turning our project into a playable game.

The main game scene

Now it's time to bring everything we did together into a playable game scene.

Create a new scene and add a Node named Main. (The reason we are using Node instead of Node2D is because this node will be a container for handling game logic. It does not require 2D functionality itself.)

Click the Instance button (represented by a chain link icon) and select your saved `player.tscn`.



Now, add the following nodes as children of Main, and name them as shown (values are in seconds):

- Timer (named MobTimer) - to control how often mobs spawn
- Timer (named ScoreTimer) - to increment the score every second
- Timer (named StartTimer) - to give a delay before starting
- Marker2D (named StartPosition) - to indicate the player's start position

Set the Wait Time property of each of the Timer nodes as follows:

- MobTimer: 0.5
- ScoreTimer: 1
- StartTimer: 2

In addition, set the One Shot property of StartTimer to "On" and set Position of the StartPosition node to (240, 450).

Spawning mobs

The Main node will be spawning new mobs, and we want them to appear at a random location on the edge of the screen. Add a Path2D node named MobPath as a child of Main. When you select Path2D, you will see some new buttons at the top of the editor:



Select the middle one ("Add Point") and draw the path by clicking to add the points at the corners shown. To have the points snap to the grid, make sure "Use Grid Snap" and "Use Smart Snap" are both selected. These options can be found to the left of the "Lock" button, appearing as a magnet next to some dots and intersecting lines, respectively.

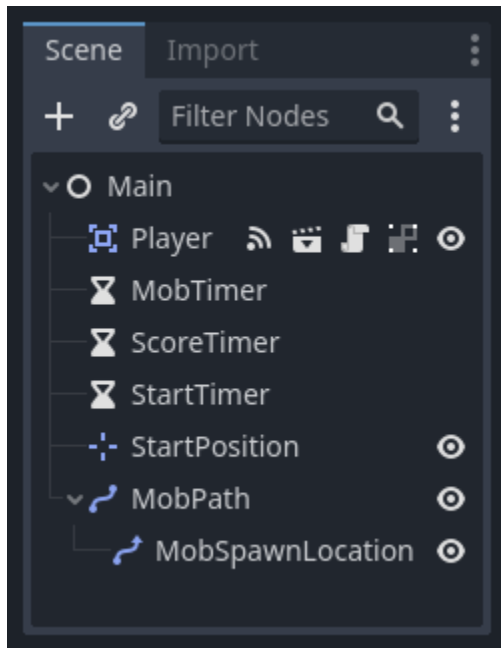


Important: Draw the path in clockwise order, or your mobs will spawn pointing outwards instead of inwards!

After placing point 4 in the image, click the "Close Curve" button and your curve will be complete.

Now that the path is defined, add a PathFollow2D node as a child of MobPath and name it MobSpawnLocation. This node will automatically rotate and follow the path as it moves, so we can use it to select a random position and direction along the path.

Your scene should look like this:



Main script

Add a script to Main. At the top of the script, we use `@export var mob_scene: PackedScene` to allow us to choose the Mob scene we want to instance.

GDScript

```
extends Node

@export var mob_scene: PackedScene
var score
```

C#

```
using Godot;

public partial class Main : Node
{
    // Don't forget to rebuild the project so the editor knows about the new export variable.

    [Export]
    public PackedScene MobScene { get; set; }
}
```

(continues on next page)

(continued from previous page)

```
private int _score;
}
```

Click the Main node and you will see the Mob Scene property in the Inspector under "Script Variables".

You can assign this property's value in two ways:

- Drag mob.tscn from the "FileSystem" dock and drop it in the Mob Scene property.
- Click the down arrow next to "[empty]" and choose "Load". Select mob.tscn.

Next, select the instance of the Player scene under Main node in the Scene dock, and access the Node dock on the sidebar. Make sure to have the Signals tab selected in the Node dock.

You should see a list of the signals for the Player node. Find and double-click the hit signal in the list (or right-click it and select "Connect..."). This will open the signal connection dialog. We want to make a new function named game_over, which will handle what needs to happen when a game ends. Type "game_over" in the "Receiver Method" box at the bottom of the signal connection dialog and click "Connect". You are aiming to have the hit signal emitted from Player and handled in the Main script. Add the following code to the new function, as well as a new_game function that will set everything up for a new game:

GDScript

```
func game_over():
    $ScoreTimer.stop()
    $MobTimer.stop()

func new_game():
    score = 0
    $Player.start($StartPosition.position)
    $StartTimer.start()
```

C#

```
public void GameOver()
{
    GetNode<Timer>("MobTimer").Stop();
    GetNode<Timer>("ScoreTimer").Stop();
}

public void NewGame()
{
    _score = 0;

    var player = GetNode<Player>("Player");
    var startPosition = GetNode<Marker2D>("StartPosition");
    player.Start(startPosition.Position);

    GetNode<Timer>("StartTimer").Start();
}
```

Now connect the timeout() signal of each of the Timer nodes (StartTimer, ScoreTimer, and MobTimer) to the main script. StartTimer will start the other two timers. ScoreTimer will increment the score by 1.

GDScript


```
func _on_score_timer_timeout():
    score += 1

func _on_start_timer_timeout():
    $MobTimer.start()
    $ScoreTimer.start()
```

C#

```
private void OnScoreTimerTimeout()
{
    _score++;
}

private void OnStartTimerTimeout()
{
    GetNode<Timer>("MobTimer").Start();
    GetNode<Timer>("ScoreTimer").Start();
}
```

In `_on_mob_timer_timeout()`, we will create a mob instance, pick a random starting location along the `Path2D`, and set the mob in motion. The `PathFollow2D` node will automatically rotate as it follows the path, so we will use that to select the mob's direction as well as its position. When we spawn a mob, we'll pick a random value between 150.0 and 250.0 for how fast each mob will move (it would be boring if they were all moving at the same speed).

Note that a new instance must be added to the scene using `add_child()`.

GScript

```
func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on Path2D.
    var mob_spawn_location = get_node("MobPath/MobSpawnLocation")
    mob_spawn_location.progress_ratio = randf()

    # Set the mob's direction perpendicular to the path direction.
    var direction = mob_spawn_location.rotation + PI / 2

    # Set the mob's position to a random location.
    mob.position = mob_spawn_location.position

    # Add some randomness to the direction.
    direction += randf_range(-PI / 4, PI / 4)
    mob.rotation = direction

    # Choose the velocity for the mob.
    var velocity = Vector2(randf_range(150.0, 250.0), 0.0)
    mob.linear_velocity = velocity.rotated(direction)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)
```

C#

```
private void OnMobTimerTimeout()
{
    // Note: Normally it is best to use explicit types rather than the `var`
    // keyword. However, var is acceptable to use here because the types are
    // obviously Mob and PathFollow2D, since they appear later on the line.

    // Create a new instance of the Mob scene.
    Mob mob = MobScene.Instantiate<Mob>();

    // Choose a random location on Path2D.
    var mobSpawnLocation = GetNode<PathFollow2D>("MobPath/MobSpawnLocation");
    mobSpawnLocation.ProgressRatio = GD.Randf();

    // Set the mob's direction perpendicular to the path direction.
    float direction = mobSpawnLocation.Rotation + Mathf.Pi / 2;

    // Set the mob's position to a random location.
    mob.Position = mobSpawnLocation.Position;

    // Add some randomness to the direction.
    direction += (float)GD.RandRange(-Mathf.Pi / 4, Mathf.Pi / 4);
    mob.Rotation = direction;

    // Choose the velocity.
    var velocity = new Vector2((float)GD.RandRange(150.0, 250.0), 0);
    mob.LinearVelocity = velocity.Rotated(direction);

    // Spawn the mob by adding it to the Main scene.
    AddChild(mob);
}
```

Important: Why PI? In functions requiring angles, Godot uses radians, not degrees. Pi represents a half turn in radians, about 3.1415 (there is also TAU which is equal to 2 * PI). If you're more comfortable working with degrees, you'll need to use the `deg_to_rad()` and `rad_to_deg()` functions to convert between the two.

Testing the scene

Let's test the scene to make sure everything is working. Add this `new_game` call to `_ready()`:

GDScript

```
func _ready():
    new_game()
```

C#

```
public override void _Ready()
{
    NewGame();
}
```

Let's also assign Main as our "Main Scene" - the one that runs automatically when the game launches. Press the "Play" button and select main.tscn when prompted.

Tip: If you had already set another scene as the "Main Scene", you can right click main.tscn in the FileSystem dock and select "Set As Main Scene".

You should be able to move the player around, see mobs spawning, and see the player disappear when hit by a mob.

When you're sure everything is working, remove the call to `new_game()` from `_ready()`.

What's our game lacking? Some user interface. In the next lesson, we'll add a title screen and display the player's score.

Heads up display

The final piece our game needs is a User Interface (UI) to display things like score, a "game over" message, and a restart button.

Create a new scene, click the "Other Node" button and add a CanvasLayer node named HUD. "HUD" stands for "heads-up display", an informational display that appears as an overlay on top of the game view.

The CanvasLayer node lets us draw our UI elements on a layer above the rest of the game, so that the information it displays isn't covered up by any game elements like the player or mobs.

The HUD needs to display the following information:

- Score, changed by ScoreTimer.
- A message, such as "Game Over" or "Get Ready!"
- A "Start" button to begin the game.

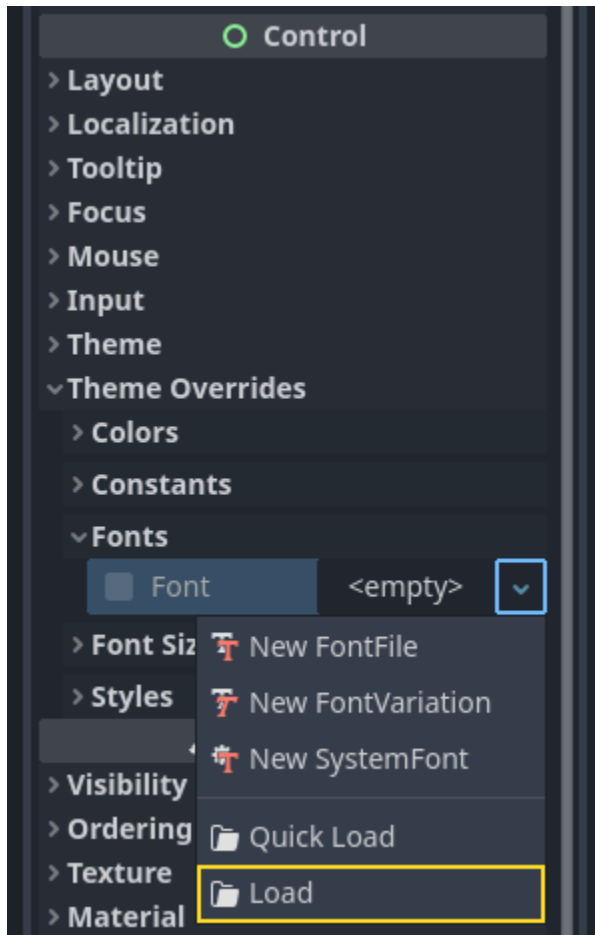
The basic node for UI elements is Control. To create our UI, we'll use two types of Control nodes: Label and Button.

Create the following as children of the HUD node:

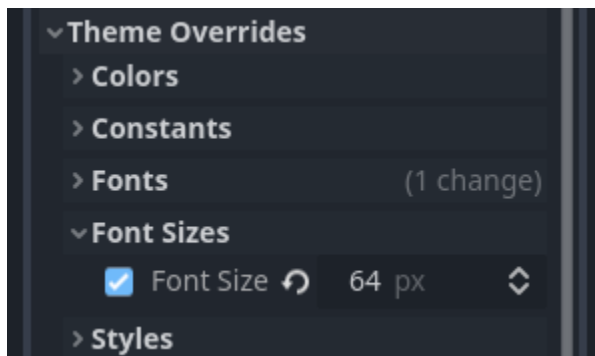
- Label named ScoreLabel.
- Label named Message.
- Button named StartButton.
- Timer named MessageTimer.

Click on the ScoreLabel and type a number into the Text field in the Inspector. The default font for Control nodes is small and doesn't scale well. There is a font file included in the game assets called "Xolonium-Regular.ttf". To use this font, do the following:

Under "Theme Overrides > Fonts", choose "Load" and select the "Xolonium-Regular.ttf" file.

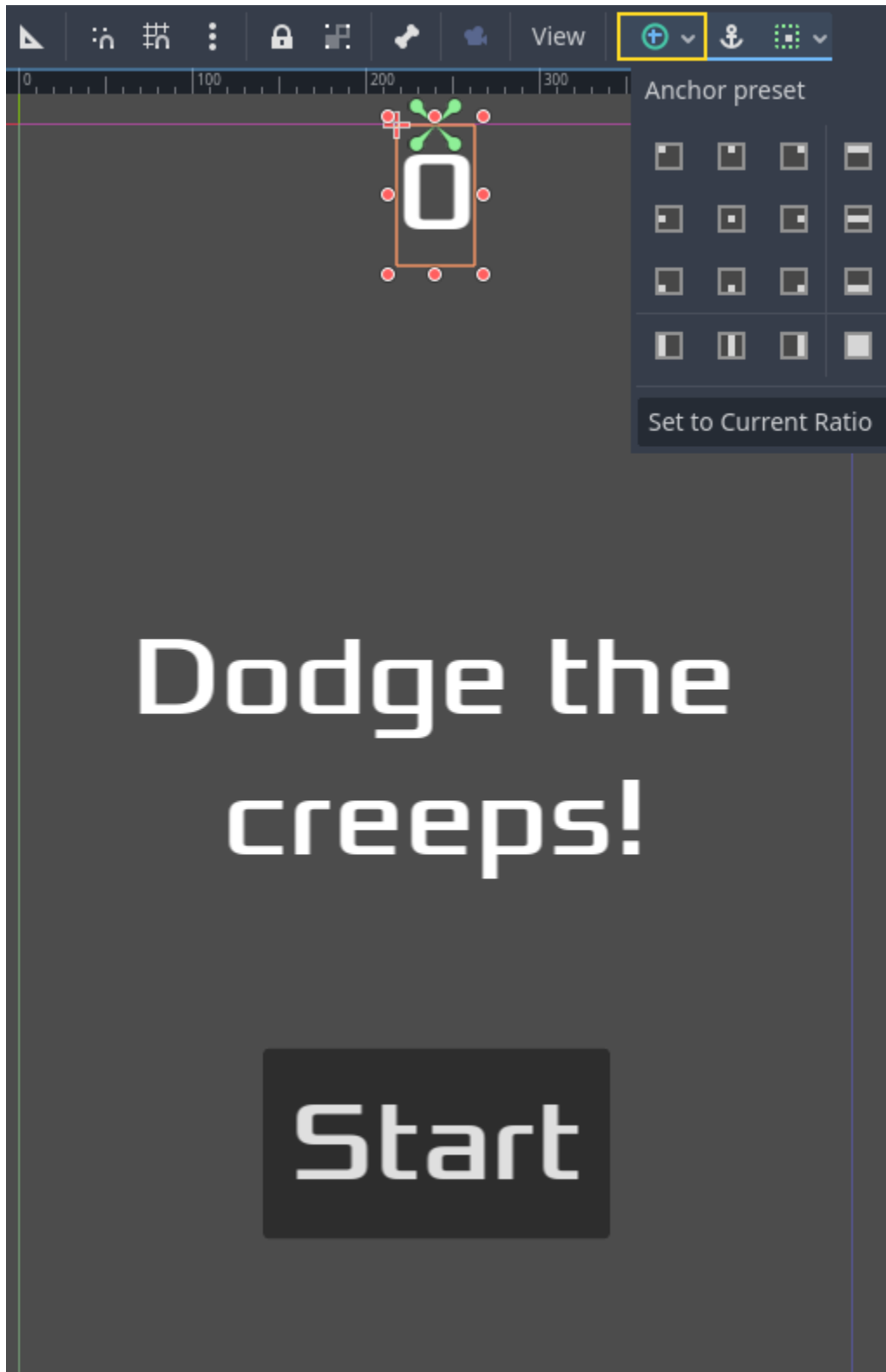


The font size is still too small, increase it to 64 under "Theme Overrides > Font Sizes". Once you've done this with the ScoreLabel, repeat the changes for the Message and StartButton nodes.



Note: Anchors: Control nodes have a position and size, but they also have anchors. Anchors define the origin - the reference point for the edges of the node.

Arrange the nodes as shown below. You can drag the nodes to place them manually, or for more precise placement, use "Anchor Presets".



ScoreLabel

1. Add the text 0.
2. Set the "Horizontal Alignment" and "Vertical Alignment" to Center.
3. Choose the "Anchor Preset" Center Top.

Message

1. Add the text Dodge the Creeps!.
2. Set the "Horizontal Alignment" and "Vertical Alignment" to Center.
3. Set the "Autowrap Mode" to Word, otherwise the label will stay on one line.
4. Under "Control - Layout/Transform" set "Size X" to 480 to use the entire width of the screen.
5. Choose the "Anchor Preset" Center.

StartButton

1. Add the text Start.
2. Under "Control - Layout/Transform", set "Size X" to 200 and "Size Y" to 100 to add a little bit more padding between the border and text.
3. Choose the "Anchor Preset" Center Bottom.
4. Under "Control - Layout/Transform", set "Position Y" to 580.

On the MessageTimer, set the Wait Time to 2 and set the One Shot property to "On".

Now add this script to HUD:

GDScript

```
extends CanvasLayer

# Notifies `Main` node that the button has been pressed
signal start_game
```

C#

```
using Godot;

public partial class HUD : CanvasLayer
{
    // Don't forget to rebuild the project so the editor knows about the new signal.

    [Signal]
    public delegate void StartGameEventHandler();
}
```

We now want to display a message temporarily, such as "Get Ready", so we add the following code

GDScript

```
func show_message(text):
    $Message.text = text
```

(continues on next page)

(continued from previous page)

```
$Message.show()
$MessageTimer.start()
```

C#

```
public void ShowMessage(string text)
{
    var message = GetNode<Label>("Message");
    message.Text = text;
    message.Show();

    GetNode<Timer>("MessageTimer").Start();
}
```

We also need to process what happens when the player loses. The code below will show "Game Over" for 2 seconds, then return to the title screen and, after a brief pause, show the "Start" button.

GDScript

```
func show_game_over():
    show_message("Game Over")
    # Wait until the MessageTimer has counted down.
    await $MessageTimer.timeout

    $Message.text = "Dodge the Creeps!"
    $Message.show()
    # Make a one-shot timer and wait for it to finish.
    await get_tree().create_timer(1.0).timeout
    $StartButton.show()
```

C#

```
async public void ShowGameOver()
{
    ShowMessage("Game Over");

    var messageTimer = GetNode<Timer>("MessageTimer");
    await ToSignal(messageTimer, Timer.SignalName.Timeout);

    var message = GetNode<Label>("Message");
    message.Text = "Dodge the Creeps!";
    message.Show();

    await ToSignal(GetTree().CreateTimer(1.0), SceneTreeTimer.SignalName.Timeout);
    GetNode<Button>("StartButton").Show();
}
```

This function is called when the player loses. It will show "Game Over" for 2 seconds, then return to the title screen and, after a brief pause, show the "Start" button.

Note: When you need to pause for a brief time, an alternative to using a Timer node is to use the SceneTree's `create_timer()` function. This can be very useful to add delays such as in the above code, where we want to

wait some time before showing the "Start" button.

Add the code below to HUD to update the score

GDScript

```
func update_score(score):
    $ScoreLabel.text = str(score)
```

C#

```
public void UpdateScore(int score)
{
    GetNode<Label>("ScoreLabel").Text = score.ToString();
}
```

Connect the `timeout()` signal of `MessageTimer` and the `pressed()` signal of `StartButton`, and add the following code to the new functions:

GDScript

```
func _on_start_button_pressed():
    $StartButton.hide()
    start_game.emit()

func _on_message_timer_timeout():
    $Message.hide()
```

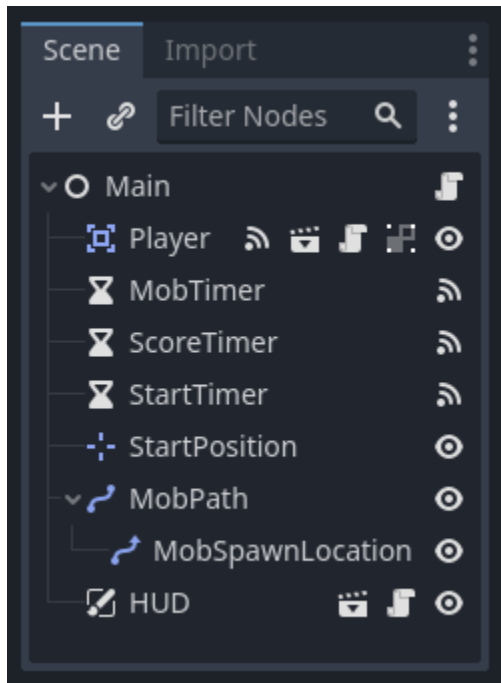
C#

```
private void OnStartButtonPressed()
{
    GetNode<Button>("StartButton").Hide();
    EmitSignal(SignalName.StartGame);
}

private void OnMessageTimerTimeout()
{
    GetNode<Label>("Message").Hide();
}
```


Connecting HUD to Main

Now that we're done creating the HUD scene, go back to Main. Instance the HUD scene in Main like you did the Player scene. The scene tree should look like this, so make sure you didn't miss anything:



Now we need to connect the HUD functionality to our Main script. This requires a few additions to the Main scene:

In the Node tab, connect the HUD's `start_game` signal to the `new_game()` function of the Main node by clicking the "Pick" button in the "Connect a Signal" window and selecting the `new_game()` method or type "new_game" below "Receiver Method" in the window. Verify that the green connection icon now appears next to `func new_game()` in the script.

In `new_game()`, update the score display and show the "Get Ready" message:

GDScript

```
$HUD.update_score(score)
$HUD.show_message("Get Ready")
```

C#

```
var hud = GetNode<HUD>("HUD");
hud.UpdateScore(_score);
hud.ShowMessage("Get Ready!");
```

In `game_over()` we need to call the corresponding HUD function:

GDScript

```
$HUD.show_game_over()
```

C#

```
GetNode<HUD>("HUD").ShowGameOver();
```

Finally, add this to `_on_score_timer_timeout()` to keep the display in sync with the changing score:

GDScript

```
$HUD.update_score(score)
```

C#

```
GetNode<HUD>("HUD").UpdateScore(_score);
```

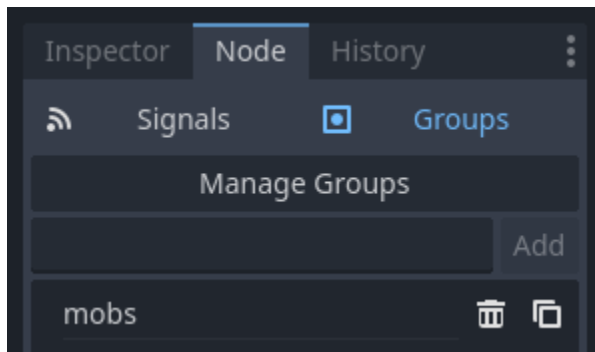
Warning: Remember to remove the call to `new_game()` from `_ready()` if you haven't already, otherwise your game will start automatically.

Now you're ready to play! Click the "Play the Project" button. You will be asked to select a main scene, so choose `main.tscn`.

Removing old creeps

If you play until "Game Over" and then start a new game right away, the creeps from the previous game may still be on the screen. It would be better if they all disappeared at the start of a new game. We just need a way to tell all the mobs to remove themselves. We can do this with the "group" feature.

In the Mob scene, select the root node and click the "Node" tab next to the Inspector (the same place where you find the node's signals). Next to "Signals", click "Groups" and you can type a new group name and click "Add".



Now all mobs will be in the "mobs" group. We can then add the following line to the `new_game()` function in Main:

GDScript

```
get_tree().call_group("mobs", "queue_free")
```

C#

```
// Note that for calling Godot-provided methods with strings,  
// we have to use the original Godot snake_case name.  
GetTree().CallGroup("mobs", Node.MethodName.QueueFree);
```

The `call_group()` function calls the named function on every node in a group - in this case we are telling every mob to delete itself.

The game's mostly done at this point. In the next and last part, we'll polish it a bit by adding a background, looping music, and some keyboard shortcuts.

Finishing up

We have now completed all the functionality for our game. Below are some remaining steps to add a bit more "juice" to improve the game experience.

Feel free to expand the gameplay with your own ideas.

Background

The default gray background is not very appealing, so let's change its color. One way to do this is to use a `ColorRect` node. Make it the first node under Main so that it will be drawn behind the other nodes. `ColorRect` only has one property: `Color`. Choose a color you like and select "Layout" -> "Anchors Preset" -> "Full Rect" either in the toolbar at the top of the viewport or in the inspector so that it covers the screen.

You could also add a background image, if you have one, by using a `TextureRect` node instead.

Sound effects

Sound and music can be the single most effective way to add appeal to the game experience. In your game's art folder, you have two sound files: "House In a Forest Loop.ogg" for background music, and "gameover.wav" for when the player loses.

Add two `AudioStreamPlayer` nodes as children of Main. Name one of them Music and the other DeathSound. On each one, click on the Stream property, select "Load", and choose the corresponding audio file.

All audio is automatically imported with the Loop setting disabled. If you want the music to loop seamlessly, click on the Stream file arrow, select Make Unique, then click on the Stream file and check the Loop box.

To play the music, add `$Music.play()` in the `new_game()` function and `$Music.stop()` in the `game_over()` function.

Finally, add `$DeathSound.play()` in the `game_over()` function.

GDScript

```
func game_over():
    ...
    $Music.stop()
    $DeathSound.play()

func new_game():
    ...
    $Music.play()
```

C#

```
public void GameOver()
{
    ...
    GetNode<AudioStreamPlayer>("Music").Stop();
    GetNode<AudioStreamPlayer>("DeathSound").Play();
}

public void NewGame()
```

(continues on next page)

(continued from previous page)

```
{  
  ...  
  GetNode<AudioStreamPlayer>("Music").Play();  
}
```

Keyboard shortcut

Since the game is played with keyboard controls, it would be convenient if we could also start the game by pressing a key on the keyboard. We can do this with the "Shortcut" property of the Button node.

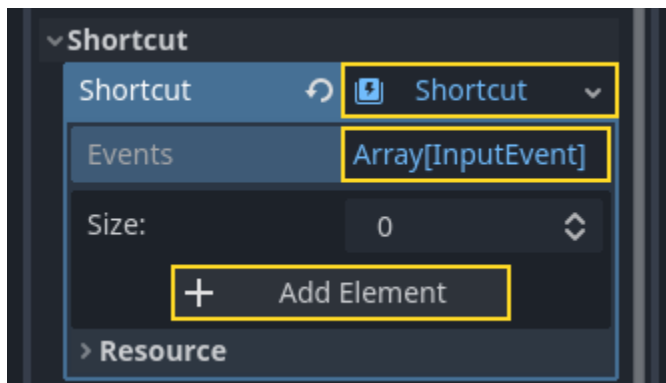
In a previous lesson, we created four input actions to move the character. We will create a similar input action to map to the start button.

Select "Project" -> "Project Settings" and then click on the "Input Map" tab. In the same way you created the movement input actions, create a new input action called `start_game` and add a key mapping for the Enter key.

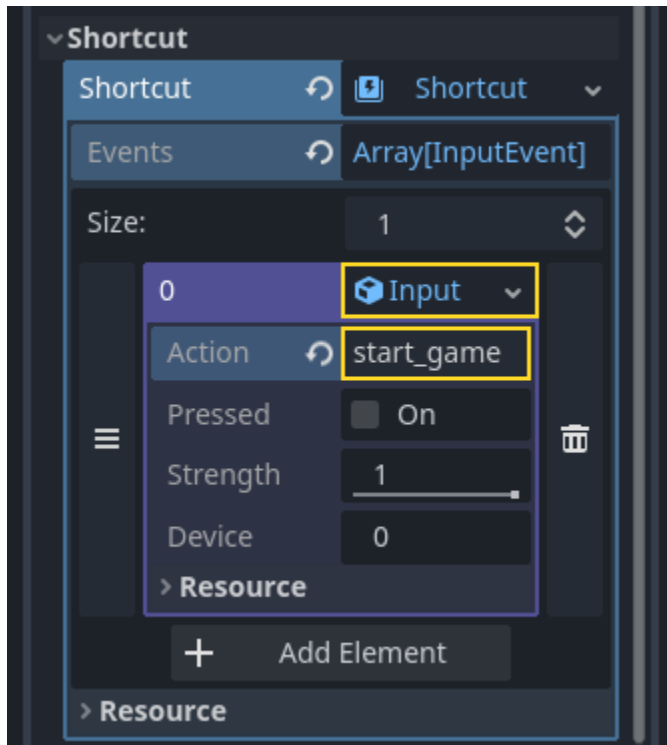


Now would be a good time to add controller support if you have one available. Attach or pair your controller and then under each input action that you wish to add controller support for, click on the "+" button and press the corresponding button, d-pad, or stick direction that you want to map to the respective input action.

In the HUD scene, select the StartButton and find its Shortcut property in the Inspector. Create a new Shortcut resource by clicking within the box, open the Events array and add a new array element to it by clicking on `Array[InputEvent]` (size 0).



Create a new `InputEventAction` and name it `start_game`.



Now when the start button appears, you can either click it or press Enter to start the game.

And with that, you completed your first 2D game in Godot.

You got to make a player-controlled character, enemies that spawn randomly around the game board, count the score, implement a game over and replay, user interface, sounds, and more. Congratulations!

There's still much to learn, but you can take a moment to appreciate what you achieved.

And when you're ready, you can move on to Your first 3D game to learn to create a complete 3D game from scratch, in Godot.

2.10 Your first 3D game

In this step-by-step tutorial series, you will create your first complete 3D game with Godot. By the end of the series, you will have a simple yet finished project of your own like the animated gif below.

The game we'll code here is similar to Your first 2D game, with a twist: you can now jump and your goal is to squash the creeps. This way, you will both recognize patterns you learned in the previous tutorial and build upon them with new code and features.

You will learn to:

- Work with 3D coordinates with a jumping mechanic.
- Use kinematic bodies to move 3D characters and detect when and how they collide.
- Use physics layers and a group to detect interactions with specific entities.
- Code basic procedural gameplay by instancing monsters at regular time intervals.
- Design a movement animation and change its speed at run-time.

- Draw a user interface on a 3D game.

And more.

This tutorial is for beginners who followed the complete getting started series. We'll start slow with detailed instructions and shorten them as we do similar steps. If you're an experienced programmer, you can browse the complete demo's source code here: [Squash the Creep source code](#).

Note: You can follow this series without having done the 2D one. However, if you're new to game development, we recommend you to start with 2D. 3D game code is always more complex and the 2D series will give you foundations to follow along more comfortably.

We prepared some game assets so we can jump straight to the code. You can download them here: [Squash the Creeps assets](#).

We will first work on a basic prototype for the player's movement. We will then add the monsters that we'll spawn randomly around the screen. After that, we'll implement the jump and squashing mechanic before refining the game with some nice animation. We'll wrap up with the score and the retry screen.

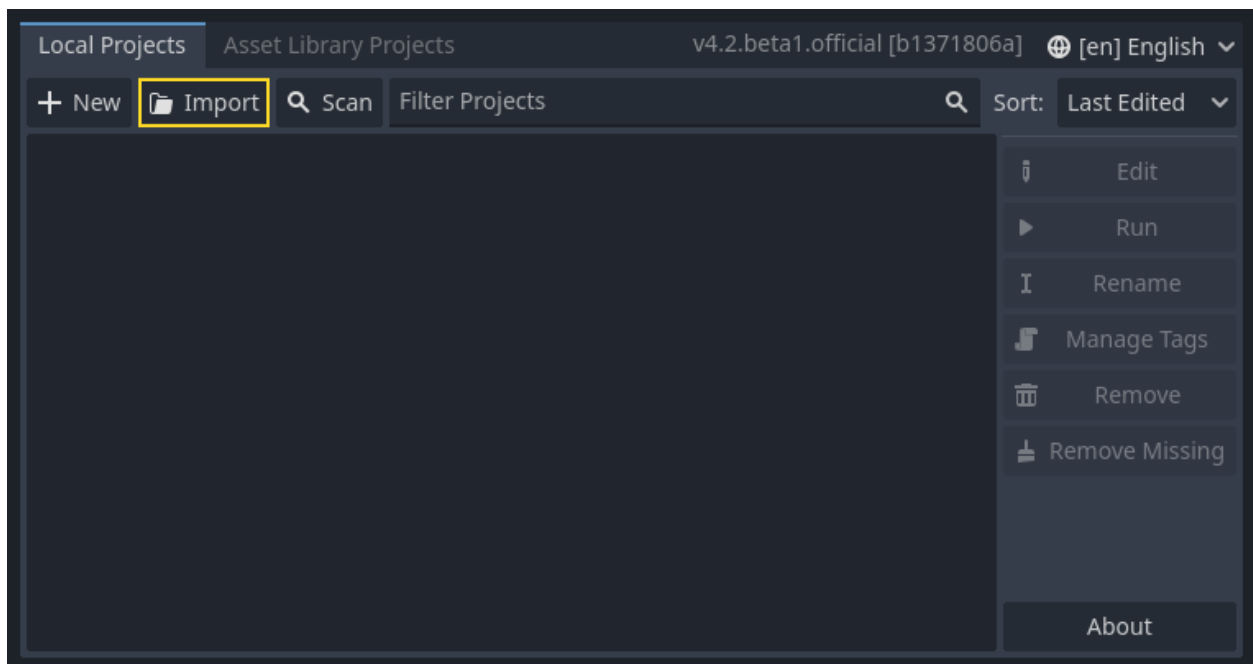
2.10.1 Contents

Setting up the game area

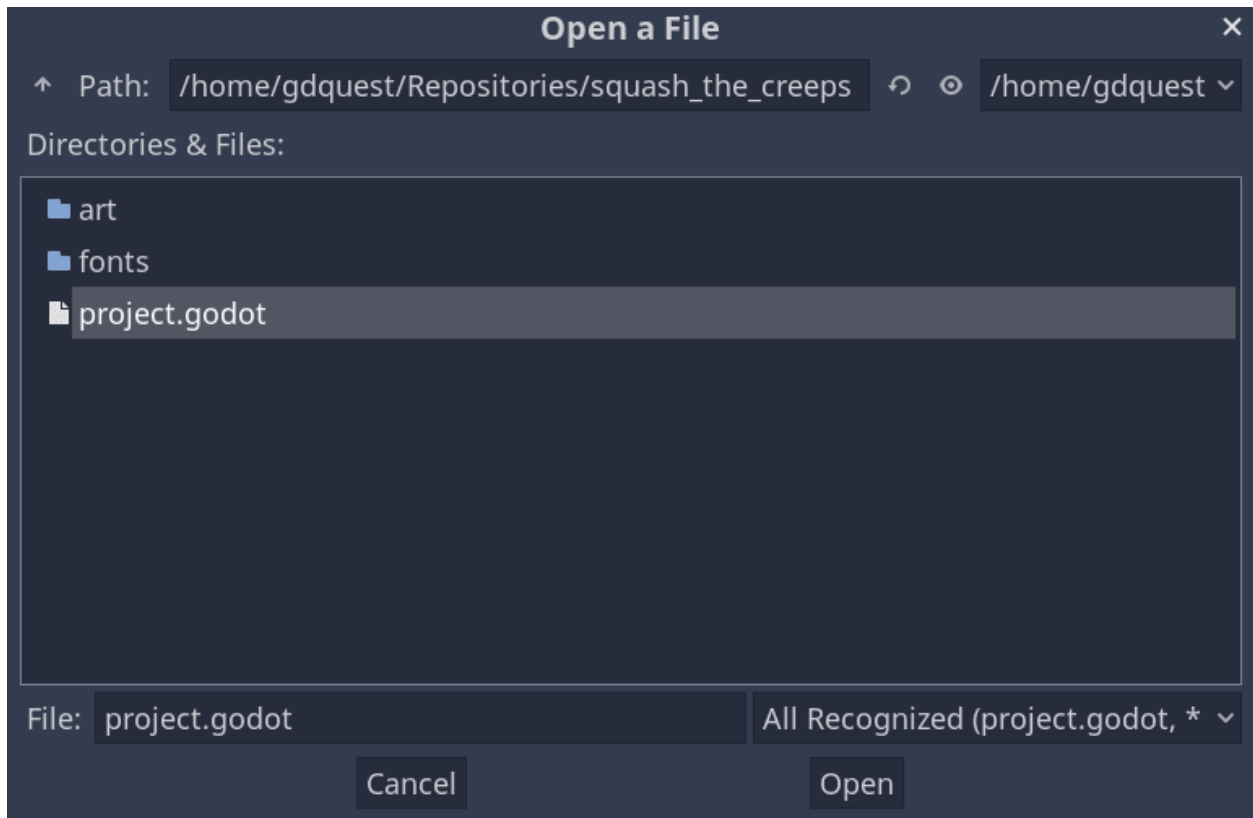
In this first part, we're going to set up the game area. Let's get started by importing the start assets and setting up the game scene.

We've prepared a Godot project with the 3D models and sounds we'll use for this tutorial, linked in the index page. If you haven't done so yet, you can download the archive here: [Squash the Creeps assets](#).

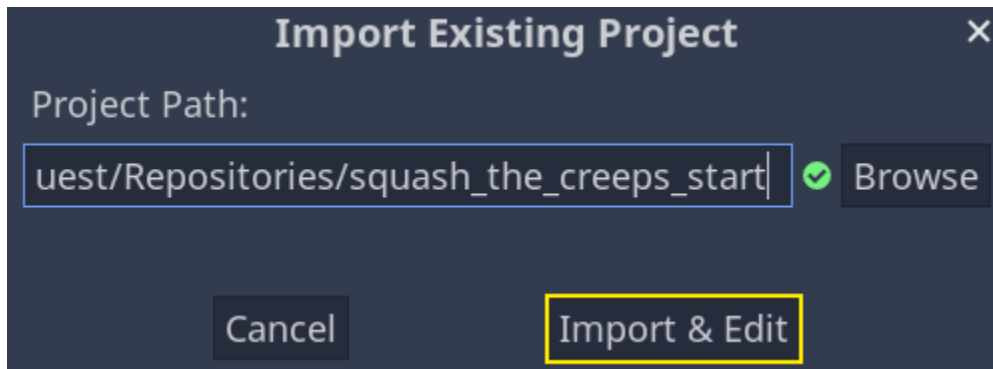
Once you downloaded it, extract the .zip archive on your computer. Open the Godot Project Manager and click the Import button.



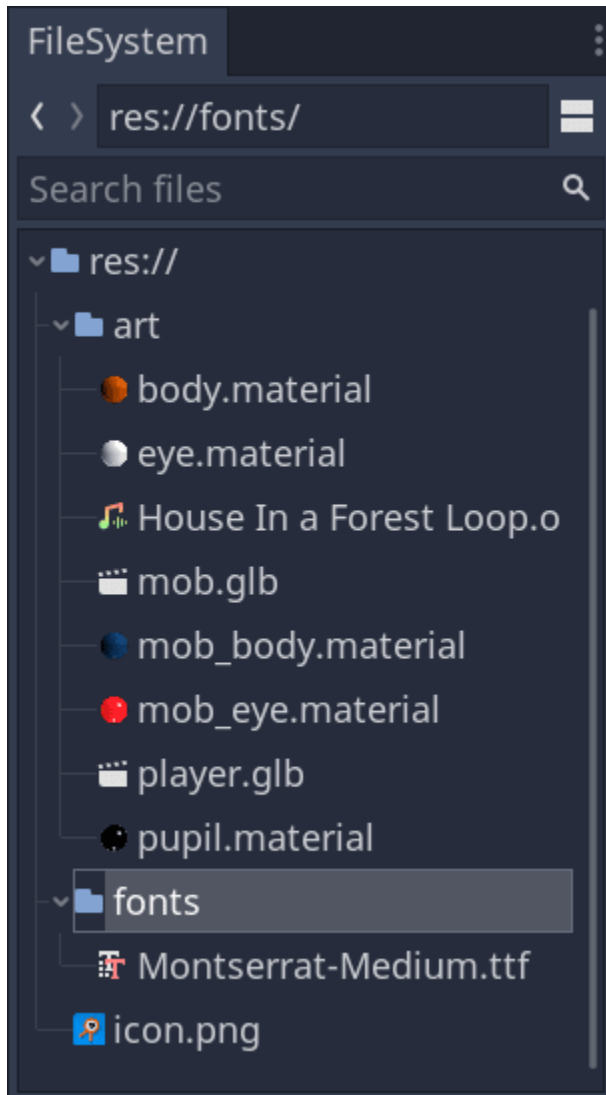
In the import popup, enter the full path to the freshly created directory `squash_the_creeps_start/`. You can click the Browse button on the right to open a file browser and navigate to the `project.godot` file the folder contains.



Click Import & Edit to open the project in the editor.



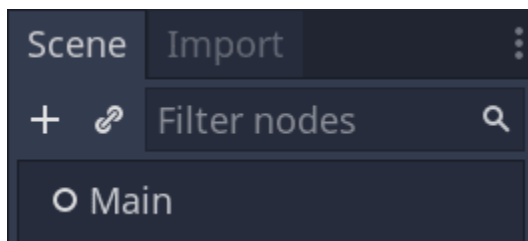
The start project contains an icon and two folders: art/ and fonts/. There, you will find the art assets and music we'll use in the game.



There are two 3D models, `player.glb` and `mob.glb`, some materials that belong to these models, and a music track.

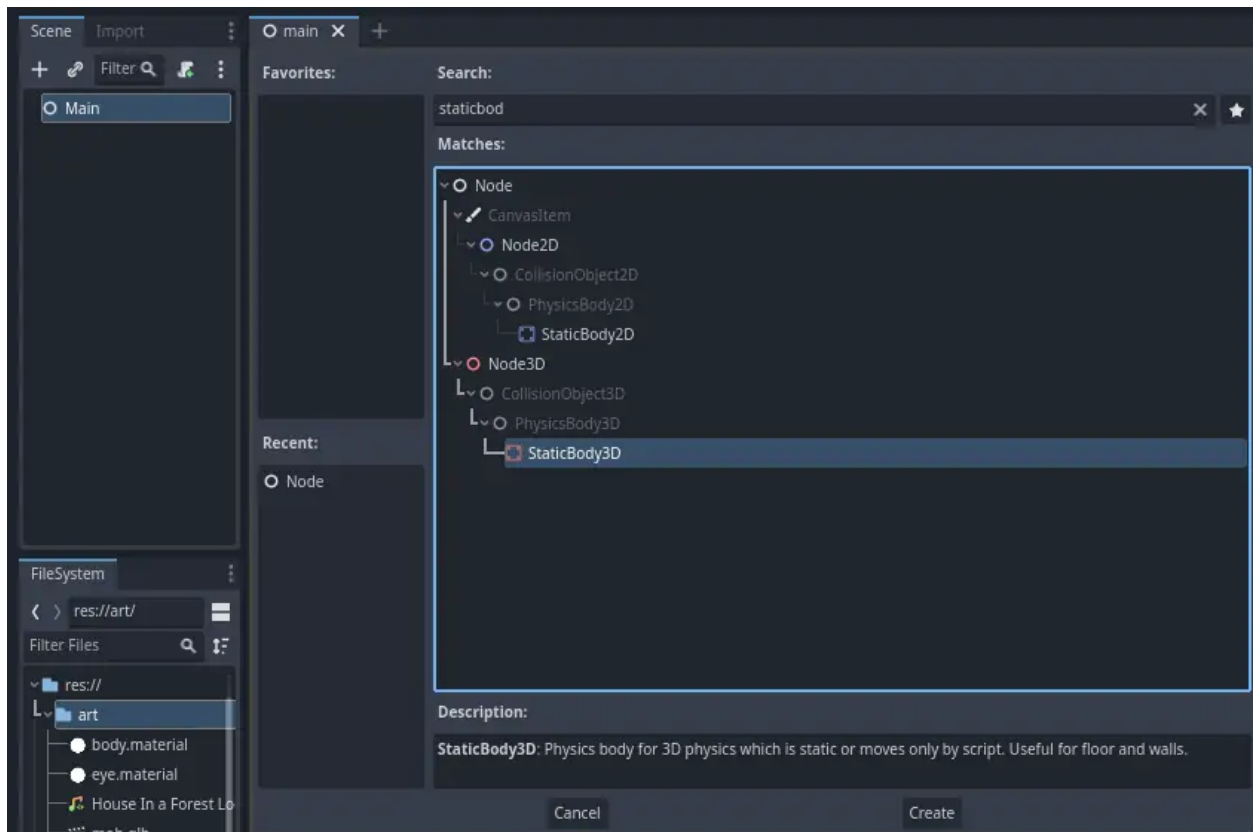
Setting up the playable area

We're going to create our main scene with a plain Node as its root. In the Scene dock, click the Add Child Node button represented by a "+" icon in the top-left and double-click on Node. Name the node Main. An alternate method to rename the node is to right-click on Node and choose Rename (or F2). Alternatively, to add a node to the scene, you can press Ctrl + a (or Cmd + a on macOS).

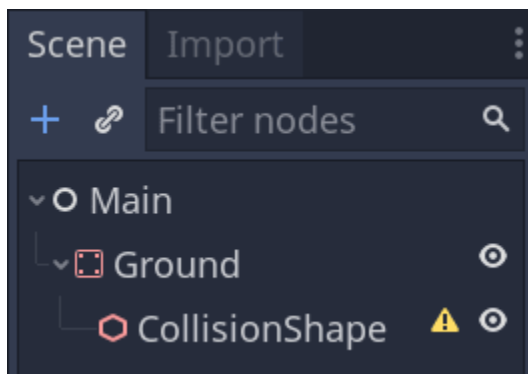


Save the scene as `main.tscn` by pressing Ctrl + s (Cmd + s on macOS).

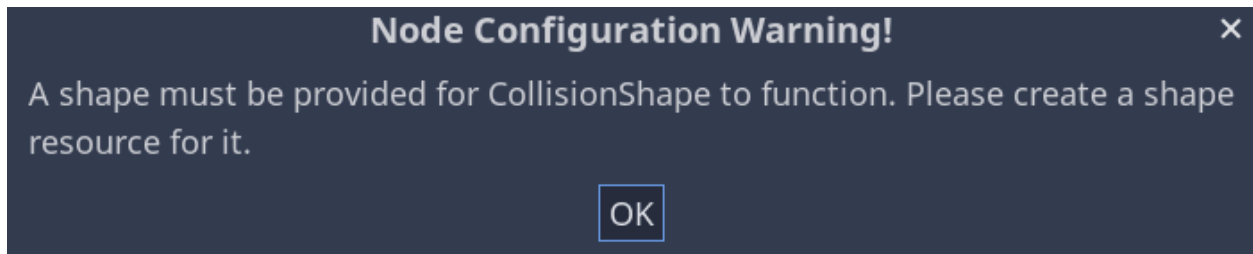
We'll start by adding a floor that'll prevent the characters from falling. To create static colliders like the floor, walls, or ceilings, you can use `StaticBody3D` nodes. They require `CollisionShape3D` child nodes to define the collision area. With the Main node selected, add a `StaticBody3D` node, then a `CollisionShape3D`. Rename the `StaticBody3D` to `Ground`.



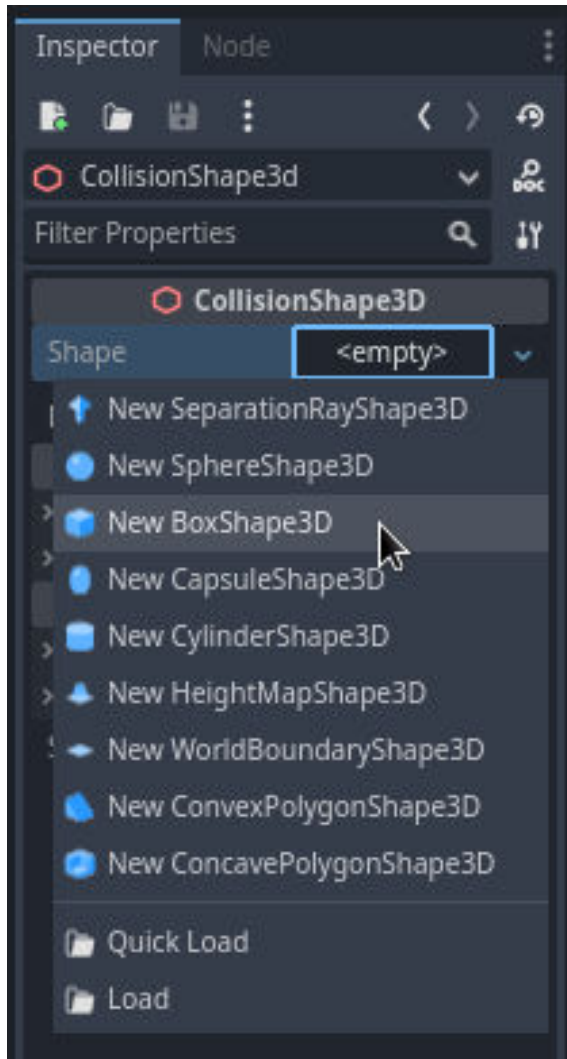
Your scene tree should look like this



A warning sign next to the `CollisionShape3D` appears because we haven't defined its shape. If you click the icon, a popup appears to give you more information.

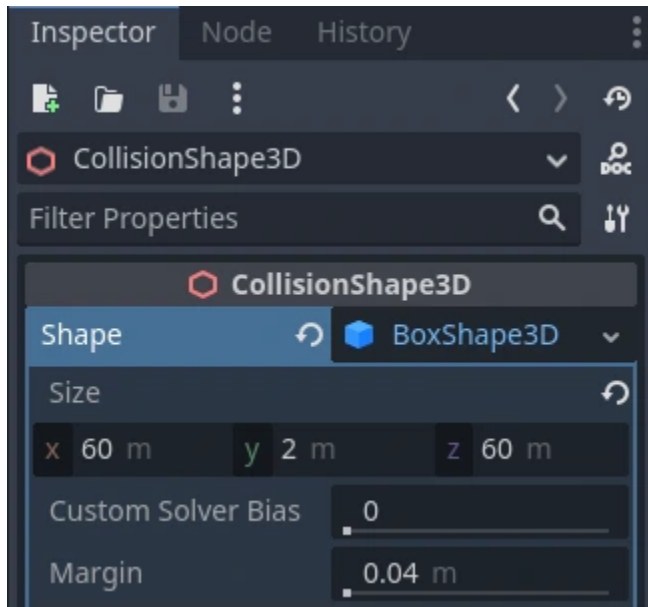


To create a shape, select the CollisionShape3D node, head to the Inspector and click the <empty> field next to the Shape property. Create a new BoxShape3D.

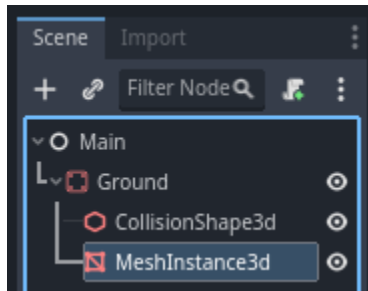


The box shape is perfect for flat ground and walls. Its thickness makes it reliable to block even fast-moving objects.

A box's wireframe appears in the viewport with three orange dots. You can click and drag these to edit the shape's extents interactively. We can also precisely set the size in the inspector. Click on the BoxShape3D to expand the resource. Set its Size to 60 on the X axis, 2 for the Y axis, and 60 for the Z axis.



Collision shapes are invisible. We need to add a visual floor that goes along with it. Select the Ground node and add a MeshInstance3D as its child.



In the Inspector, click on the field next to Mesh and create a BoxMesh resource to create a visible box.

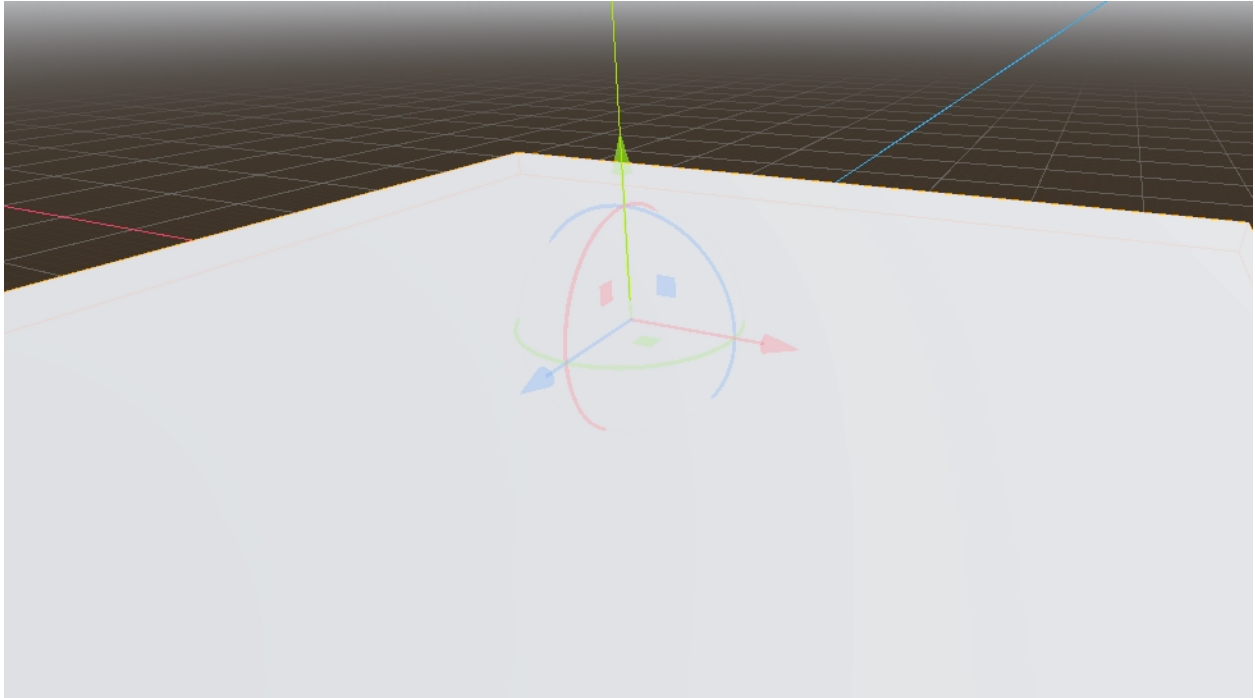


Once again, it's too small by default. Click the box icon to expand the resource and set its Size to 60, 2, and 60.

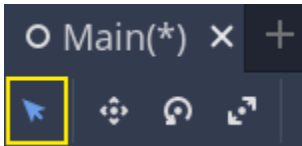


You should see a wide grey slab that covers the grid and blue and red axes in the viewport.

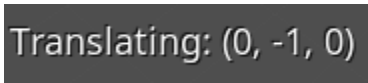
We're going to move the ground down so we can see the floor grid. Select the Ground node, hold the Ctrl key down to turn on grid snapping, and click and drag down on the Y axis. It's the green arrow in the move gizmo.



Note: If you can't see the 3D object manipulator like on the image above, ensure the Select Mode is active in the toolbar above the view.



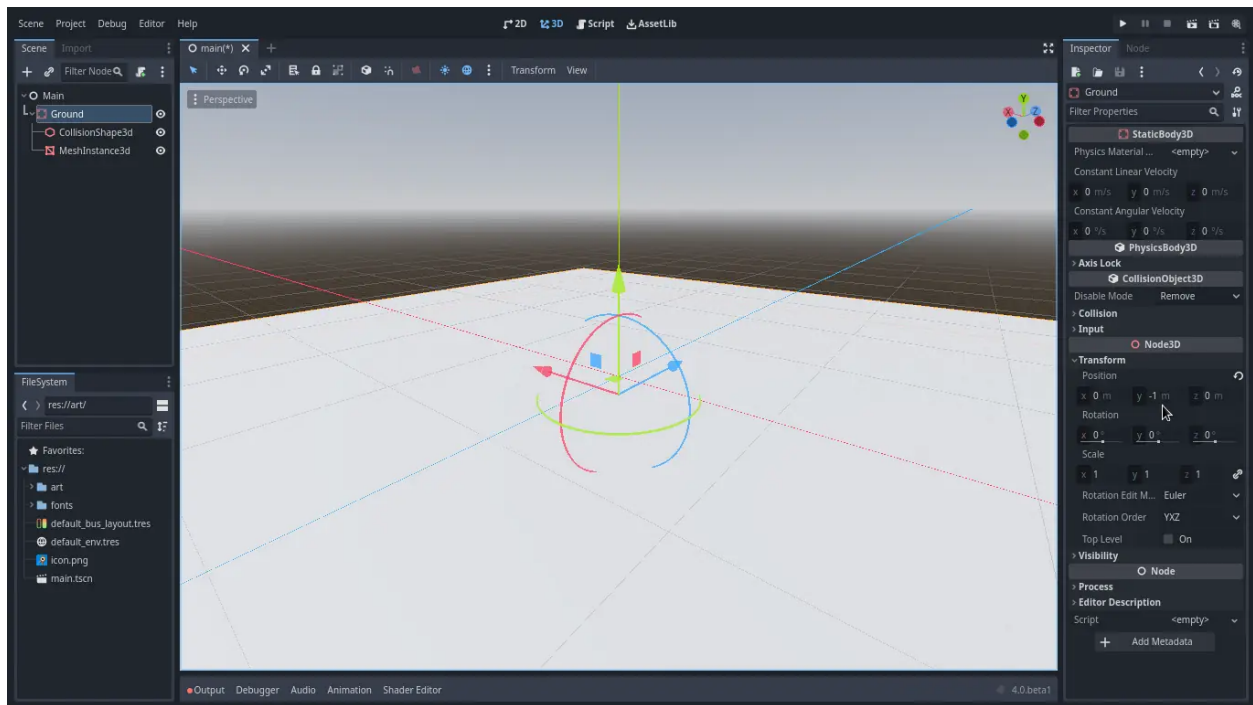
Move the ground down 1 meter, in order to have a visible editor grid. A label in the bottom-left corner of the viewport tells you how much you're translating the node.



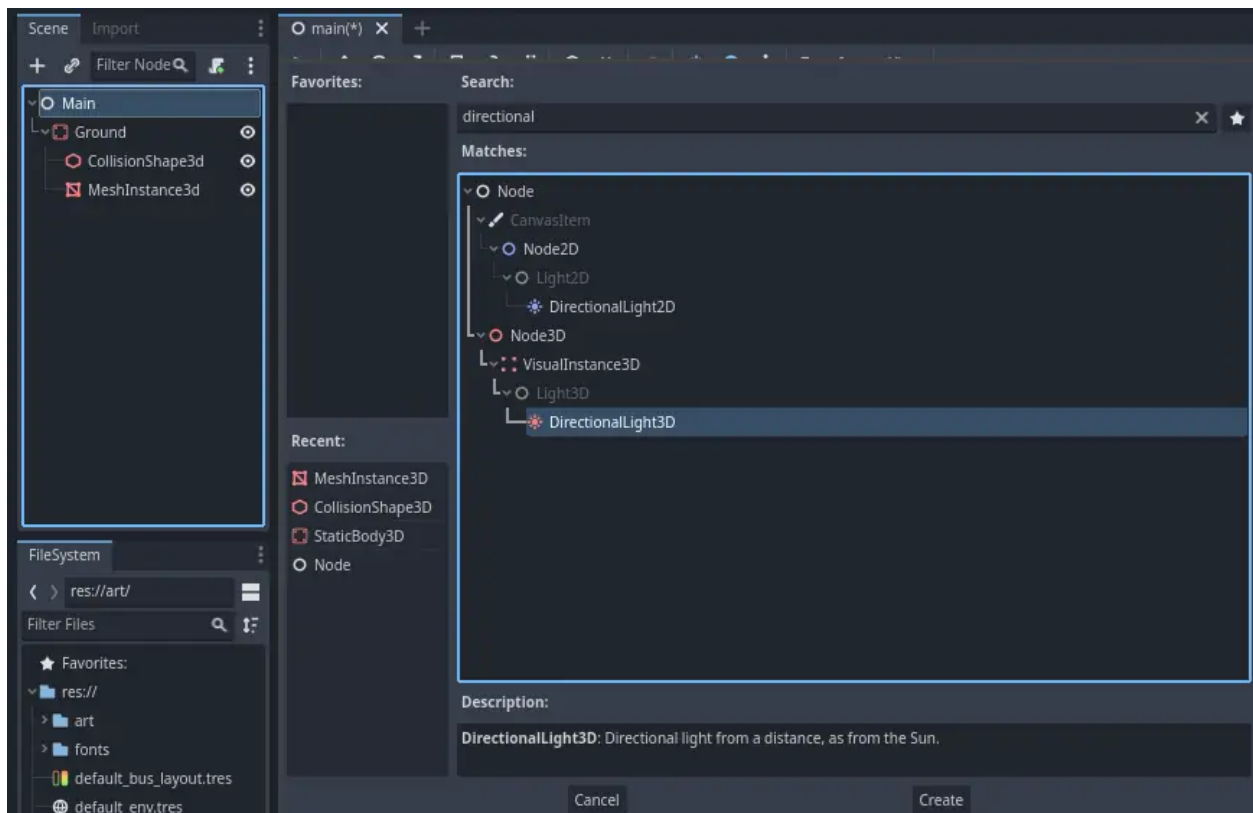
Translating: (0, -1, 0)

Note: Moving the Ground node down moves both children along with it. Ensure you move the Ground node, not the MeshInstance3D or the CollisionShape3D.

Ultimately, Ground's `transform.position.y` should be -1



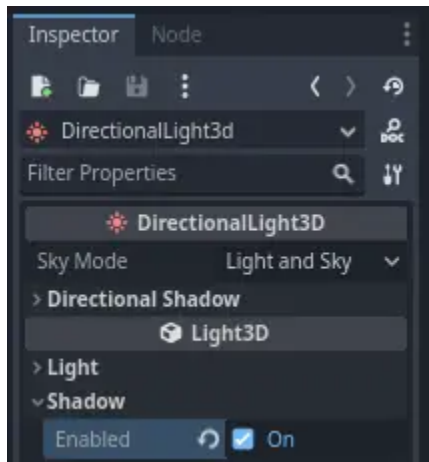
Let's add a directional light so our scene isn't all grey. Select the Main node and add a child node `DirectionalLight3D`.



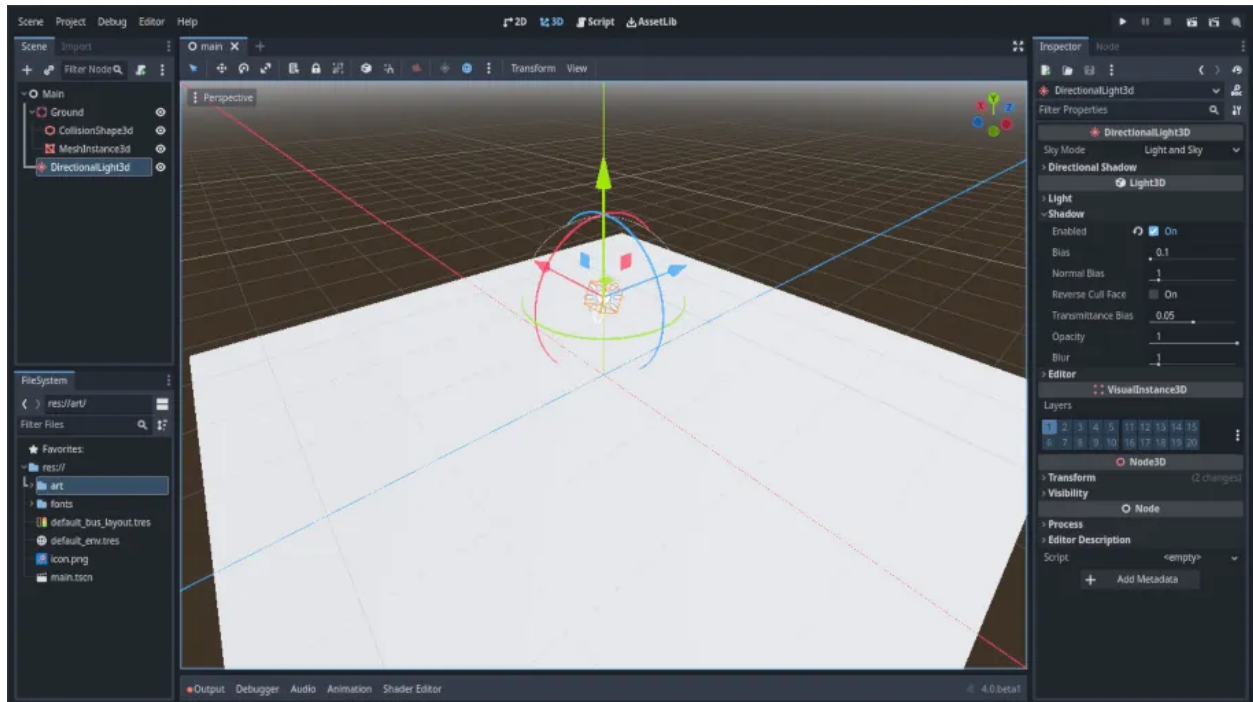
We need to move and rotate the `DirectionalLight3D` node. Move it up by clicking and dragging on the manipulator's green arrow and click and drag on the red arc to rotate it around the X axis, until the ground

is lit.

In the Inspector, turn on Shadow -> Enabled by clicking the checkbox.



At this point, your project should look like this.

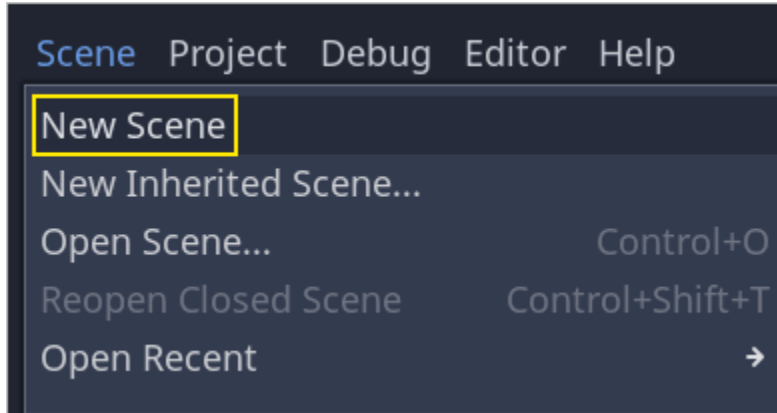


That's our starting point. In the next part, we will work on the player scene and base movement.

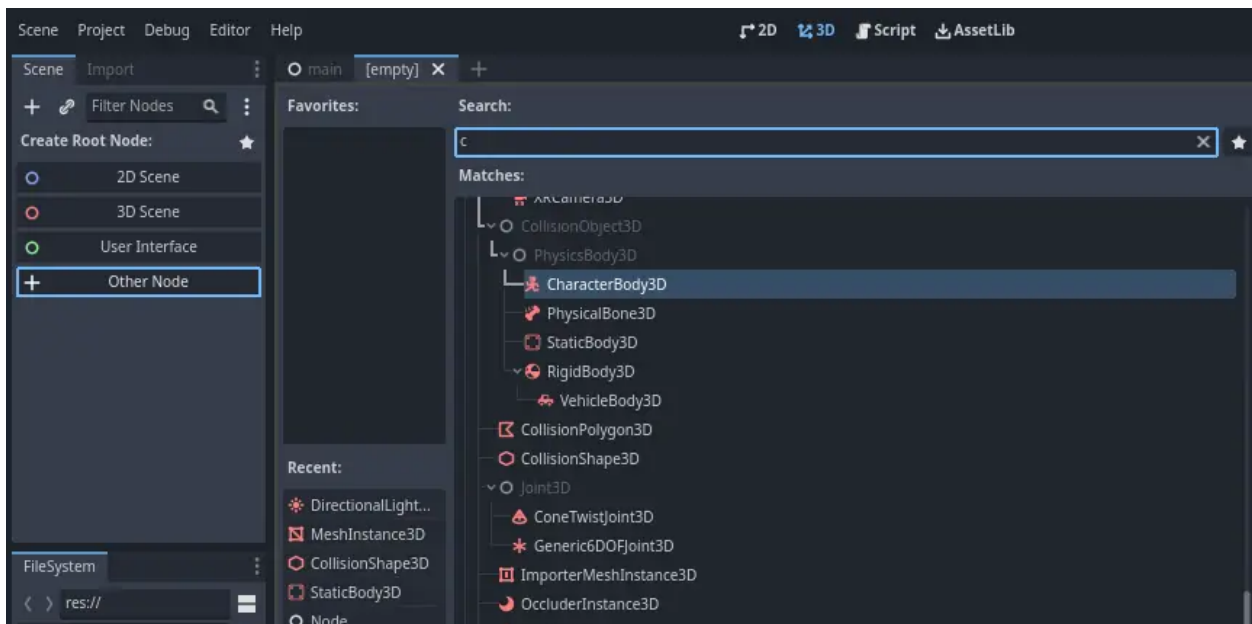
Player scene and input actions

In the next two lessons, we will design the player scene, register custom input actions, and code player movement. By the end, you'll have a playable character that moves in eight directions.

Create a new scene by going to the Scene menu in the top-left and clicking New Scene.



Create a `CharacterBody3D` node as the root



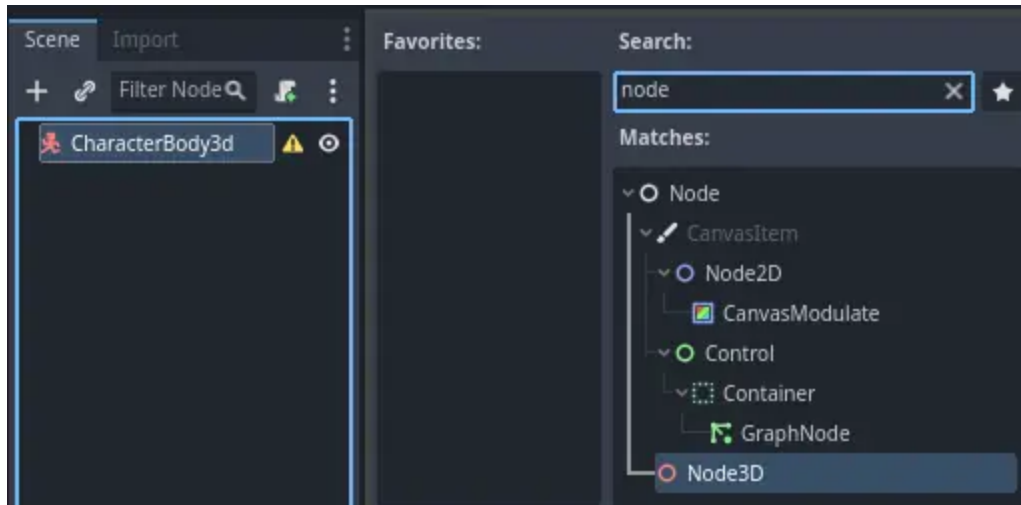
Name the `CharacterBody3D` to `Player`. Character bodies are complementary to the area and rigid bodies used in the 2D game tutorial. Like rigid bodies, they can move and collide with the environment, but instead of being controlled by the physics engine, you dictate their movement. You will see how we use the node's unique features when we code the jump and squash mechanics.

See also:

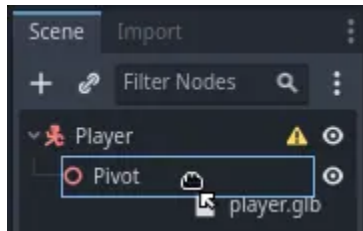
To learn more about the different physics node types, see the [Physics introduction](#).

For now, we're going to create a basic rig for our character's 3D model. This will allow us to rotate the model later via code while it plays an animation.

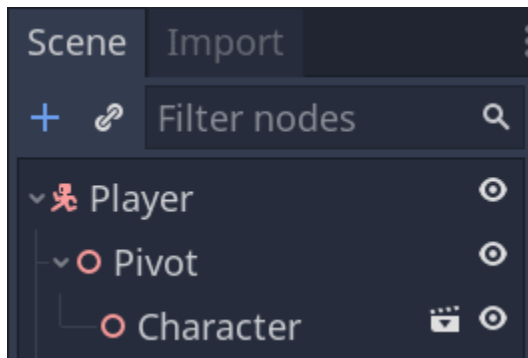
Add a `Node3D` node as a child of `Player` and name it `Pivot`



Then, in the FileSystem dock, expand the art/ folder by double-clicking it and drag and drop player.glb onto Pivot.

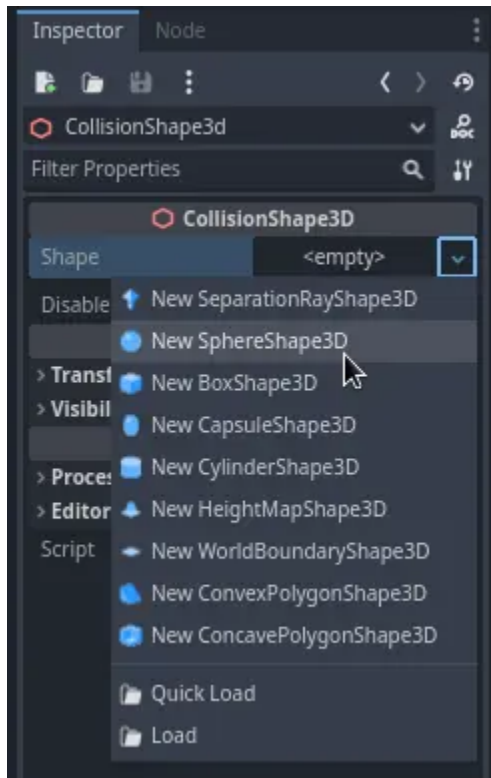


This should instantiate the model as a child of Pivot. You can rename it to Character.

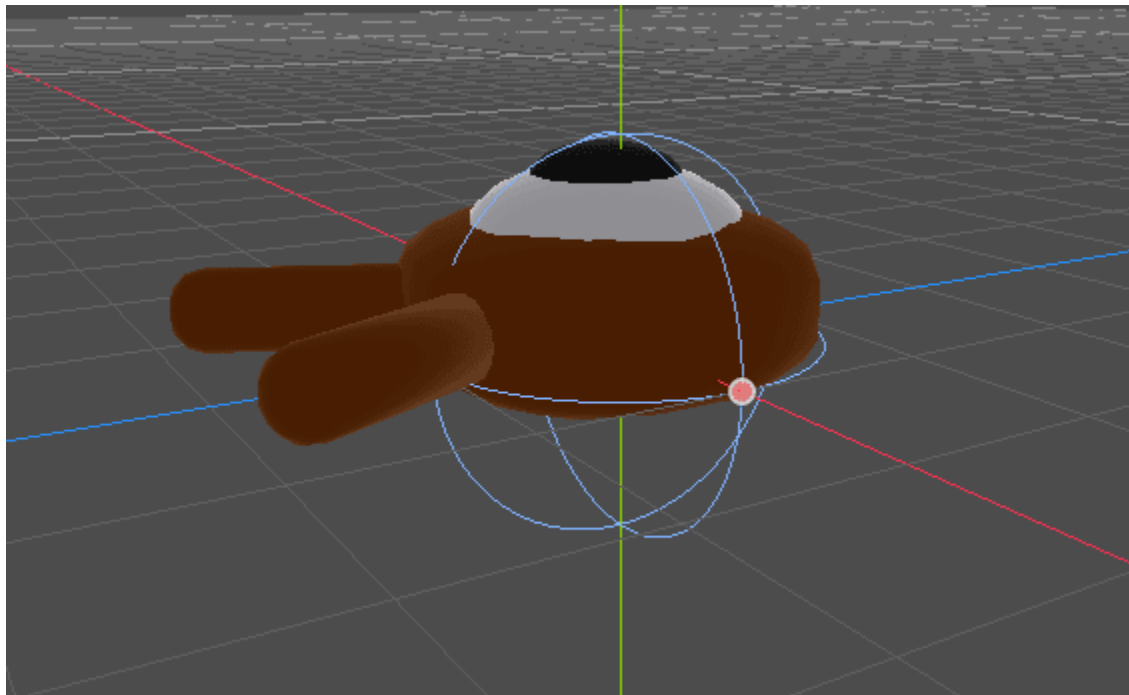


Note: The .glb files contain 3D scene data based on the open source GLTF 2.0 specification. They're a modern and powerful alternative to a proprietary format like FBX, which Godot also supports. To produce these files, we designed the model in [Blender 3D](#) and exported it to GLTF.

As with all kinds of physics nodes, we need a collision shape for our character to collide with the environment. Select the Player node again and add a child node CollisionShape3D. In the Inspector, on the Shape property, add a new SphereShape3D.

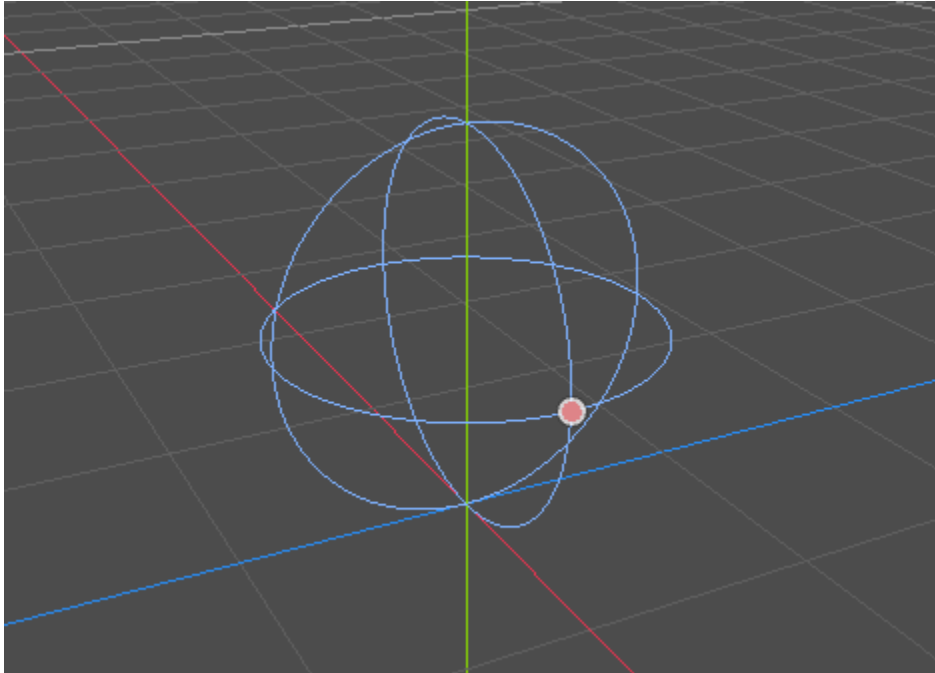


The sphere's wireframe appears below the character.

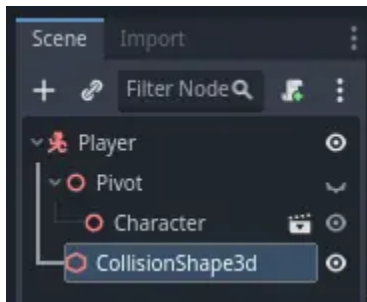


It will be the shape the physics engine uses to collide with the environment, so we want it to better fit the 3D model. Shrink it a bit by dragging the orange dot in the viewport. My sphere has a radius of about 0.8 meters.

Then, move the shape up so its bottom roughly aligns with the grid's plane.



You can toggle the model's visibility by clicking the eye icon next to the Character or the Pivot nodes.



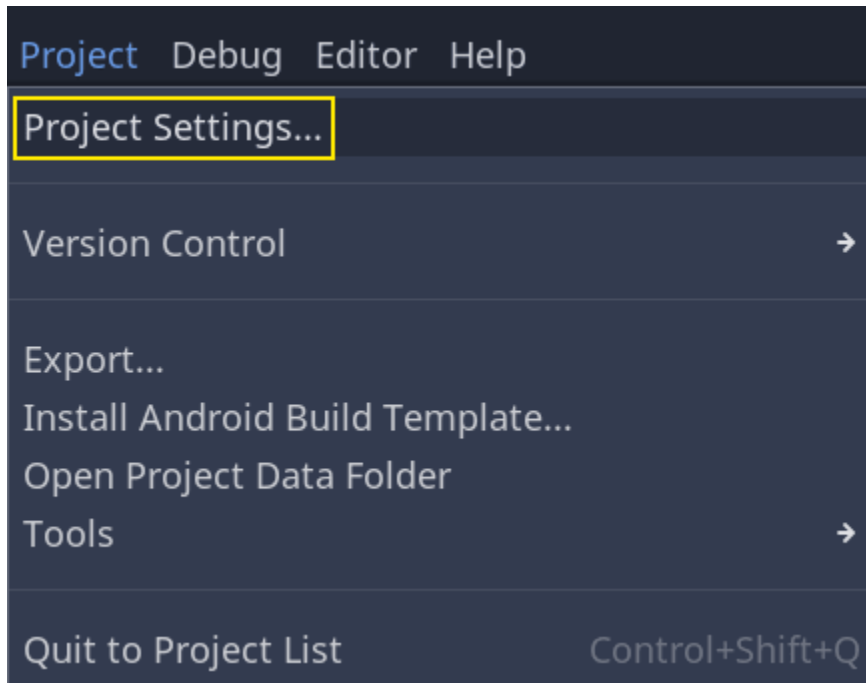
Save the scene as `player.tscn`

With the nodes ready, we can almost get coding. But first, we need to define some input actions.

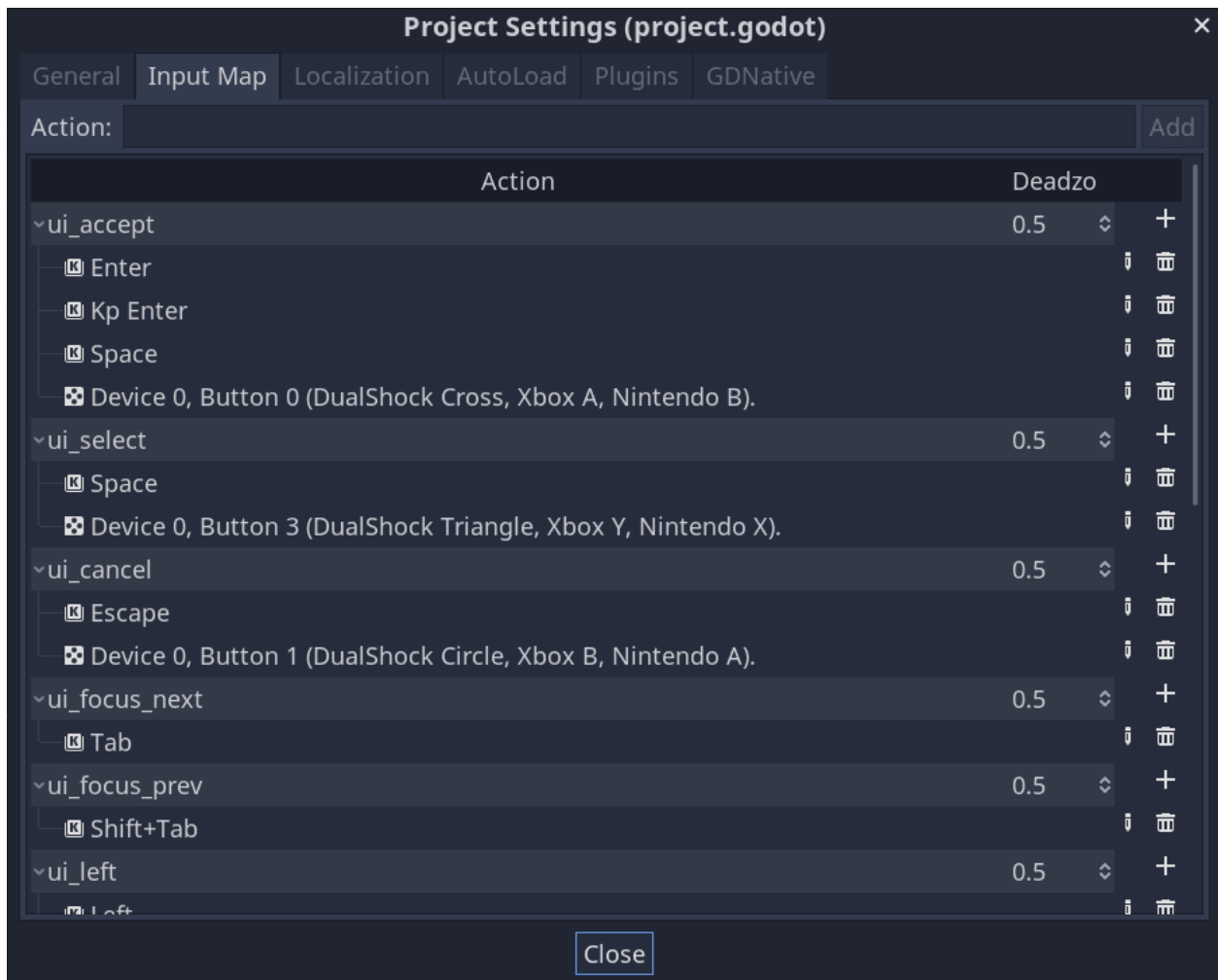
Creating input actions

To move the character, we will listen to the player's input, like pressing the arrow keys. In Godot, while we could write all the key bindings in code, there's a powerful system that allows you to assign a label to a set of keys and buttons. This simplifies our scripts and makes them more readable.

This system is the Input Map. To access its editor, head to the Project menu and select Project Settings.



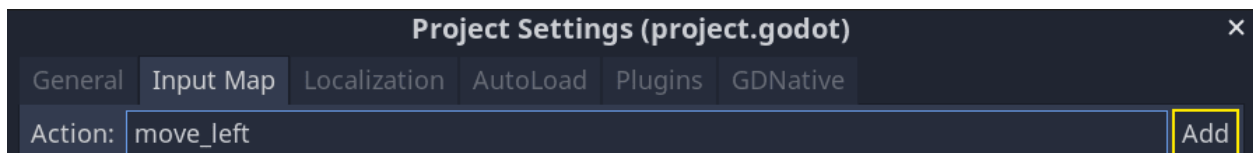
At the top, there are multiple tabs. Click on Input Map. This window allows you to add new actions at the top; they are your labels. In the bottom part, you can bind keys to these actions.



Godot projects come with some predefined actions designed for user interface design, which we could use here. But we're defining our own to support gamepads.

We're going to name our actions `move_left`, `move_right`, `move_forward`, `move_back`, and `jump`.

To add an action, write its name in the bar at the top and press Enter.

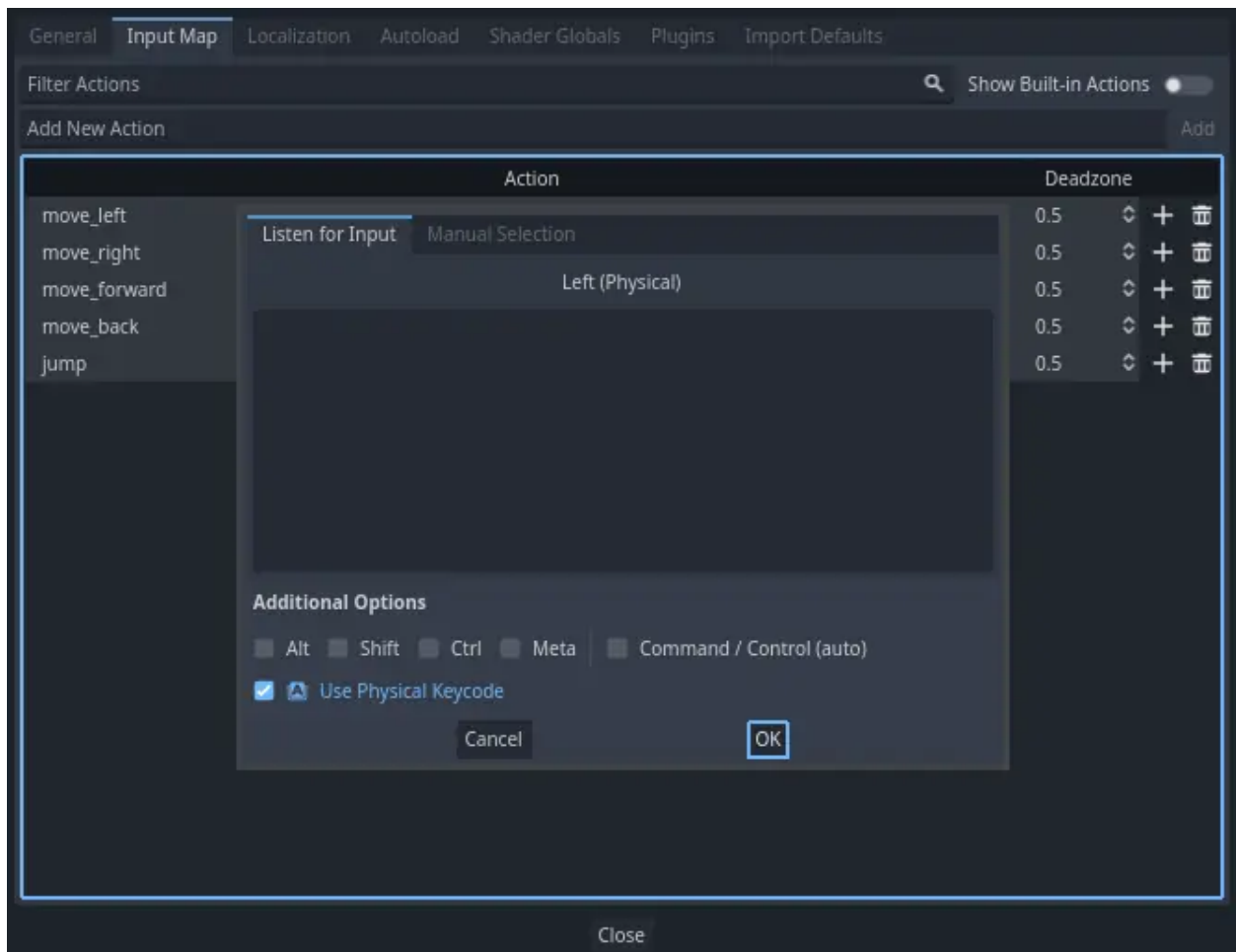


Create the following five actions:

move_left	0.5	⬅	⬆
move_right	0.5	➡	⬆
move_forward	0.5	⬅	➡
move_back	0.5	➡	➡
jump	0.5	⬅	⬆

To bind a key or button to an action, click the "+" button to its right. Do this for `move_left`. Press the left

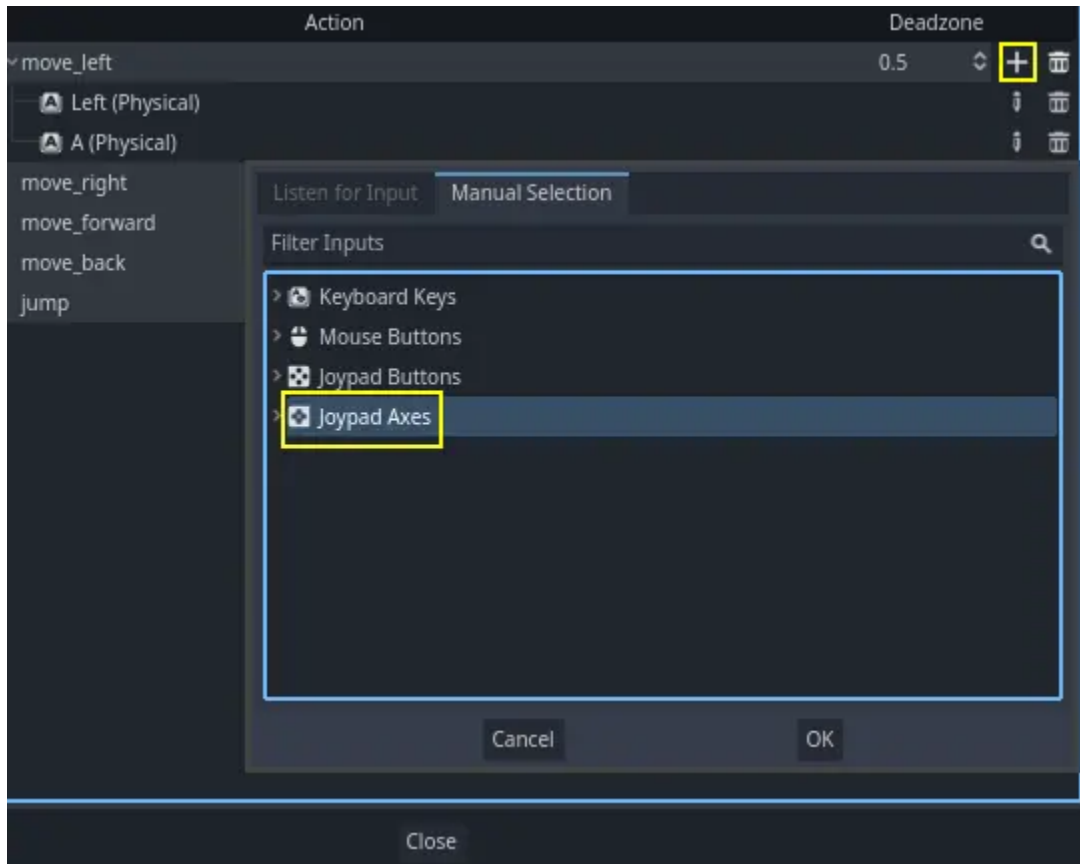
arrow key and click OK.



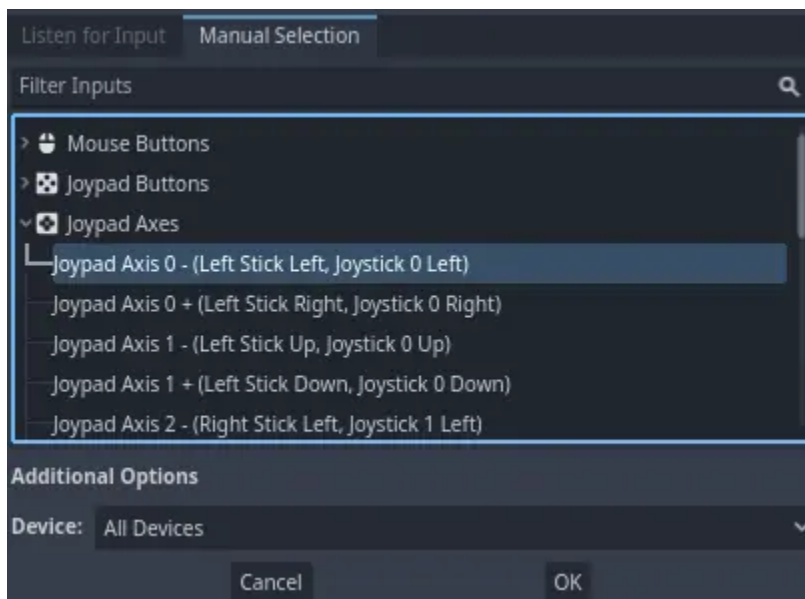
Bind also the A key, onto the action `move_left`.



Let's now add support for a gamepad's left joystick. Click the "+" button again but this time, select Manual Selection -> Joypad Axes.



Select the negative X axis of the left joystick.

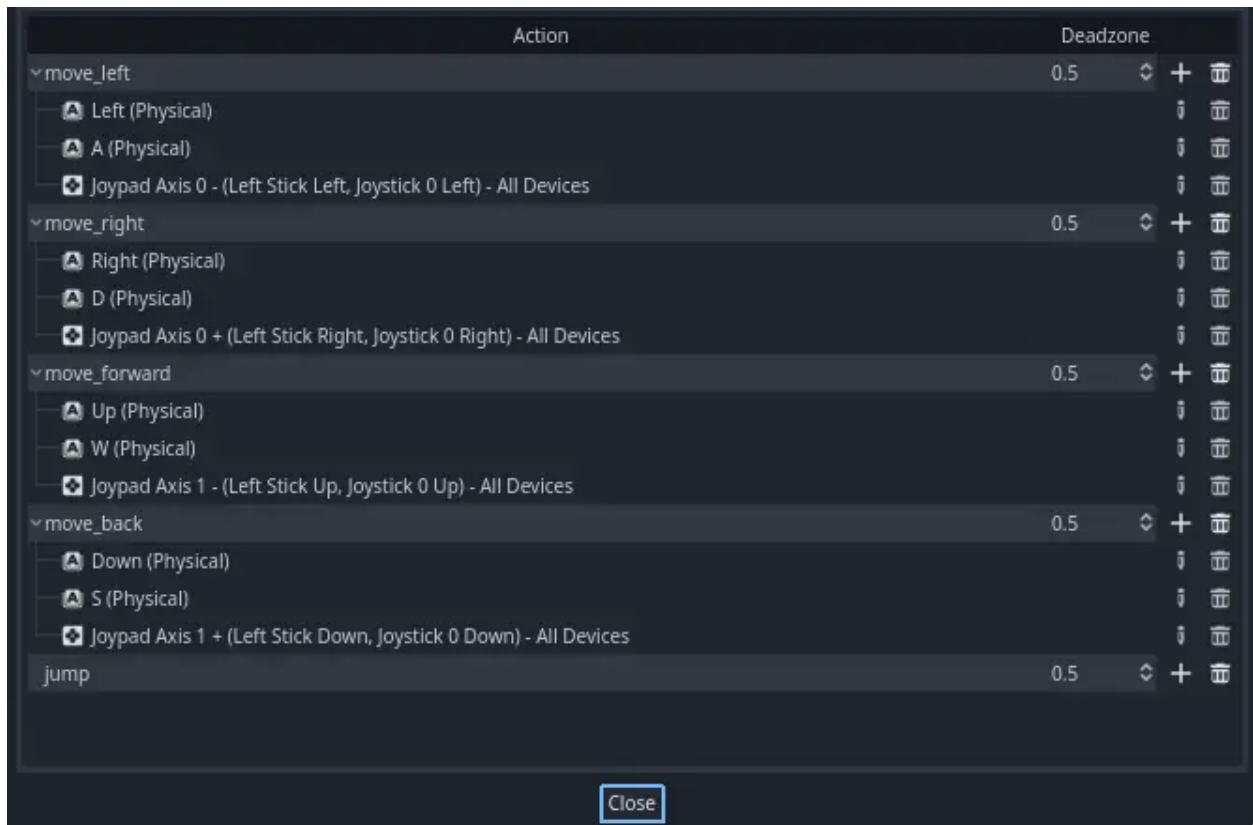


Leave the other values as default and press OK

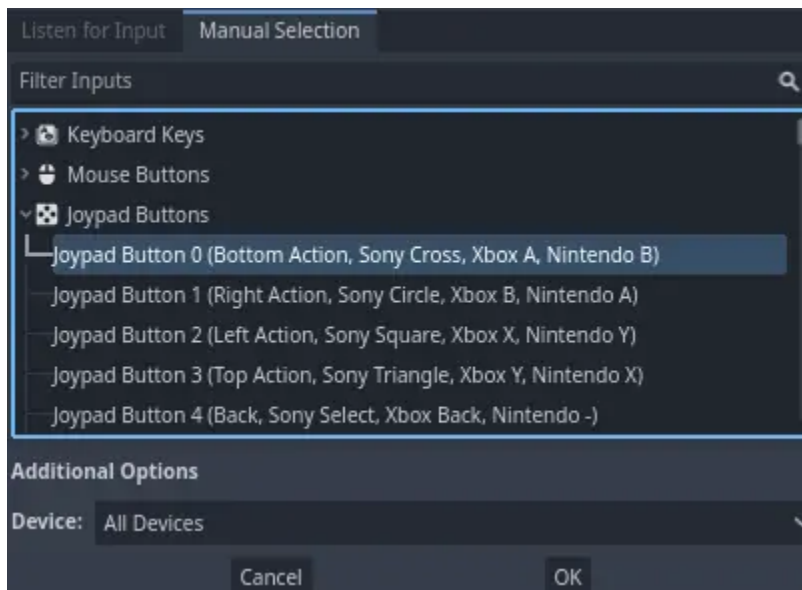
Note: If you want controllers to have different input actions, you should use the Devices option in Additional Options. Device 0 corresponds to the first plugged gamepad, Device 1 corresponds to the second plugged

gamepad, and so on.

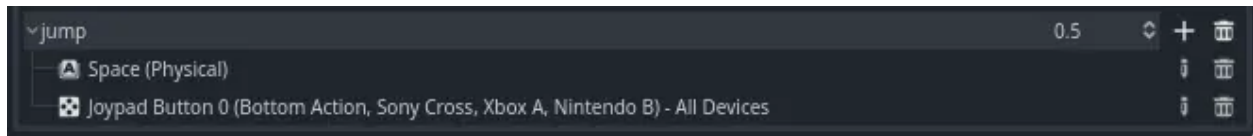
Do the same for the other input actions. For example, bind the right arrow, D, and the left joystick's positive axis to `move_right`. After binding all keys, your interface should look like this.



The final action to set up is the jump action. Bind the Space key and the gamepad's A button.



Your jump input action should look like this.



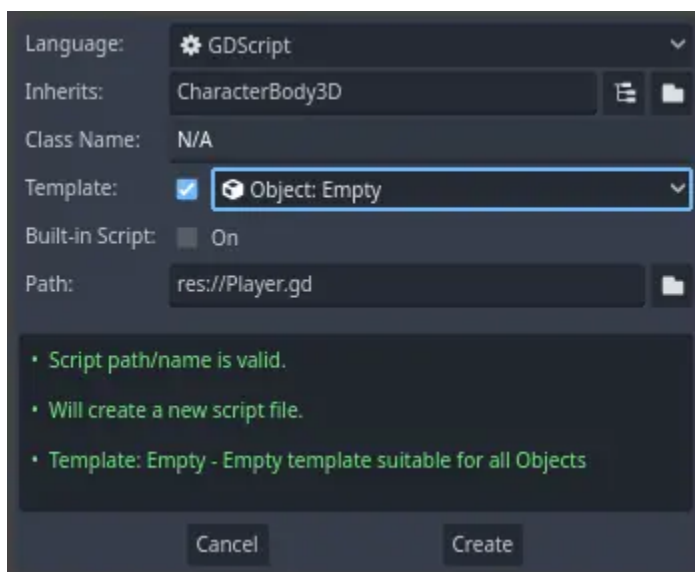
That's all the actions we need for this game. You can use this menu to label any groups of keys and buttons in your projects.

In the next part, we'll code and test the player's movement.

Moving the player with code

It's time to code! We're going to use the input actions we created in the last part to move the character.

Right-click the Player node and select Attach Script to add a new script to it. In the popup, set the Template to Empty before pressing the Create button.



Let's start with the class's properties. We're going to define a movement speed, a fall acceleration representing gravity, and a velocity we'll use to move the character.

GDScript

```
extends CharacterBody3D

# How fast the player moves in meters per second.
@export var speed = 14
# The downward acceleration when in the air, in meters per second squared.
@export var fall_acceleration = 75

var target_velocity = Vector3.ZERO
```

C#

```
using Godot;

public partial class Player : CharacterBody3D
{
```

(continues on next page)

(continued from previous page)

```
// Don't forget to rebuild the project so the editor knows about the new export variable.

// How fast the player moves in meters per second.
[Export]
public int Speed { get; set; } = 14;
// The downward acceleration when in the air, in meters per second squared.
[Export]
public int FallAcceleration { get; set; } = 75;

private Vector3 _targetVelocity = Vector3.Zero;
}
```

These are common properties for a moving body. The `target_velocity` is a 3D vector combining a speed with a direction. Here, we define it as a property because we want to update and reuse its value across frames.

Note: The values are quite different from 2D code because distances are in meters. While in 2D, a thousand units (pixels) may only correspond to half of your screen's width, in 3D, it's a kilometer.

Let's code the movement. We start by calculating the input direction vector using the global `Input` object, in `_physics_process()`.

GDScript

```
func _physics_process(delta):
    # We create a local variable to store the input direction.
    var direction = Vector3.ZERO

    # We check for each move input and update the direction accordingly.
    if Input.is_action_pressed("move_right"):
        direction.x += 1
    if Input.is_action_pressed("move_left"):
        direction.x -= 1
    if Input.is_action_pressed("move_back"):
        # Notice how we are working with the vector's x and z axes.
        # In 3D, the XZ plane is the ground plane.
        direction.z += 1
    if Input.is_action_pressed("move_forward"):
        direction.z -= 1
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // We create a local variable to store the input direction.
    var direction = Vector3.Zero;

    // We check for each move input and update the direction accordingly.
    if (Input.IsActionPressed("move_right"))
    {
        direction.X += 1.0f;
    }
    if (Input.IsActionPressed("move_left"))
```

(continues on next page)

(continued from previous page)

```

{
    direction.X -= 1.0f;
}
if (Input.IsActionPressed("move_back"))
{
    // Notice how we are working with the vector's X and Z axes.
    // In 3D, the XZ plane is the ground plane.
    direction.Z += 1.0f;
}
if (Input.IsActionPressed("move_forward"))
{
    direction.Z -= 1.0f;
}
}

```

Here, we're going to make all calculations using the `_physics_process()` virtual function. Like `_process()`, it allows you to update the node every frame, but it's designed specifically for physics-related code like moving a kinematic or rigid body.

See also:

To learn more about the difference between `_process()` and `_physics_process()`, see [Idle](#) and [Physics Processing](#).

We start by initializing a direction variable to `Vector3.ZERO`. Then, we check if the player is pressing one or more of the `move_*` inputs and update the vector's x and z components accordingly. These correspond to the ground plane's axes.

These four conditions give us eight possibilities and eight possible directions.

In case the player presses, say, both W and D simultaneously, the vector will have a length of about 1.4. But if they press a single key, it will have a length of 1. We want the vector's length to be consistent, and not move faster diagonally. To do so, we can call its `normalized()` method.

GDScript

```

#func _physics_process(delta):
    #...

    if direction != Vector3.ZERO:
        direction = direction.normalized()
        $Pivot.look_at(position + direction, Vector3.UP)

```

C#

```

public override void _PhysicsProcess(double delta)
{
    // ...

    if (direction != Vector3.Zero)
    {
        direction = direction.Normalized();
        GetNode<Node3D>("Pivot").LookAt(Position + direction, Vector3.Up);
    }
}

```

Here, we only normalize the vector if the direction has a length greater than zero, which means the player is pressing a direction key.

In this case, we also get the Pivot node and call its `look_at()` method. This method takes a position in space to look at in global coordinates and the up direction. In this case, we can use the `Vector3.UP` constant.

Note: A node's local coordinates, like position, are relative to their parent. Global coordinates, like `global_position` are relative to the world's main axes you can see in the viewport instead.

In 3D, the property that contains a node's position is `position`. By adding the direction to it, we get a position to look at that's one meter away from the Player.

Then, we update the velocity. We have to calculate the ground velocity and the fall speed separately. Be sure to go back one tab so the lines are inside the `_physics_process()` function but outside the condition we just wrote above.

GDScript

```
func _physics_process(delta):
    #...
    if direction != Vector3.ZERO:
        #...

    # Ground Velocity
    target_velocity.x = direction.x * speed
    target_velocity.z = direction.z * speed

    # Vertical Velocity
    if not is_on_floor(): # If in the air, fall towards the floor. Literally gravity
        target_velocity.y = target_velocity.y - (fall_acceleration * delta)

    # Moving the Character
    velocity = target_velocity
    move_and_slide()
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...
    if (direction != Vector3.Zero)
    {
        // ...
    }

    // Ground velocity
    _targetVelocity.X = direction.X * Speed;
    _targetVelocity.Z = direction.Z * Speed;

    // Vertical velocity
    if (!IsOnFloor()) // If in the air, fall towards the floor. Literally gravity
    {
        _targetVelocity.Y -= FallAcceleration * (float)delta;
    }
}
```

(continues on next page)

(continued from previous page)

```
// Moving the character
Velocity = _target Velocity;
MoveAndSlide();
}
```

The `CharacterBody3D.is_on_floor()` function returns true if the body collided with the floor in this frame. That's why we apply gravity to the Player only while it is in the air.

For the vertical velocity, we subtract the fall acceleration multiplied by the delta time every frame. This line of code will cause our character to fall in every frame, as long as it is not on or colliding with the floor.

The physics engine can only detect interactions with walls, the floor, or other bodies during a given frame if movement and collisions happen. We will use this property later to code the jump.

On the last line, we call `CharacterBody3D.move_and_slide()` which is a powerful method of the `CharacterBody3D` class that allows you to move a character smoothly. If it hits a wall midway through a motion, the engine will try to smooth it out for you. It uses the velocity value native to the `CharacterBody3D`

And that's all the code you need to move the character on the floor.

Here is the complete `Player.gd` code for reference.

GDScript

```
extends CharacterBody3D

# How fast the player moves in meters per second.
@export var speed = 14
# The downward acceleration when in the air, in meters per second squared.
@export var fall_acceleration = 75

var target_velocity = Vector3.ZERO

func _physics_process(delta):
    var direction = Vector3.ZERO

    if Input.is_action_pressed("move_right"):
        direction.x += 1
    if Input.is_action_pressed("move_left"):
        direction.x -= 1
    if Input.is_action_pressed("move_back"):
        direction.z += 1
    if Input.is_action_pressed("move_forward"):
        direction.z -= 1

    if direction != Vector3.ZERO:
        direction = direction.normalized()
        $Pivot.look_at(position + direction, Vector3.UP)

    # Ground Velocity
    target_velocity.x = direction.x * speed
    target_velocity.z = direction.z * speed
```

(continues on next page)

(continued from previous page)

```

# Vertical Velocity
if not is_on_floor(): # If in the air, fall towards the floor. Literally gravity
    target_velocity.y = target_velocity.y - (fall_acceleration * delta)

# Moving the Character
velocity = target_velocity
move_and_slide()

```

C#

```

using Godot;

public partial class Player : CharacterBody3D
{
    // How fast the player moves in meters per second.
    [Export]
    public int Speed { get; set; } = 14;
    // The downward acceleration when in the air, in meters per second squared.
    [Export]
    public int FallAcceleration { get; set; } = 75;

    private Vector3 _targetVelocity = Vector3.Zero;

    public override void _PhysicsProcess(double delta)
    {
        var direction = Vector3.Zero;

        if (Input.IsActionPressed("move_right"))
        {
            direction.X += 1.0f;
        }
        if (Input.IsActionPressed("move_left"))
        {
            direction.X -= 1.0f;
        }
        if (Input.IsActionPressed("move_back"))
        {
            direction.Z += 1.0f;
        }
        if (Input.IsActionPressed("move_forward"))
        {
            direction.Z -= 1.0f;
        }

        if (direction != Vector3.Zero)
        {
            direction = direction.Normalized();
            GetNode<Node3D>("Pivot").LookAt(Position + direction, Vector3.Up);
        }

        // Ground velocity
        _targetVelocity.X = direction.X * Speed;
    }
}

```

(continues on next page)

(continued from previous page)

```

    _targetVelocity.Z = direction.Z * Speed;

    // Vertical velocity
    if (!IsOnFloor()) // If in the air, fall towards the floor. Literally gravity
    {
        _targetVelocity.Y -= FallAcceleration * (float)delta;
    }

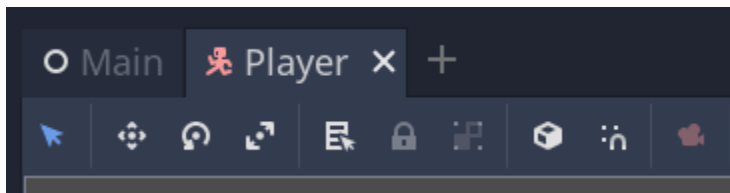
    // Moving the character
    Velocity = _targetVelocity;
    MoveAndSlide();
}
}

```

Testing our player's movement

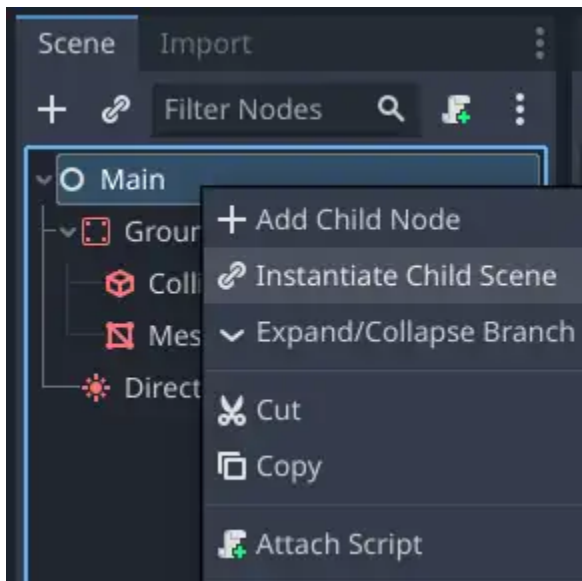
We're going to put our player in the Main scene to test it. To do so, we need to instantiate the player and then add a camera. Unlike in 2D, in 3D, you won't see anything if your viewport doesn't have a camera pointing at something.

Save your Player scene and open the Main scene. You can click on the Main tab at the top of the editor to do so.



If you closed the scene before, head to the FileSystem dock and double-click `main.tscn` to re-open it.

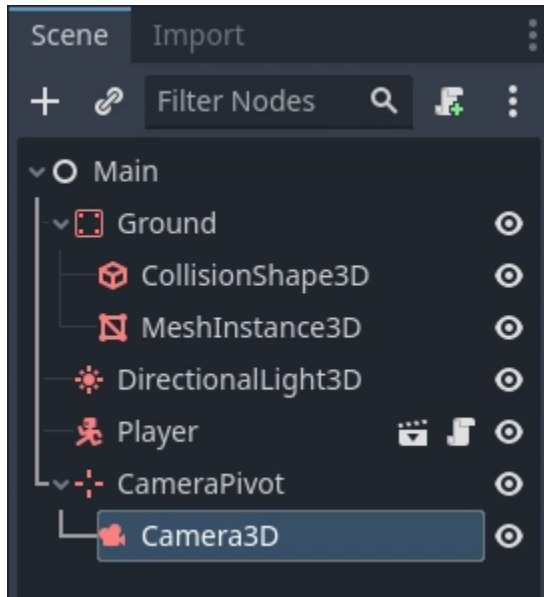
To instantiate the Player, right-click on the Main node and select `Instantiate Child Scene`.



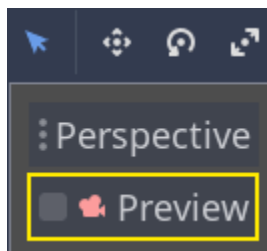
In the popup, double-click `player.tscn`. The character should appear in the center of the viewport.

Adding a camera

Let's add the camera next. Like we did with our Player's Pivot, we're going to create a basic rig. Right-click on the Main node again and select Add Child Node. Create a new Marker3D, and name it CameraPivot. Select CameraPivot and add a child node Camera3D to it. Your scene tree should look like this.

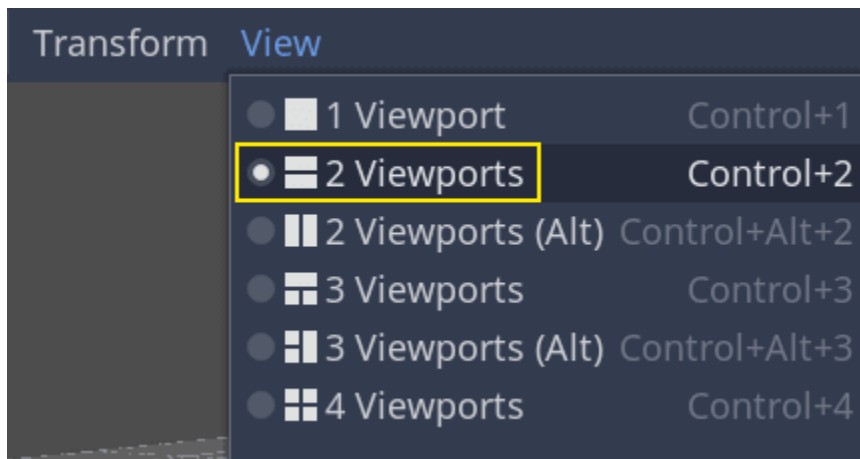
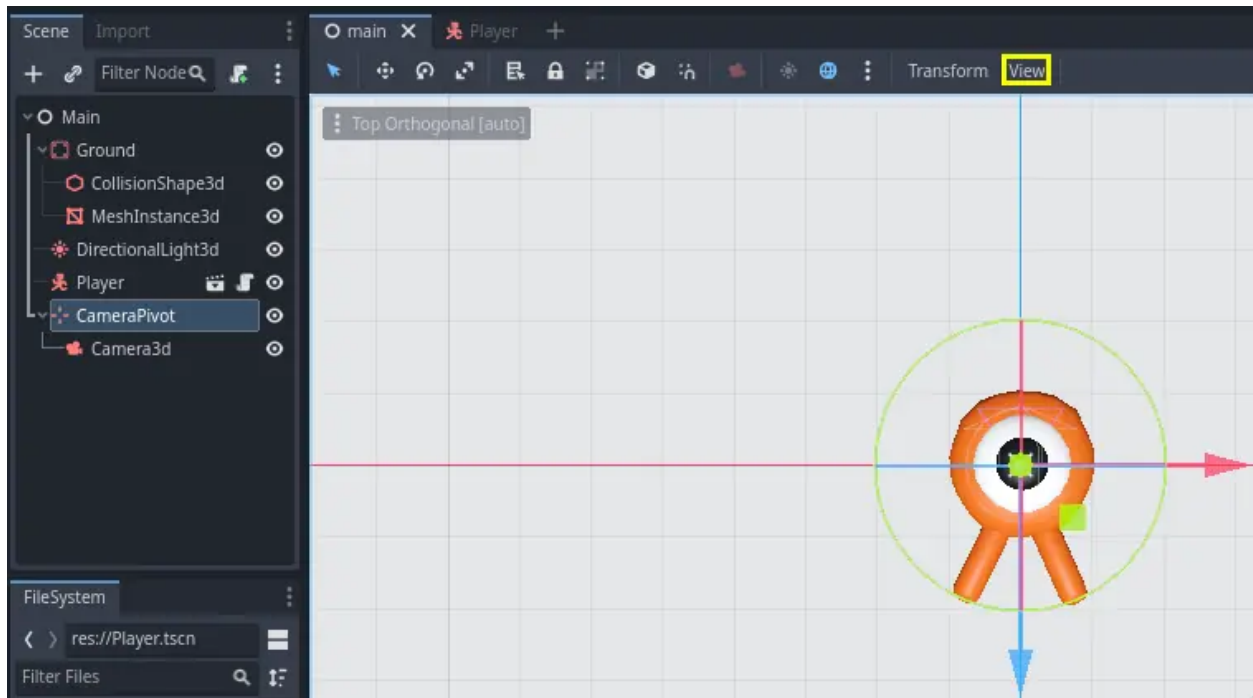


Notice the Preview checkbox that appears in the top-left when you have the Camera selected. You can click it to preview the in-game camera projection.

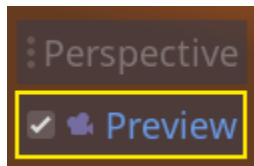


We're going to use the Pivot to rotate the camera as if it was on a crane. Let's first split the 3D view to be able to freely navigate the scene and see what the camera sees.

In the toolbar right above the viewport, click on View, then 2 Viewports. You can also press Ctrl + 2 (Cmd + 2 on macOS).



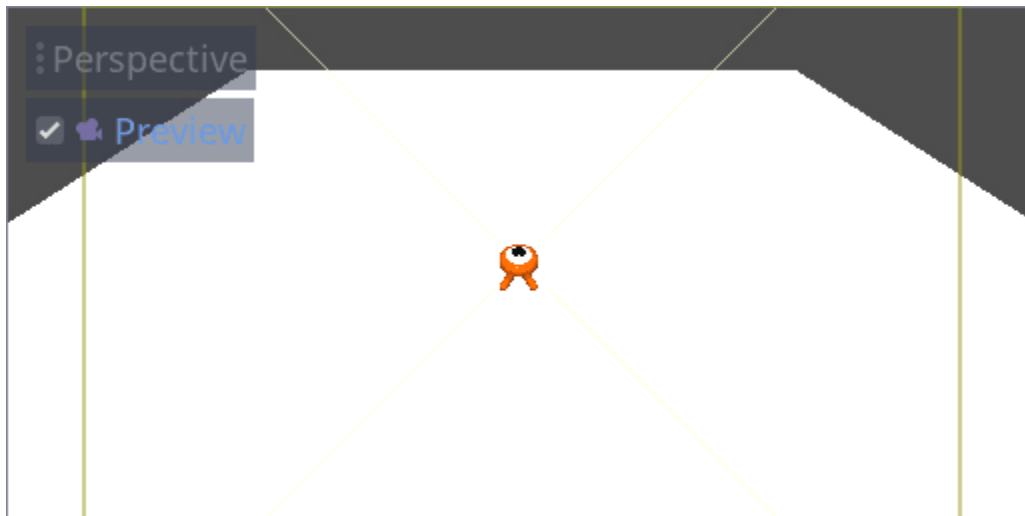
On the bottom view, select your Camera3D and turn on camera Preview by clicking the checkbox.



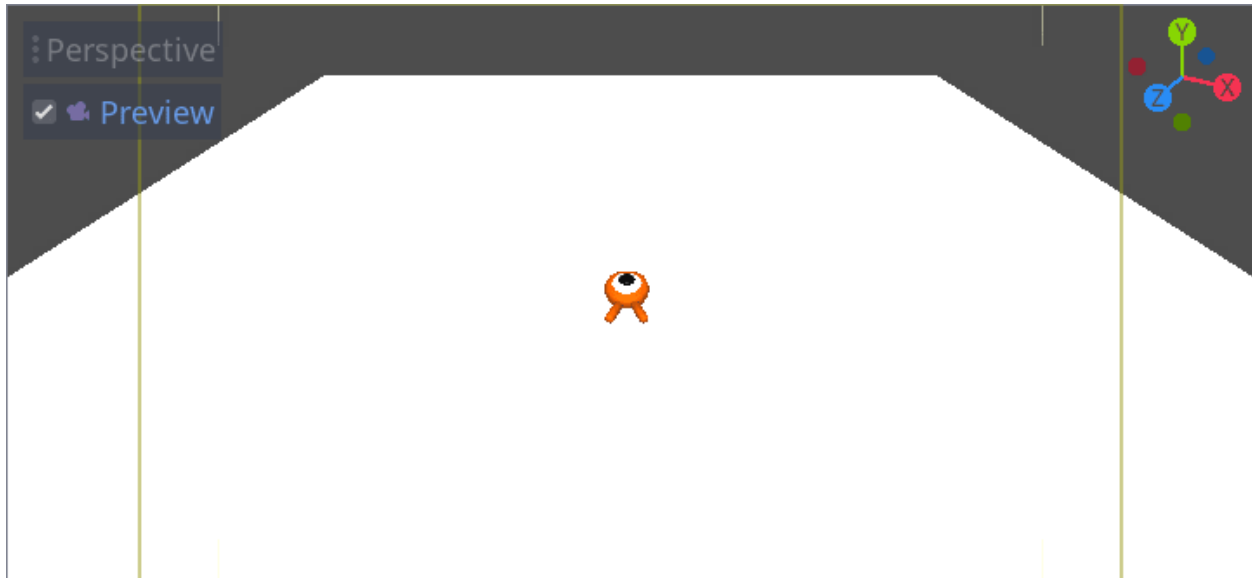
In the top view, move the camera about 19 units on the Z axis (the blue one).



Here's where the magic happens. Select the CameraPivot and rotate it -45 degrees around the X axis (using the red circle). You'll see the camera move as if it was attached to a crane.



You can run the scene by pressing F6 and press the arrow keys to move the character.

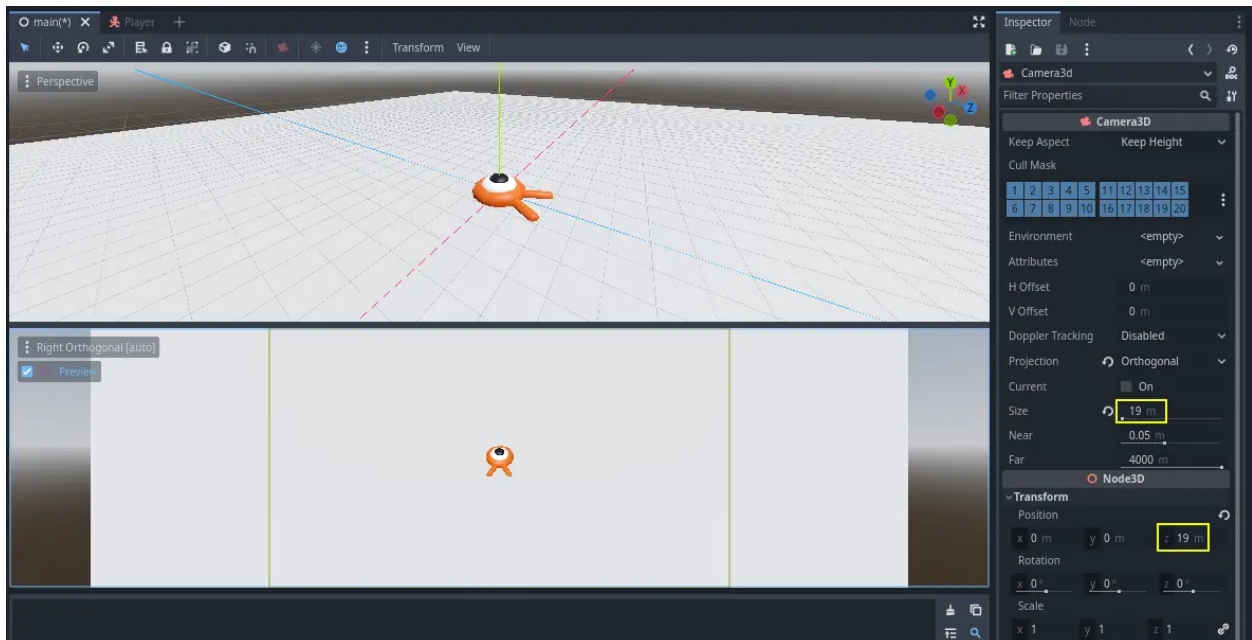


We can see some empty space around the character due to the perspective projection. In this game, we're going to use an orthographic projection instead to better frame the gameplay area and make it easier for the player to read distances.

Select the Camera again and in the Inspector, set the Projection to Orthogonal and the Size to 19. The character should now look flatter and the ground should fill the background.

Note: When using an orthogonal camera in Godot 4, directional shadow quality is dependent on the camera's Far value. The higher the Far value, the further away the camera will be able to see. However, higher Far values also decrease shadow quality as the shadow rendering has to cover a greater distance.

If directional shadows look too blurry after switching to an orthogonal camera, decrease the camera's Far property to a lower value such as 100. Don't decrease this Far property too much, or objects in the distance will start disappearing.



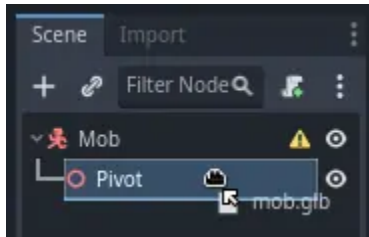
Test your scene and you should be able to move in all 8 directions and not glitch through the floor! Ultimately, we have both player movement and the view in place. Next, we will work on the monsters.

Designing the mob scene

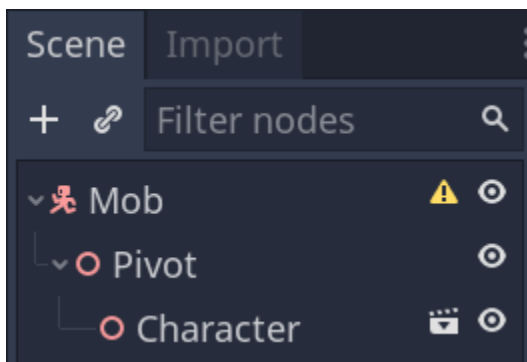
In this part, you're going to code the monsters, which we'll call mobs. In the next lesson, we'll spawn them randomly around the playable area.

Let's design the monsters themselves in a new scene. The node structure is going to be similar to the `player.tscn` scene.

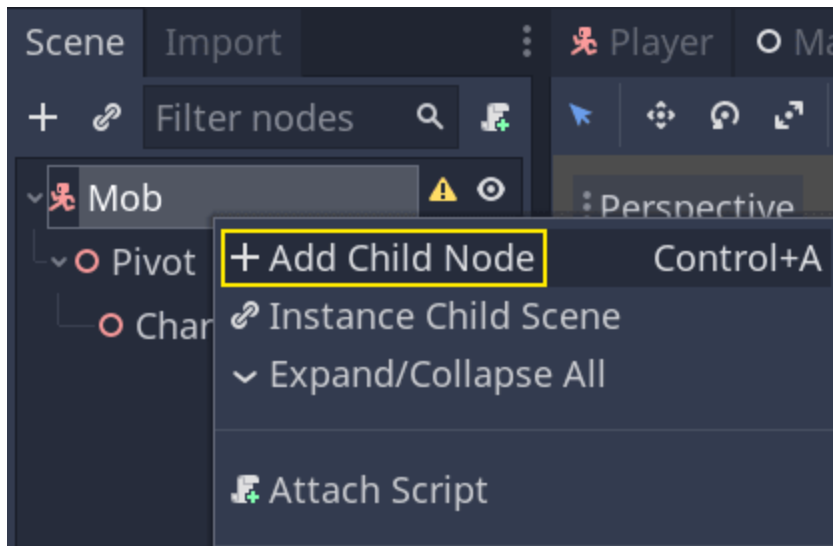
Create a scene with, once again, a `CharacterBody3D` node as its root. Name it `Mob`. Add a child node `Node3D`, name it `Pivot`. And drag and drop the file `mob.glb` from the `FileSystem` dock onto the `Pivot` to add the monster's 3D model to the scene.



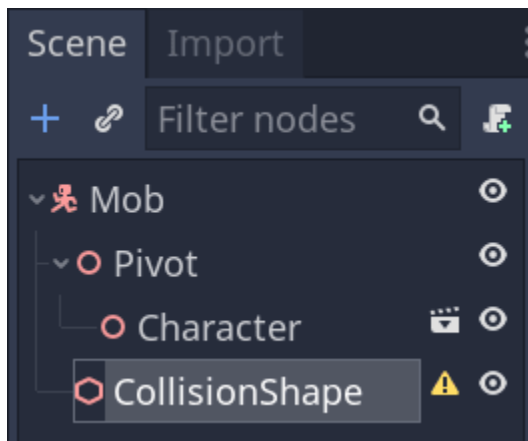
You can rename the newly created mob node into `Character`.



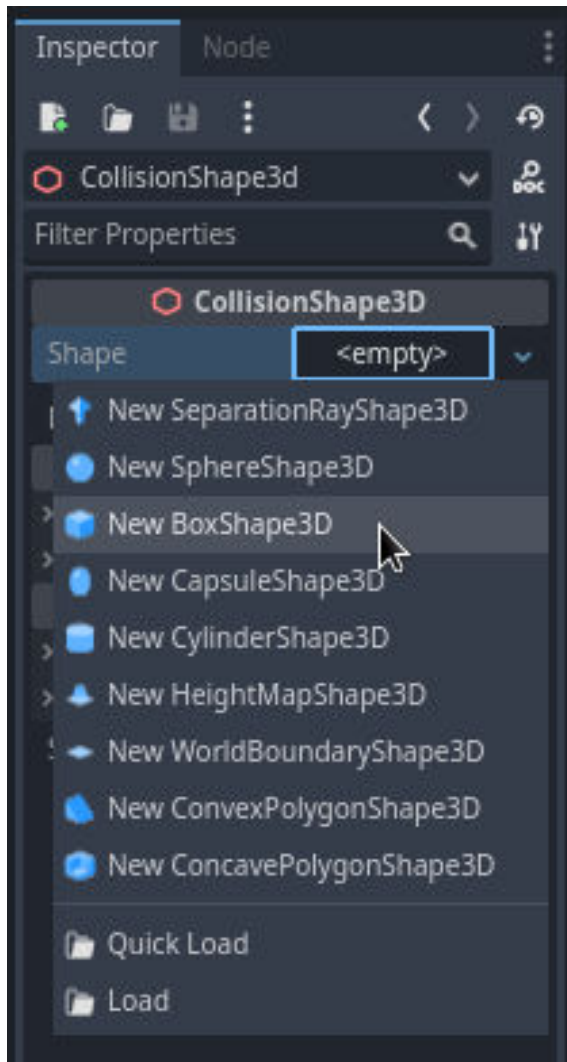
We need a collision shape for our body to work. Right-click on the `Mob` node, the scene's root, and click `Add Child Node`.



Add a CollisionShape3D.

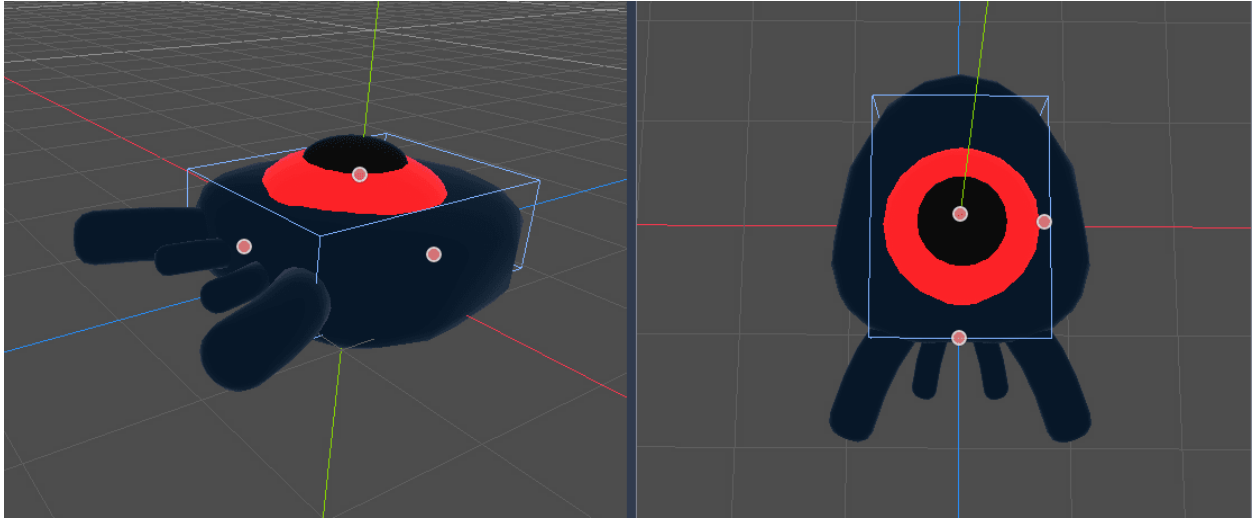


In the Inspector, assign a BoxShape3D to the Shape property.



We should change its size to fit the 3D model better. You can do so interactively by clicking and dragging on the orange dots.

The box should touch the floor and be a little thinner than the model. Physics engines work in such a way that if the player's sphere touches even the box's corner, a collision will occur. If the box is a little too big compared to the 3D model, you may die at a distance from the monster, and the game will feel unfair to the players.



Notice that my box is taller than the monster. It is okay in this game because we're looking at the scene from above and using a fixed perspective. Collision shapes don't have to match the model exactly. It's the way the game feels when you test it that should dictate their form and size.

Removing monsters off-screen

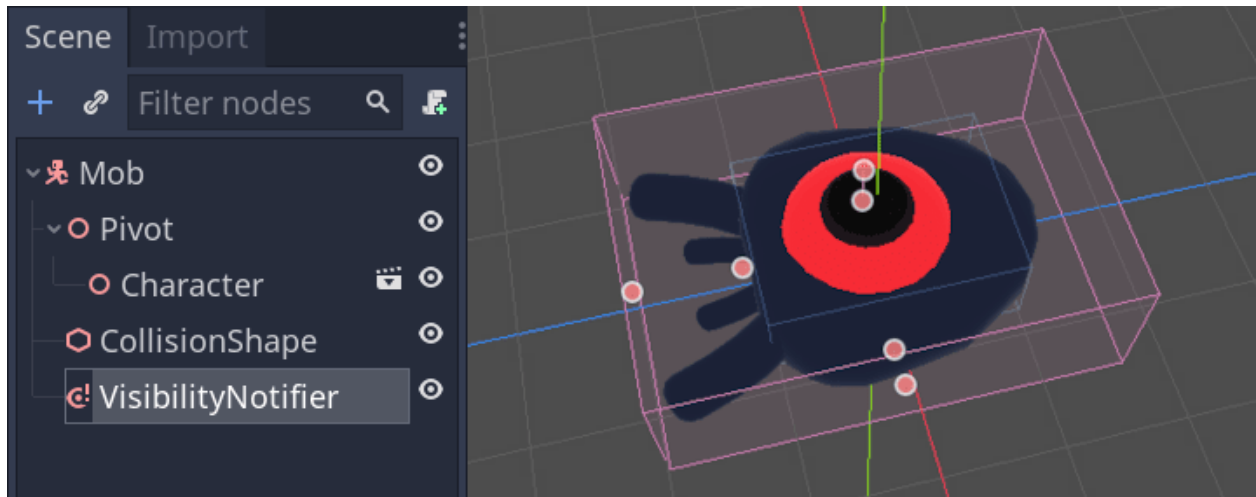
We're going to spawn monsters at regular time intervals in the game level. If we're not careful, their count could increase to infinity, and we don't want that. Each mob instance has both a memory and a processing cost, and we don't want to pay for it when the mob is outside the screen.

Once a monster leaves the screen, we don't need it anymore, so we should delete it. Godot has a node that detects when objects leave the screen, `VisibleOnScreenNotifier3D`, and we're going to use it to destroy our mobs.

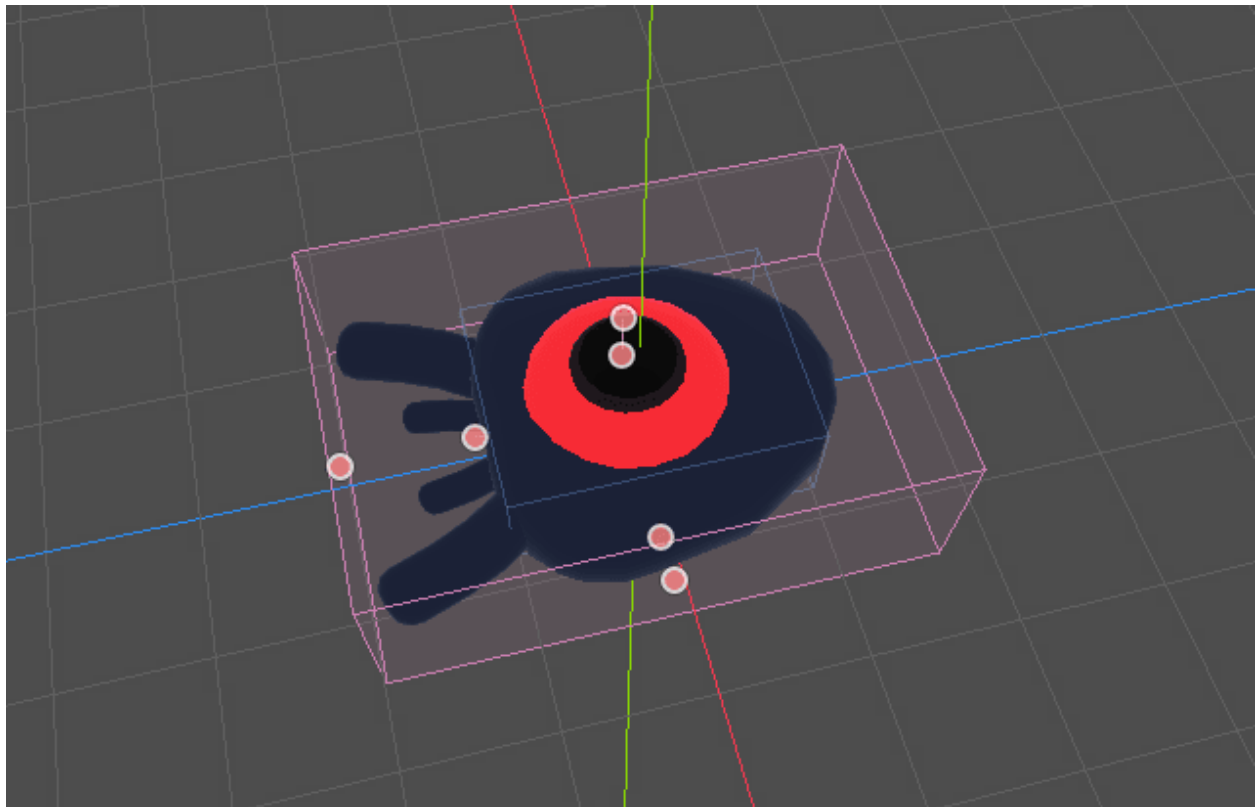
Note: When you keep instancing an object, there's a technique you can use to avoid the cost of creating and destroying instances all the time called pooling. It consists of pre-creating an array of objects and reusing them over and over.

When working with GDScript, you don't need to worry about this. The main reason to use pools is to avoid freezes with garbage-collected languages like C# or Lua. GDScript uses a different technique to manage memory, reference counting, which doesn't have that caveat. You can learn more about that here: [Memory management](#).

Select the Mob node and add a child node `VisibleOnScreenNotifier3D`. Another box, pink this time, appears. When this box completely leaves the screen, the node will emit a signal.



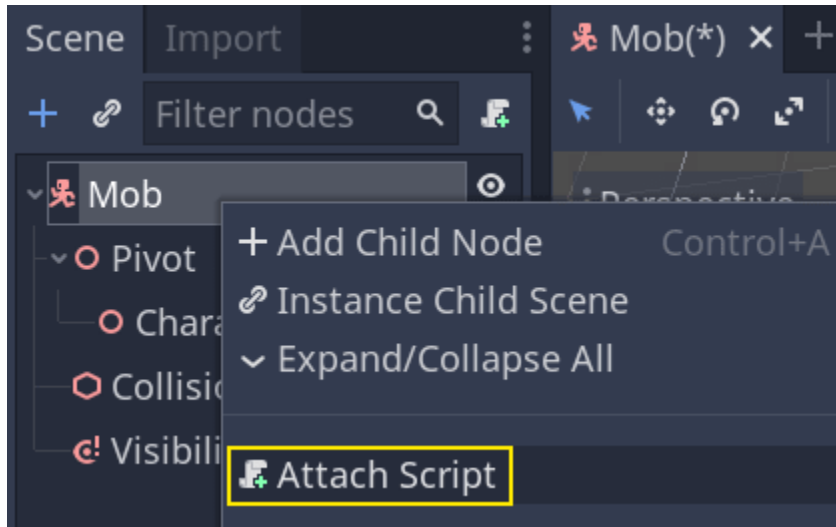
Resize it using the orange dots until it covers the entire 3D model.



Coding the mob's movement

Let's implement the monster's motion. We're going to do this in two steps. First, we'll write a script on the Mob that defines a function to initialize the monster. We'll then code the randomized spawn mechanism in the main.tscn scene and call the function from there.

Attach a script to the Mob.



Here's the movement code to start with. We define two properties, `min_speed` and `max_speed`, to define a random speed range, which we will later use to define `CharacterBody3D.velocity`.

GScript

```
extends CharacterBody3D

# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18

func _physics_process(_delta):
    move_and_slide()
```

C#

```
using Godot;

public partial class Mob : CharacterBody3D
{
    // Don't forget to rebuild the project so the editor knows about the new export variable.

    // Minimum speed of the mob in meters per second
    [Export]
    public int MinSpeed { get; set; } = 10;
    // Maximum speed of the mob in meters per second
    [Export]
    public int MaxSpeed { get; set; } = 18;
```

(continues on next page)

(continued from previous page)

```

public override void _PhysicsProcess(double delta)
{
    MoveAndSlide();
}
}

```

Similarly to the player, we move the mob every frame by calling the function `CharacterBody3D.move_and_slide()`. This time, we don't update the velocity every frame; we want the monster to move at a constant speed and leave the screen, even if it were to hit an obstacle.

We need to define another function to calculate the `CharacterBody3D.velocity`. This function will turn the monster towards the player and randomize both its angle of motion and its velocity.

The function will take a `start_position`, the mob's spawn position, and the `player_position` as its arguments.

We position the mob at `start_position` and turn it towards the player using the `look_at_from_position()` method, and randomize the angle by rotating a random amount around the Y axis. Below, `randf_range()` outputs a random value between $-\pi / 4$ radians and $\pi / 4$ radians.

GScript

```

# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the player.
    look_at_from_position(start_position, player_position, Vector3.UP)
    # Rotate this mob randomly within range of -45 and +45 degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))

```

C#

```

// This function will be called from the Main scene.
public void Initialize(Vector3 startPosition, Vector3 playerPosition)
{
    // We position the mob by placing it at startPosition
    // and rotate it towards playerPosition, so it looks at the player.
    LookAtFromPosition(startPosition, playerPosition, Vector3.Up);
    // Rotate this mob randomly within range of -45 and +45 degrees,
    // so that it doesn't move directly towards the player.
    RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi / 4.0));
}

```

We got a random position, now we need a `random_speed`. `randi_range()` will be useful as it gives random int values, and we will use `min_speed` and `max_speed`. `random_speed` is just an integer, and we just use it to multiply our `CharacterBody3D.velocity`. After `random_speed` is applied, we rotate `CharacterBody3D.velocity Vector3` towards the player.

GScript

```

func initialize(start_position, player_position):
    # ...

    # We calculate a random speed (integer)

```

(continues on next page)

(continued from previous page)

```

var random_speed = randi_range(min_speed, max_speed)
# We calculate a forward velocity that represents the speed.
velocity = Vector3.FORWARD * random_speed
# We then rotate the velocity vector based on the mob's Y rotation
# in order to move in the direction the mob is looking.
velocity = velocity.rotated(Vector3.UP, rotation.y)

```

C#

```

public void Initialize(Vector3 startPosition, Vector3 playerPosition)
{
    // ...

    // We calculate a random speed (integer).
    int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
    // We calculate a forward velocity that represents the speed.
    Velocity = Vector3.Forward * randomSpeed;
    // We then rotate the velocity vector based on the mob's Y rotation
    // in order to move in the direction the mob is looking.
    Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);
}

```

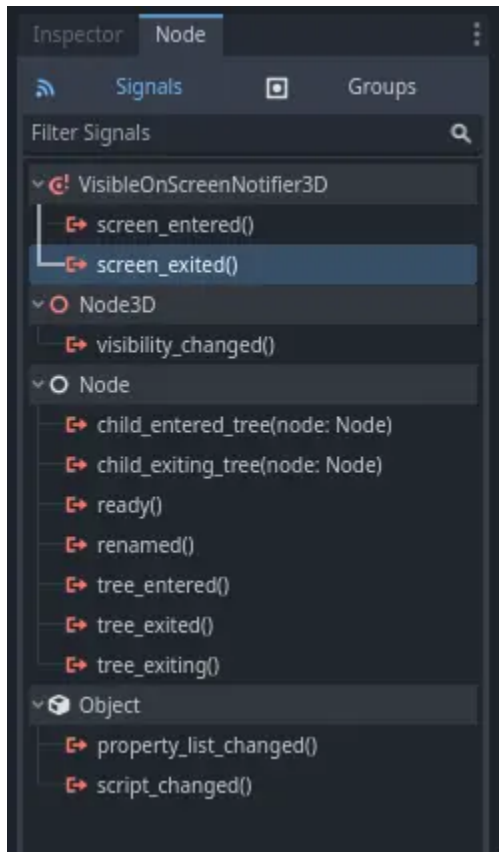
Leaving the screen

We still have to destroy the mobs when they leave the screen. To do so, we'll connect our VisibleOnScreenNotifier3D node's screen_exited signal to the Mob.

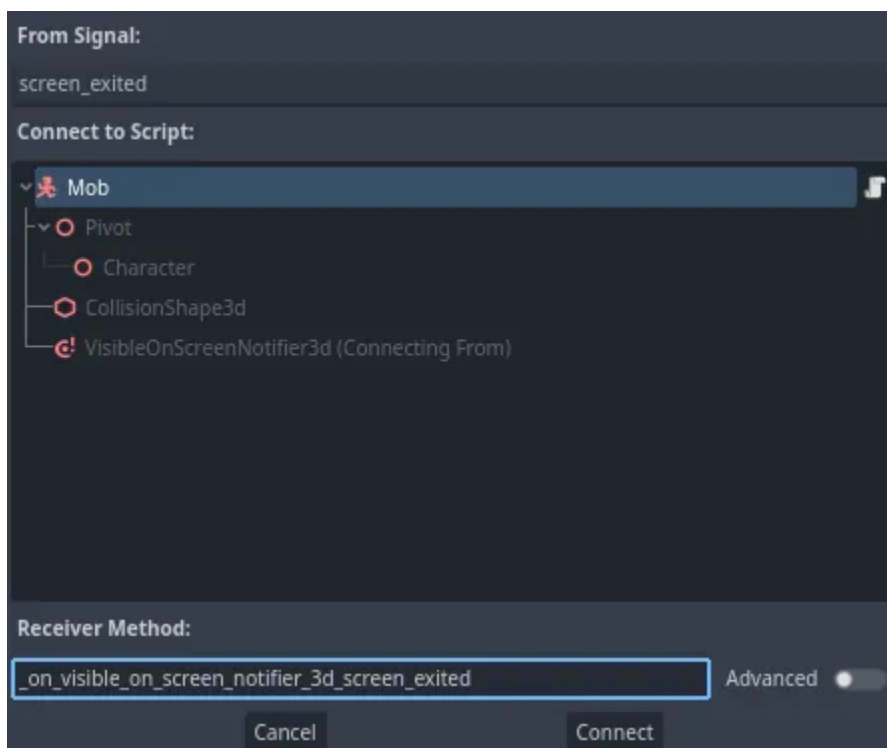
Head back to the 3D viewport by clicking on the 3D label at the top of the editor. You can also press Ctrl + F2 (Alt + 2 on macOS).



Select the VisibleOnScreenNotifier3D node and on the right side of the interface, navigate to the Node dock. Double-click the screen_exited() signal.



Connect the signal to the Mob



This will take you back to the script editor and add a new function for you,

`_on_visible_on_screen_notifier_3d_screen_exited()`. From it, call the `queue_free()` method. This function destroy the instance it's called on.

GDScript

```
func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()
```

C#

```
// We also specified this function name in PascalCase in the editor's connection window
private void OnVisibilityNotifierScreenExited()
{
    QueueFree();
}
```

Our monster is ready to enter the game! In the next part, you will spawn monsters in the game level.

Here is the complete Mob.gd script for reference.

GDScript

```
extends CharacterBody3D

# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18

func _physics_process(_delta):
    move_and_slide()

# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the player.
    look_at_from_position(start_position, player_position, Vector3.UP)
    # Rotate this mob randomly within range of -90 and +90 degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))

    # We calculate a random speed (integer)
    var random_speed = randi_range(min_speed, max_speed)
    # We calculate a forward velocity that represents the speed.
    velocity = Vector3.FORWARD * random_speed
    # We then rotate the velocity vector based on the mob's Y rotation
    # in order to move in the direction the mob is looking.
    velocity = velocity.rotated(Vector3.UP, rotation.y)

func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()
```

C#

```
using Godot;

public partial class Mob : CharacterBody3D
{
    // Minimum speed of the mob in meters per second.
    [Export]
    public int MinSpeed { get; set; } = 10;
    // Maximum speed of the mob in meters per second.
    [Export]
    public int MaxSpeed { get; set; } = 18;

    public override void _PhysicsProcess(double delta)
    {
        MoveAndSlide();
    }

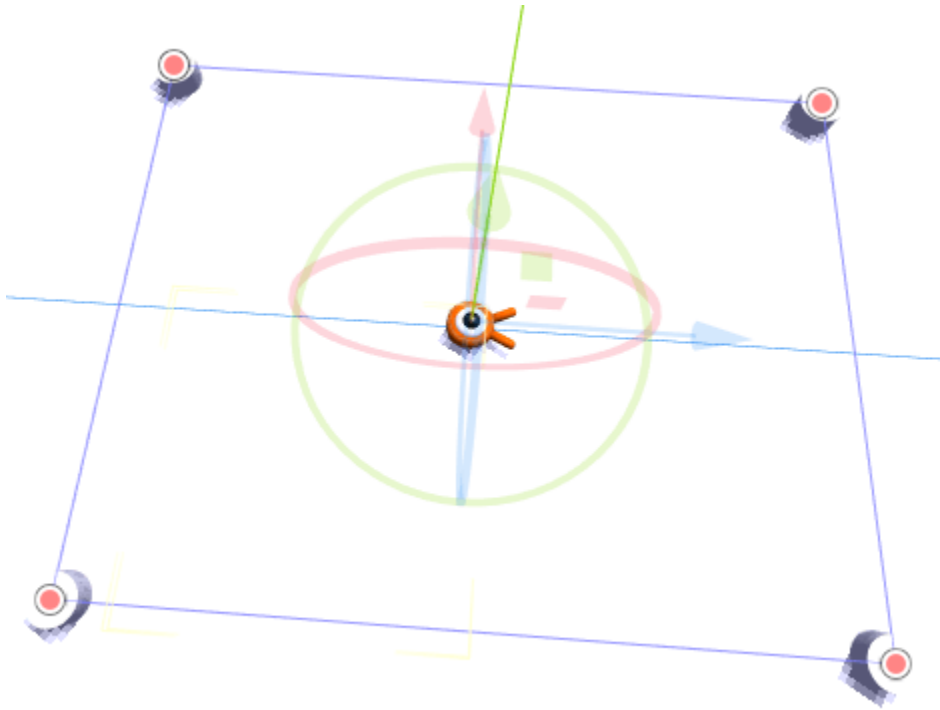
    // This function will be called from the Main scene.
    public void Initialize(Vector3 startPosition, Vector3 playerPosition)
    {
        // We position the mob by placing it at startPosition
        // and rotate it towards playerPosition, so it looks at the player.
        LookAtFromPosition(startPosition, playerPosition, Vector3.Up);
        // Rotate this mob randomly within range of -90 and +90 degrees,
        // so that it doesn't move directly towards the player.
        RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi / 4.0));

        // We calculate a random speed (integer).
        int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
        // We calculate a forward velocity that represents the speed.
        Velocity = Vector3.Forward * randomSpeed;
        // We then rotate the velocity vector based on the mob's Y rotation
        // in order to move in the direction the mob is looking.
        Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);
    }

    // We also specified this function name in PascalCase in the editor's connection window
    private void OnVisibilityNotifierScreenExited()
    {
        QueueFree();
    }
}
```

Spawning monsters

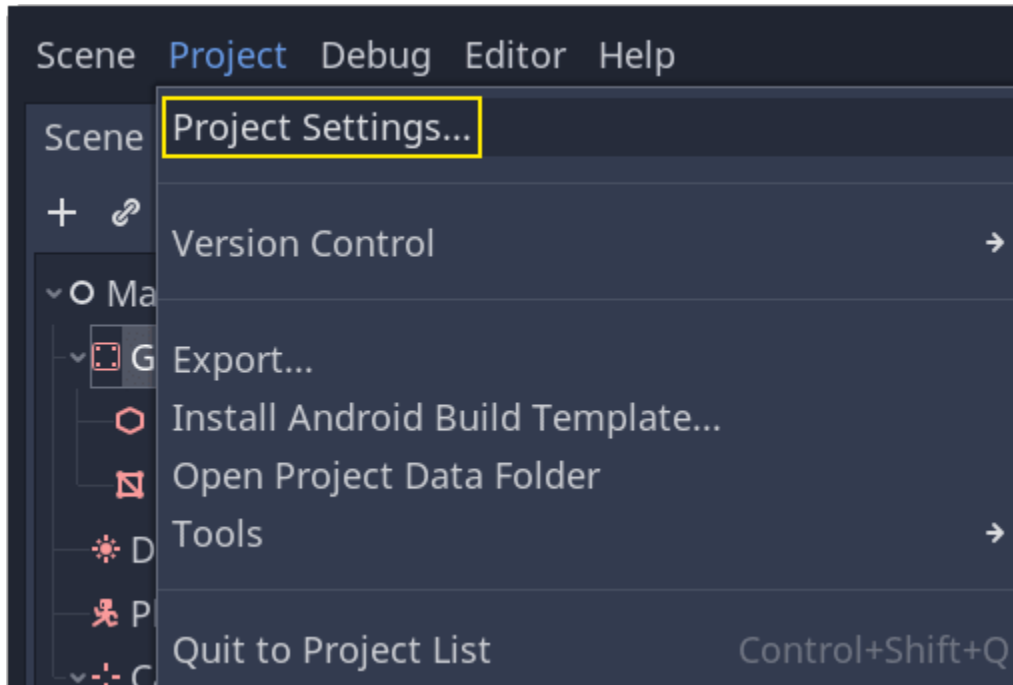
In this part, we're going to spawn monsters along a path randomly. By the end, you will have monsters roaming the game board.



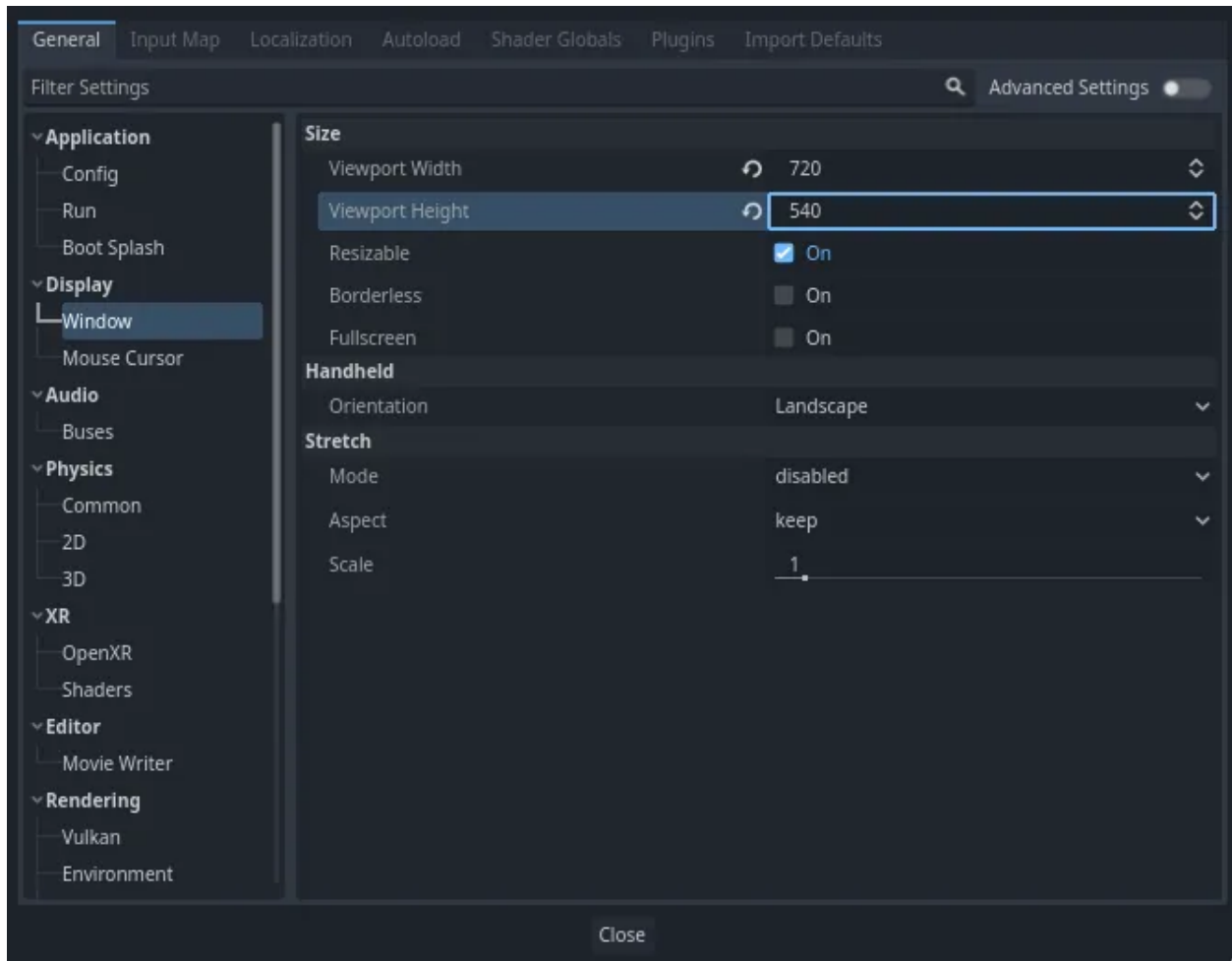
Double-click on `main.tscn` in the FileSystem dock to open the Main scene.

Before drawing the path, we're going to change the game resolution. Our game has a default window size of 1152x648. We're going to set it to 720x540, a nice little box.

Go to Project -> Project Settings.



In the left menu, navigate down to Display -> Window. On the right, set the Width to 720 and the Height to 540.

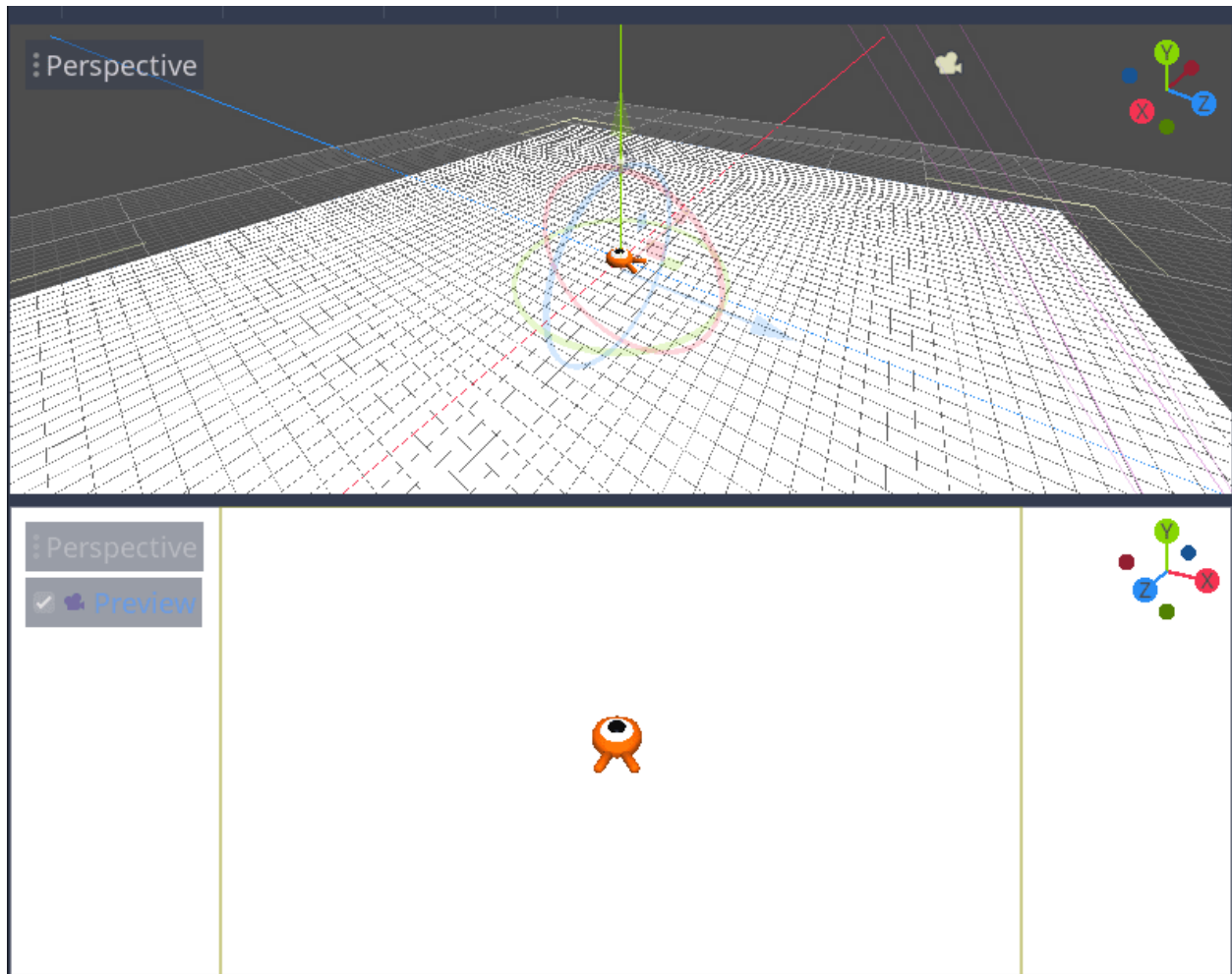


Creating the spawn path

Like you did in the 2D game tutorial, you're going to design a path and use a `PathFollow3D` node to sample random locations on it.

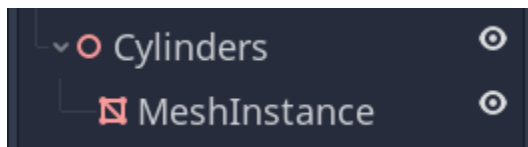
In 3D though, it's a bit more complicated to draw the path. We want it to be around the game view so monsters appear right outside the screen. But if we draw a path, we won't see it from the camera preview.

To find the view's limits, we can use some placeholder meshes. Your viewport should still be split into two parts, with the camera preview at the bottom. If that isn't the case, press `Ctrl + 2` (`Cmd + 2` on macOS) to split the view into two. Select the `Camera3D` node and click the `Preview` checkbox in the bottom viewport.

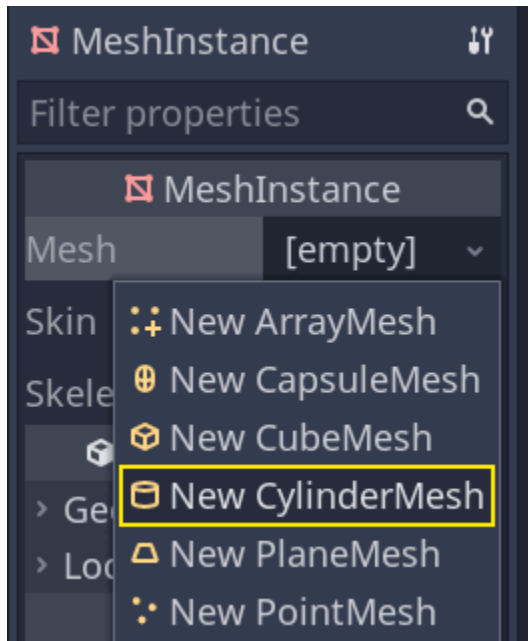


Adding placeholder cylinders

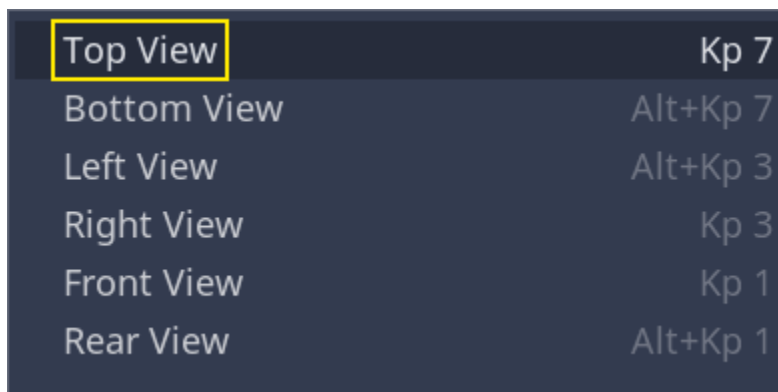
Let's add the placeholder meshes. Add a new Node3D as a child of the Main node and name it Cylinders. We'll use it to group the cylinders. Select Cylinders and add a child node MeshInstance3D



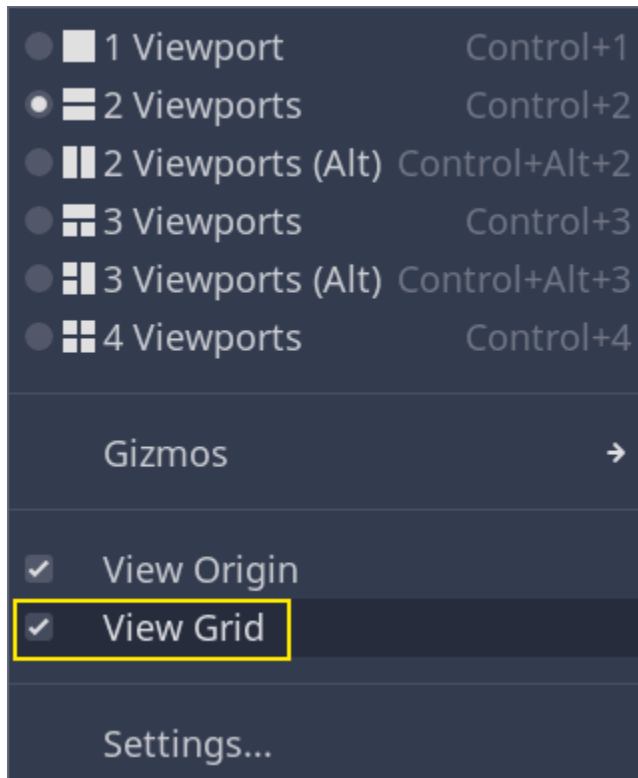
In the Inspector, assign a CylinderMesh to the Mesh property.



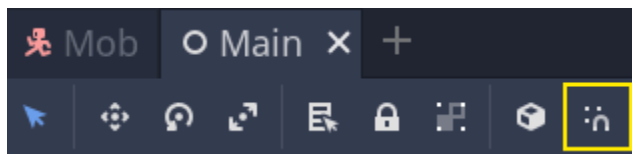
Set the top viewport to the top orthogonal view using the menu in the viewport's top-left corner. Alternatively, you can press the keypad's 7 key.



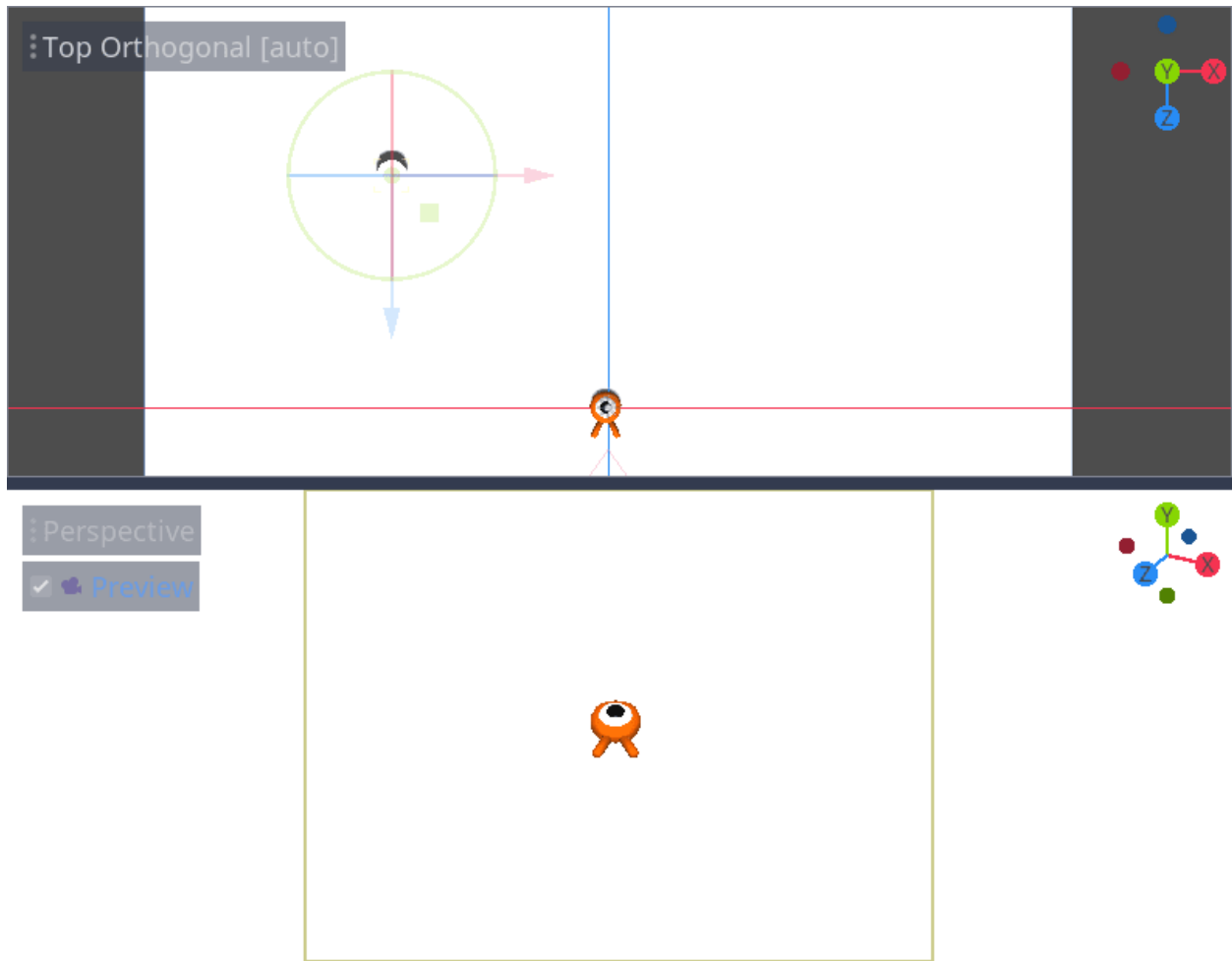
The grid may be distracting. You can toggle it by going to the View menu in the toolbar and clicking View Grid.



You now want to move the cylinder along the ground plane, looking at the camera preview in the bottom viewport. I recommend using grid snap to do so. You can toggle it by clicking the magnet icon in the toolbar or pressing Y.

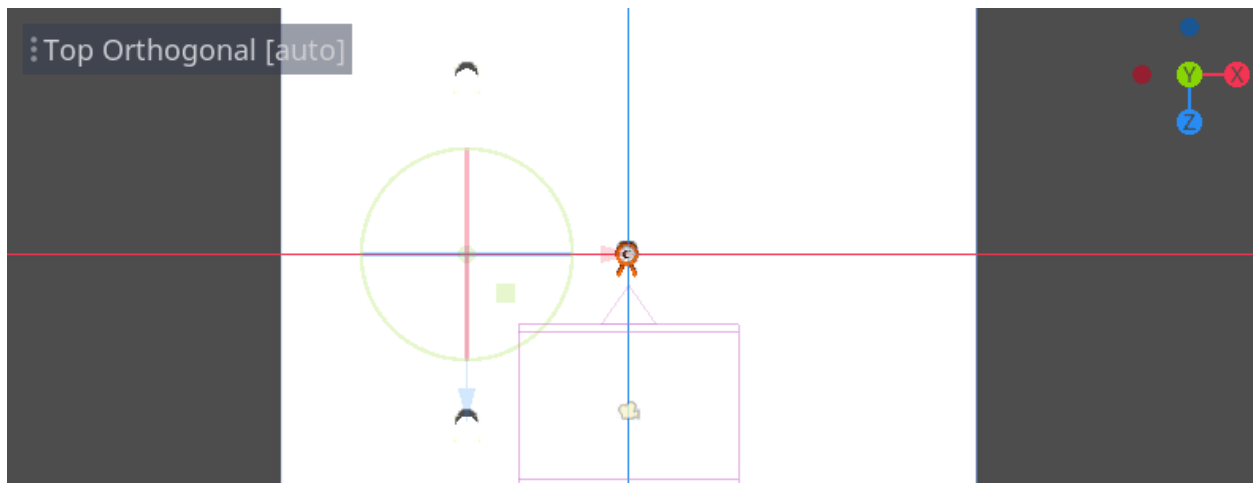


Move the cylinder so it's right outside the camera's view in the top-left corner.

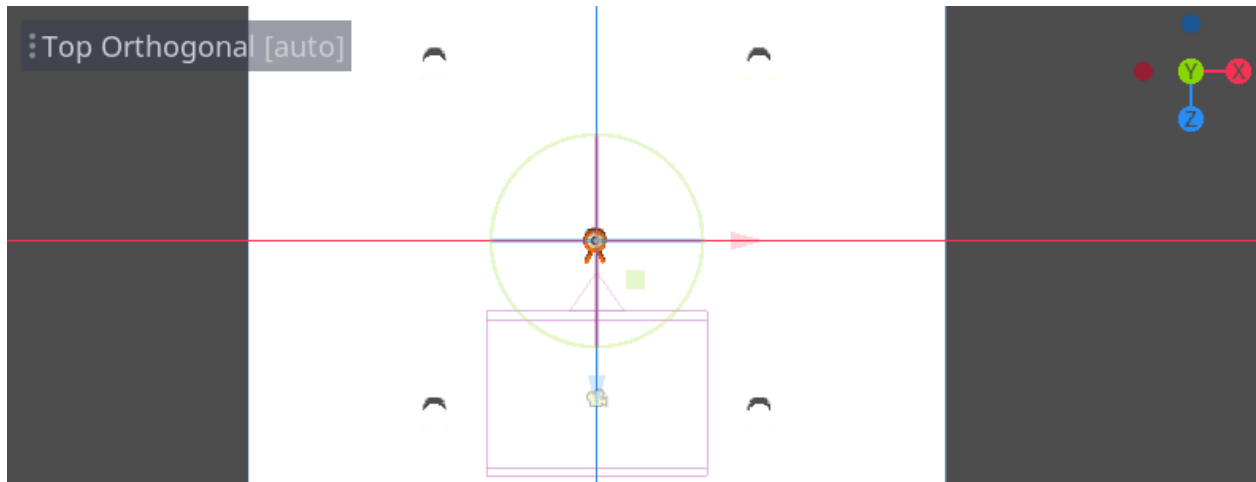


We're going to create copies of the mesh and place them around the game area. Press `Ctrl + D` (`Cmd + D` on macOS) to duplicate the node. You can also right-click the node in the Scene dock and select Duplicate. Move the copy down along the blue Z axis until it's right outside the camera's preview.

Select both cylinders by pressing the Shift key and clicking on the unselected one and duplicate them.



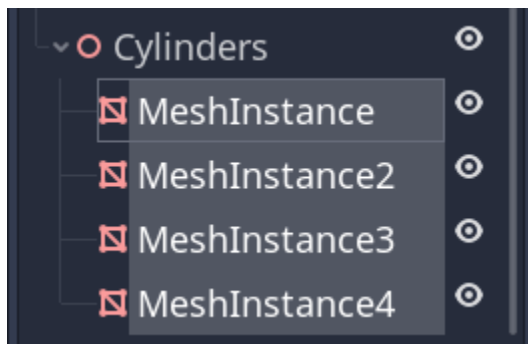
Move them to the right by dragging the red X axis.



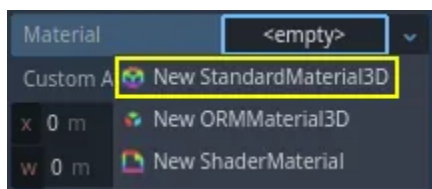
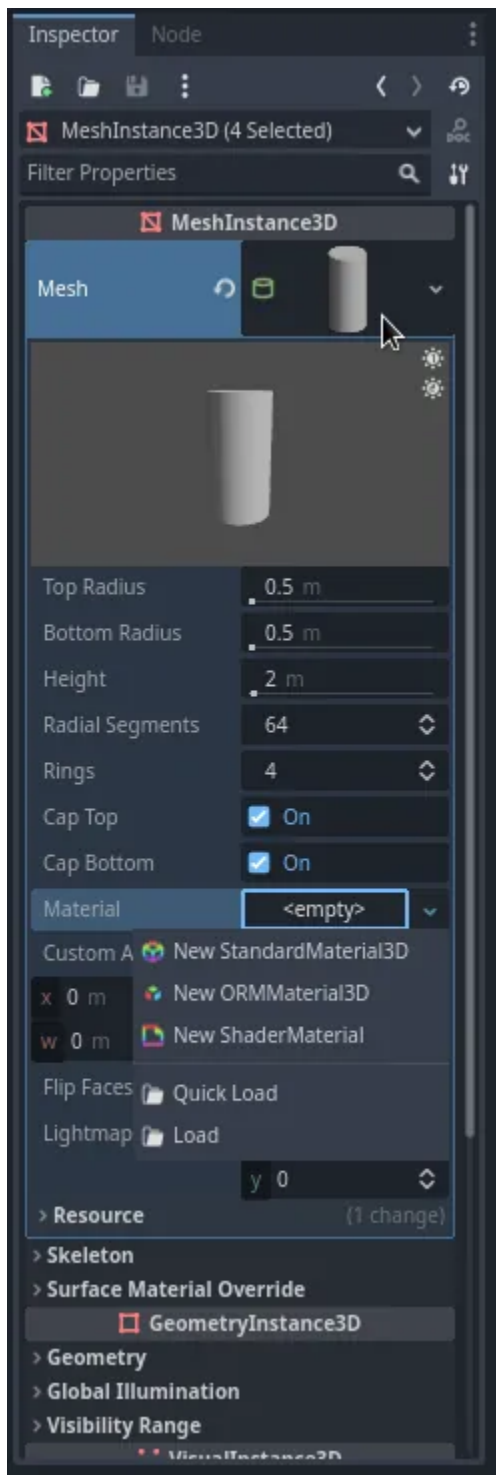
They're a bit hard to see in white, aren't they? Let's make them stand out by giving them a new material.

In 3D, materials define a surface's visual properties like its color, how it reflects light, and more. We can use them to change the color of a mesh.

We can update all four cylinders at once. Select all the mesh instances in the Scene dock. To do so, you can click on the first one and Shift click on the last one.

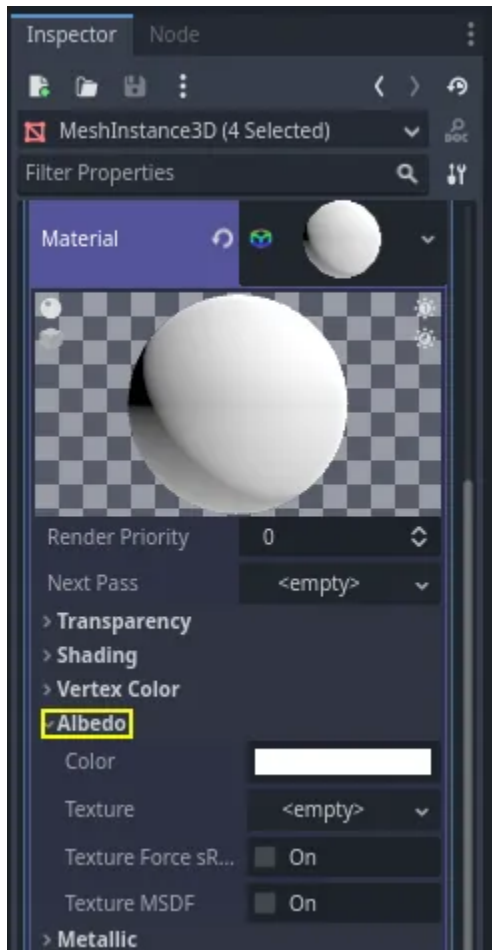


In the Inspector, expand the Material section and assign a `StandardMaterial3D` to slot 0.

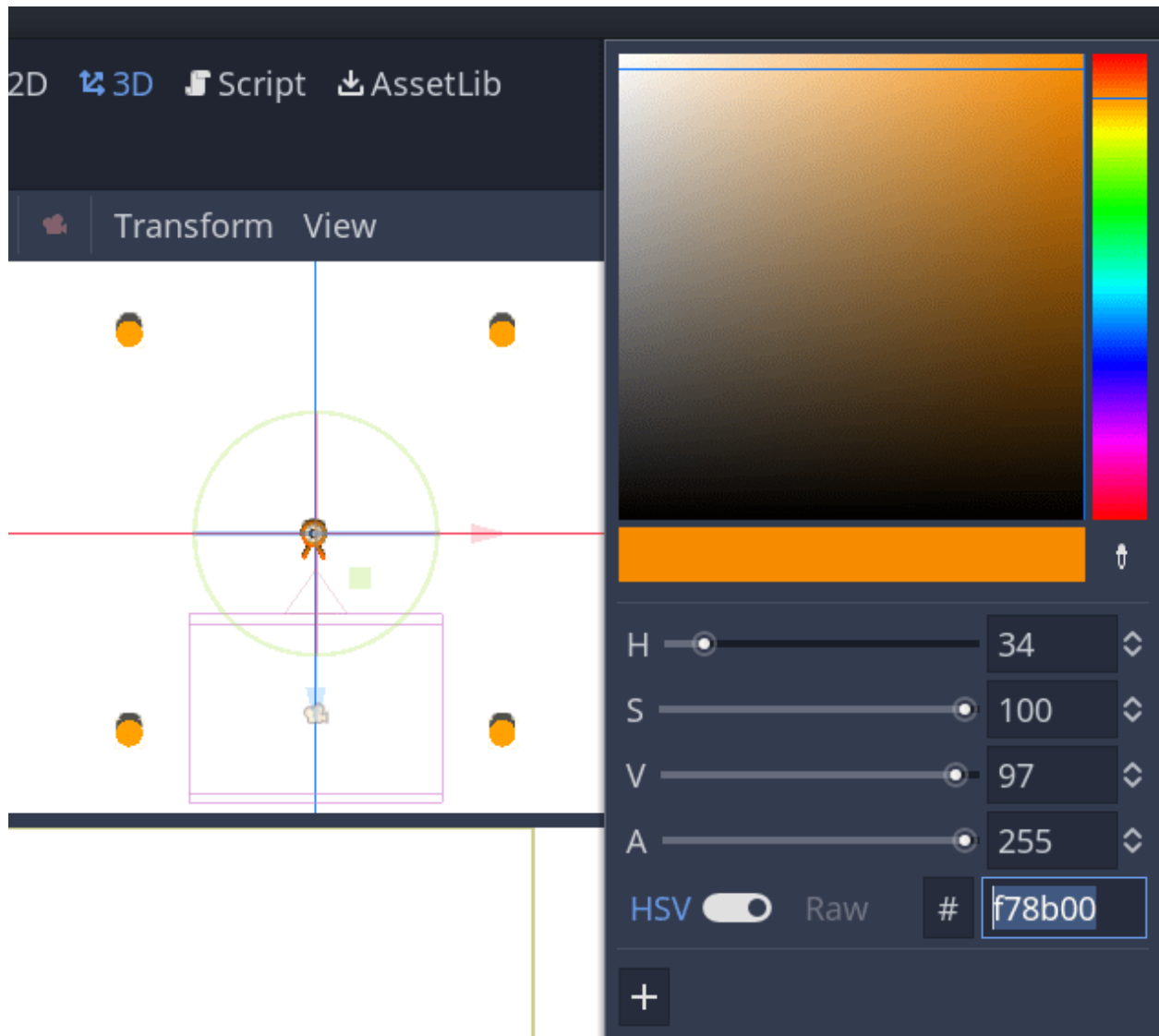


Click the sphere icon to open the material resource. You get a preview of the material and a long list of

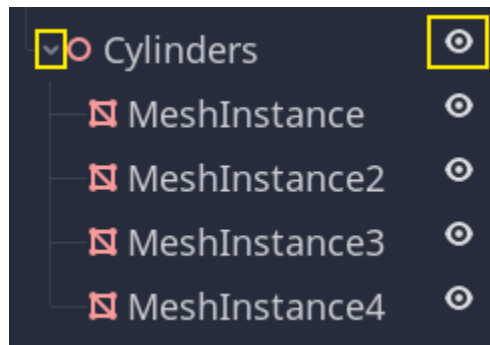
sections filled with properties. You can use these to create all sorts of surfaces, from metal to rock or water. Expand the Albedo section.



Set the color to something that contrasts with the background, like a bright orange.



We can now use the cylinders as guides. Fold them in the Scene dock by clicking the grey arrow next to them. Moving forward, you can also toggle their visibility by clicking the eye icon next to Cylinders.



Add a child node Path3D to Main node. In the toolbar, four icons appear. Click the Add Point tool, the icon with the green "+" sign.

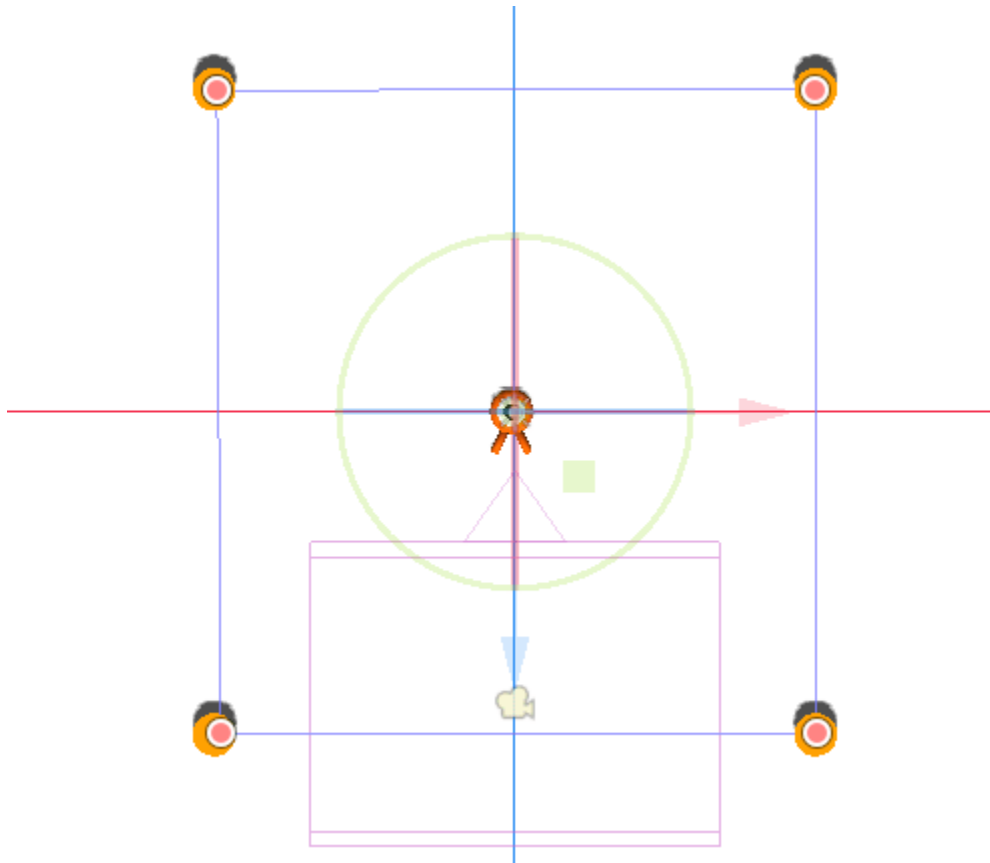


Note: You can hover any icon to see a tooltip describing the tool.

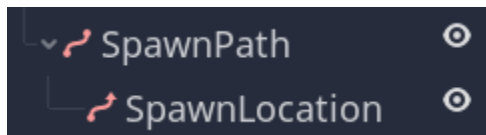
Click in the center of each cylinder to create a point. Then, click the Close Curve icon in the toolbar to close the path. If any point is a bit off, you can click and drag on it to reposition it.



Your path should look like this.



To sample random positions on it, we need a `PathFollow3D` node. Add a `PathFollow3D` as a child of the `Path3D`. Rename the two nodes to `SpawnPath` and `SpawnLocation`, respectively. It's more descriptive of what we'll use them for.



With that, we're ready to code the spawn mechanism.

Spawning monsters randomly

Right-click on the Main node and attach a new script to it.

We first export a variable to the Inspector so that we can assign mob.tscn or any other monster to it.

GDScript

```
extends Node

@export var mob_scene: PackedScene
```

C#

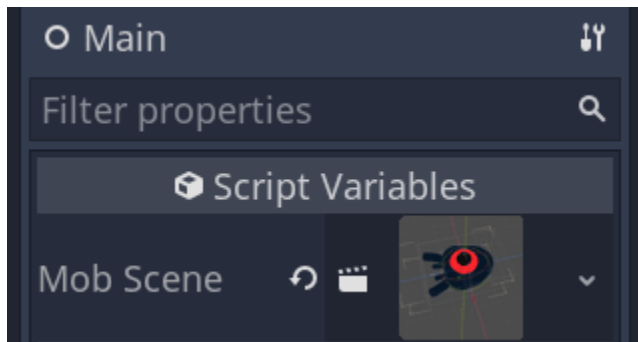
```
using Godot;

public partial class Main : Node
{
    // Don't forget to rebuild the project so the editor knows about the new export variable.

    [Export]
    public PackedScene MobScene { get; set; }
}
```

We want to spawn mobs at regular time intervals. To do this, we need to go back to the scene and add a timer. Before that, though, we need to assign the mob.tscn file to the mob_scene property above (otherwise it's null!)

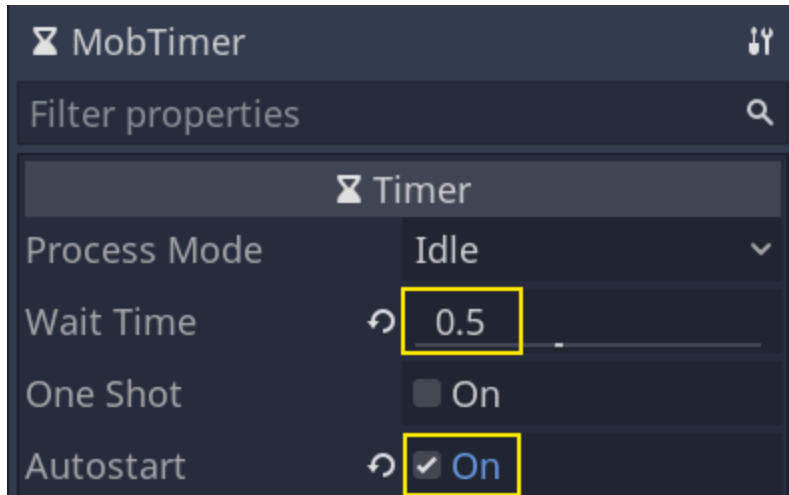
Head back to the 3D screen and select the Main node. Drag mob.tscn from the FileSystem dock to the Mob Scene slot in the Inspector.



Add a new Timer node as a child of Main. Name it MobTimer.

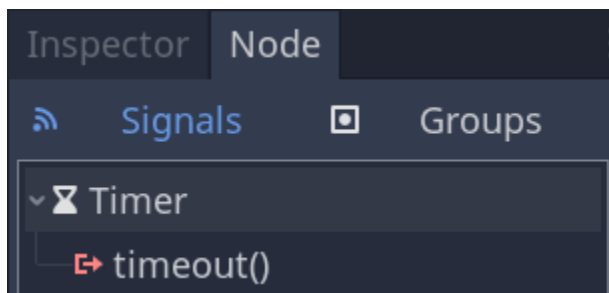


In the Inspector, set its Wait Time to 0.5 seconds and turn on Autostart so it automatically starts when we run the game.

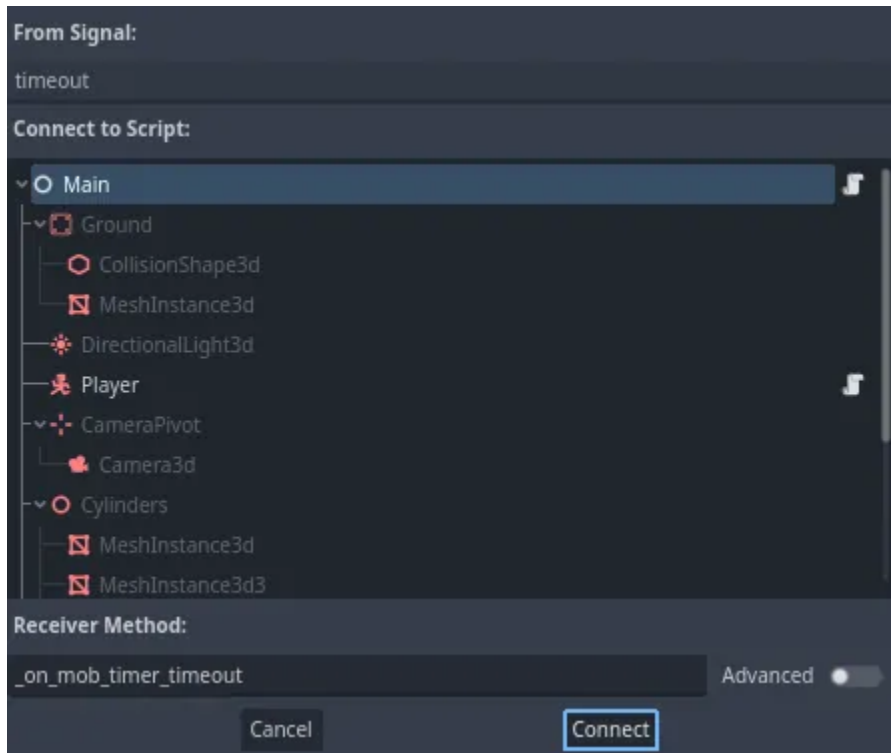


Timers emit a timeout signal every time they reach the end of their Wait Time. By default, they restart automatically, emitting the signal in a cycle. We can connect to this signal from the Main node to spawn monsters every 0.5 seconds.

With the MobTimer still selected, head to the Node dock on the right, and double-click the timeout signal.



Connect it to the Main node.



This will take you back to the script, with a new empty `_on_mob_timer_timeout()` function.

Let's code the mob spawning logic. We're going to:

1. Instantiate the mob scene.
2. Sample a random position on the spawn path.
3. Get the player's position.
4. Call the mob's `initialize()` method, passing it the random position and the player's position.
5. Add the mob as a child of the Main node.

GDScript

```
func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location = get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position, player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)
```

C#

```
// We also specified this function name in PascalCase in the editor's connection window
private void OnMobTimerTimeout()
{
    // Create a new instance of the Mob scene.
    Mob mob = MobScene.Instantiate<Mob>();

    // Choose a random location on the SpawnPath.
    // We store the reference to the SpawnLocation node.
    var mobSpawnLocation = GetNode<PathFollow3D>("SpawnPath/SpawnLocation");
    // And give it a random offset.
    mobSpawnLocation.ProgressRatio = GD.Randf();

    Vector3 playerPosition = GetNode<Player>("Player").Position;
    mob.Initialize(mobSpawnLocation.Position, playerPosition);

    // Spawn the mob by adding it to the Main scene.
    AddChild(mob);
}
```

Above, `randf()` produces a random value between 0 and 1, which is what the `PathFollow` node's `progress_ratio` expects: 0 is the start of the path, 1 is the end of the path. The path we have set is around the camera's viewport, so any random value between 0 and 1 is a random position alongside the edges of the viewport!

Here is the complete `main.gd` script so far, for reference.

GDScript

```
extends Node

@export var mob_scene: PackedScene

func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location = get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position, player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)
```

C#

```
using Godot;

public partial class Main : Node
{
```

(continues on next page)

(continued from previous page)

```
[Export]
public PackedScene MobScene { get; set; }

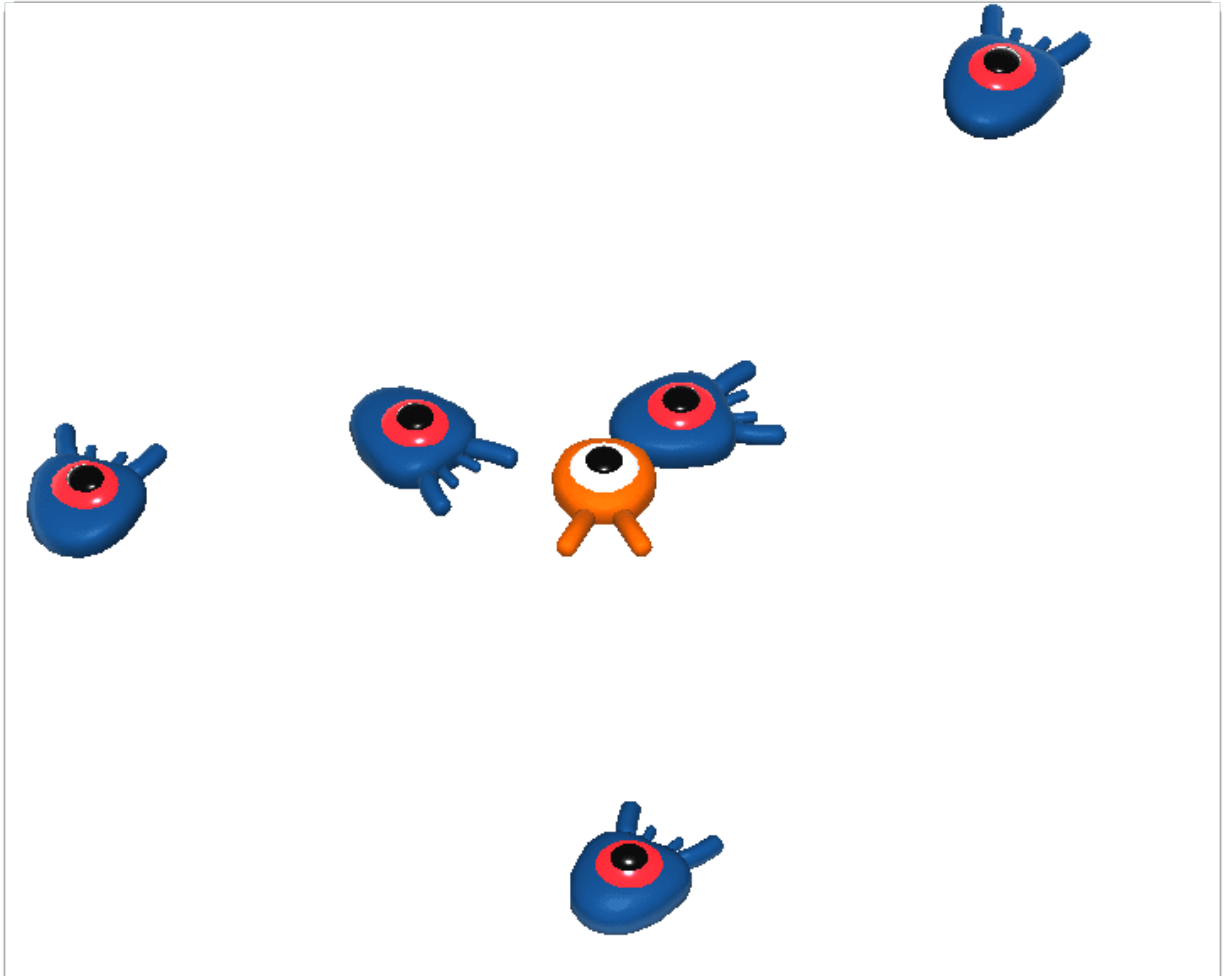
private void OnMobTimerTimeout()
{
    // Create a new instance of the Mob scene.
    Mob mob = MobScene.Instantiate<Mob>();

    // Choose a random location on the SpawnPath.
    // We store the reference to the SpawnLocation node.
    var mobSpawnLocation = GetNode<PathFollow3D>("SpawnPath/SpawnLocation");
    // And give it a random offset.
    mobSpawnLocation.ProgressRatio = GD.Randf();

    Vector3 playerPosition = GetNode<Player>("Player").Position;
    mob.Initialize(mobSpawnLocation.Position, playerPosition);

    // Spawn the mob by adding it to the Main scene.
    AddChild(mob);
}
}
```

You can test the scene by pressing F6. You should see the monsters spawn and move in a straight line.



For now, they bump and slide against one another when their paths cross. We'll address this in the next part.

Jumping and squashing monsters

In this part, we'll add the ability to jump and squash the monsters. In the next lesson, we'll make the player die when a monster hits them on the ground.

First, we have to change a few settings related to physics interactions. Enter the world of physics layers.

Controlling physics interactions

Physics bodies have access to two complementary properties: layers and masks. Layers define on which physics layer(s) an object is.

Masks control the layers that a body will listen to and detect. This affects collision detection. When you want two bodies to interact, you need at least one to have a mask corresponding to the other.

If that's confusing, don't worry, we'll see three examples in a second.

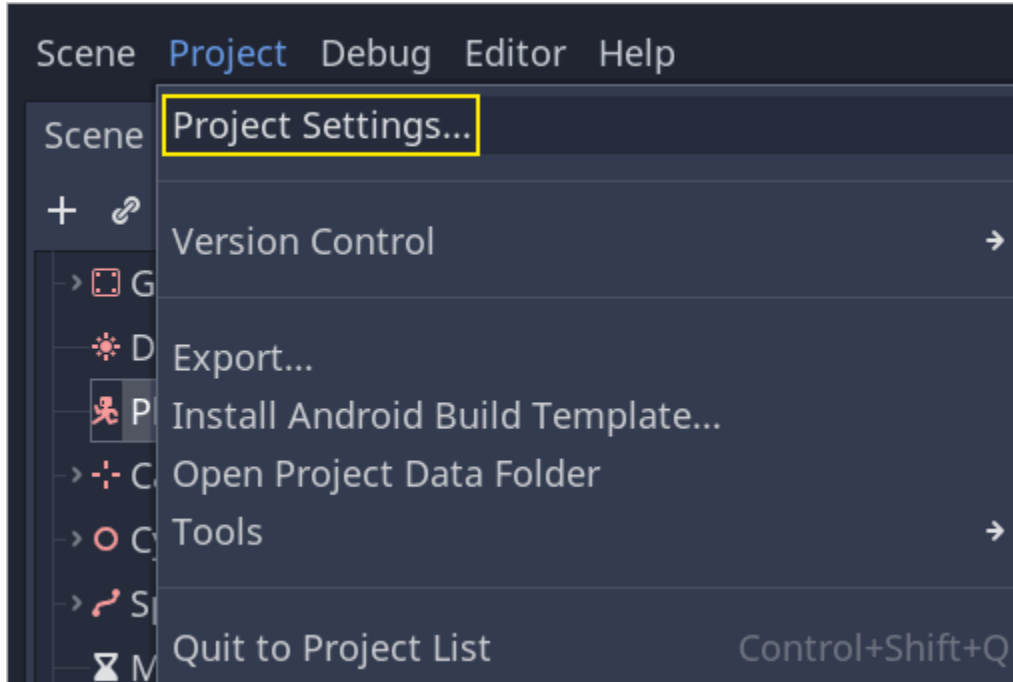
The important point is that you can use layers and masks to filter physics interactions, control performance, and remove the need for extra conditions in your code.

By default, all physics bodies and areas are set to both layer and mask 1. This means they all collide with each other.

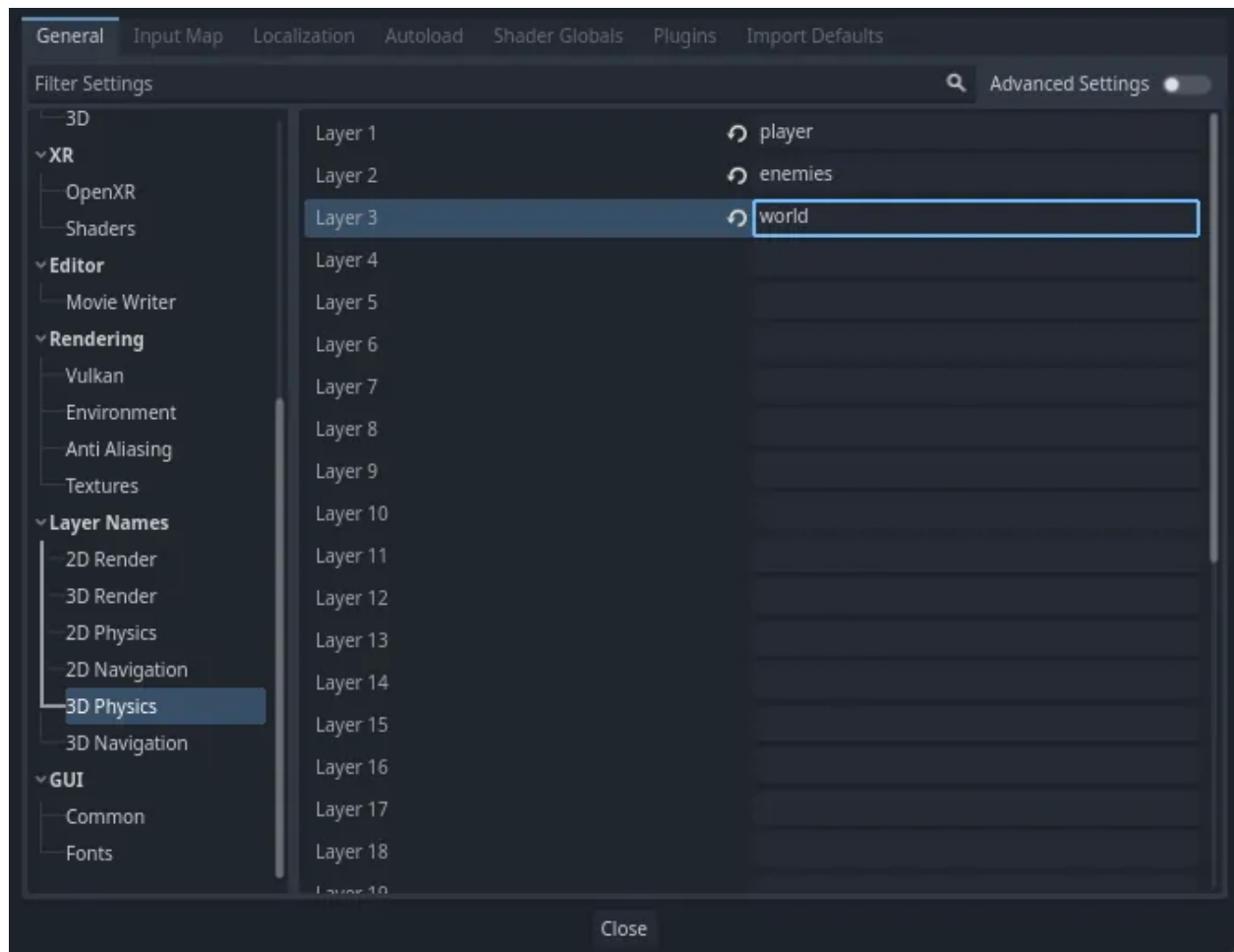
Physics layers are represented by numbers, but we can give them names to keep track of what's what.

Setting layer names

Let's give our physics layers a name. Go to Project -> Project Settings.



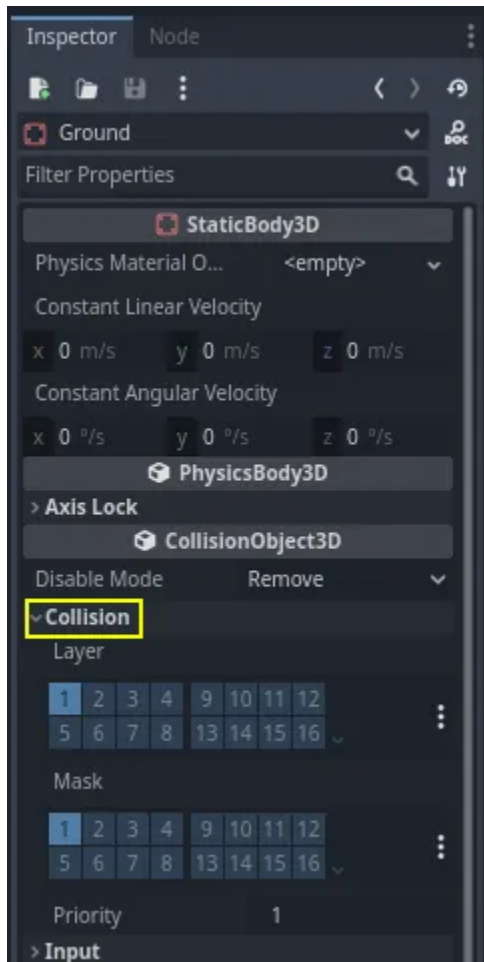
In the left menu, navigate down to Layer Names -> 3D Physics. You can see a list of layers with a field next to each of them on the right. You can set their names there. Name the first three layers player, enemies, and world, respectively.



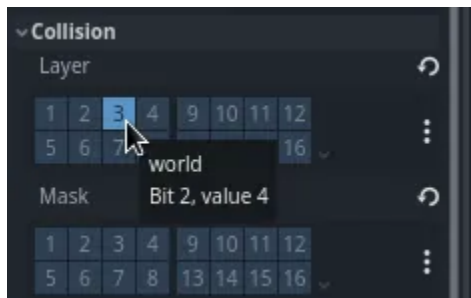
Now, we can assign them to our physics nodes.

Assigning layers and masks

In the Main scene, select the Ground node. In the Inspector, expand the Collision section. There, you can see the node's layers and masks as a grid of buttons.



The ground is part of the world, so we want it to be part of the third layer. Click the lit button to toggle off the first Layer and toggle on the third one. Then, toggle off the Mask by clicking on it.



As mentioned before, the Mask property allows a node to listen to interaction with other physics objects, but we don't need it to have collisions. Ground doesn't need to listen to anything; it's just there to prevent creatures from falling.

Note that you can click the "..." button on the right side of the properties to see a list of named checkboxes.



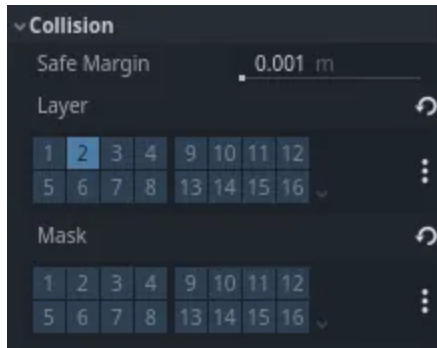
Next up are the Player and the Mob. Open `player.tscn` by double-clicking the file in the FileSystem dock.

Select the Player node and set its Collision -> Mask to both "enemies" and "world". You can leave the default Layer property as it is, because the first layer is the "player" layer.



Then, open the Mob scene by double-clicking on `mob.tscn` and select the Mob node.

Set its Collision -> Layer to "enemies" and unset its Collision -> Mask, leaving the mask empty.



These settings mean the monsters will move through one another. If you want the monsters to collide with and slide against each other, turn on the "enemies" mask.

Note: The mobs don't need to mask the "world" layer because they only move on the XZ plane. We don't apply any gravity to them by design.

Jumping

The jumping mechanic itself requires only two lines of code. Open the Player script. We need a value to control the jump's strength and update `_physics_process()` to code the jump.

After the line that defines `fall_acceleration`, at the top of the script, add the `jump_impulse`.

GDScript

```
#...
# Vertical impulse applied to the character upon jumping in meters per second.
@export var jump_impulse = 20
```

C#

```
// Don't forget to rebuild the project so the editor knows about the new export variable.

// ...
// Vertical impulse applied to the character upon jumping in meters per second.
[Export]
public int JumpImpulse { get; set; } = 20;
```

Inside `_physics_process()`, add the following code before the `move_and_slide()` codeblock.

GDScript

```
func _physics_process(delta):
    #...

    # Jumping.
    if is_on_floor() and Input.is_action_just_pressed("jump"):
        target_velocity.y = jump_impulse

    #...
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...

    // Jumping.
    if (IsOnFloor() && Input.IsActionJustPressed("jump"))
    {
        _targetVelocity.Y = JumpImpulse;
    }

    // ...
}
```

That's all you need to jump!

The `is_on_floor()` method is a tool from the `CharacterBody3D` class. It returns true if the body collided with the floor in this frame. That's why we apply gravity to the Player: so we collide with the floor instead of floating over it like the monsters.

If the character is on the floor and the player presses "jump", we instantly give them a lot of vertical speed. In games, you really want controls to be responsive and giving instant speed boosts like these, while unrealistic, feels great.

Notice that the Y axis is positive upwards. That's unlike 2D, where the Y axis is positive downwards.

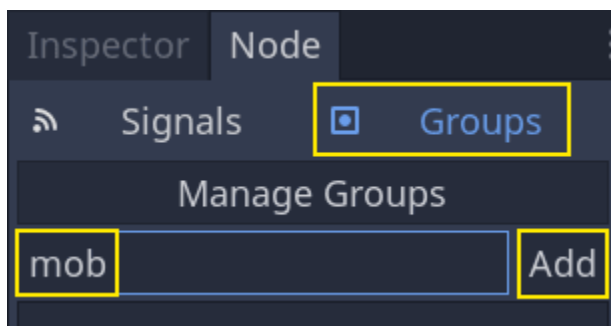
Squashing monsters

Let's add the squash mechanic next. We're going to make the character bounce over monsters and kill them at the same time.

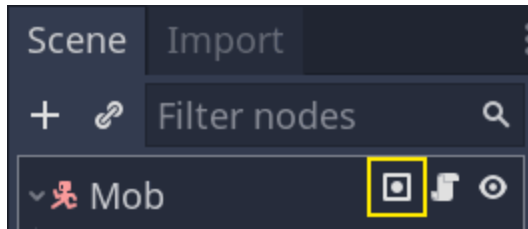
We need to detect collisions with a monster and to differentiate them from collisions with the floor. To do so, we can use Godot's group tagging feature.

Open the scene `mob.tscn` again and select the Mob node. Go to the Node dock on the right to see a list of signals. The Node dock has two tabs: Signals, which you've already used, and Groups, which allows you to assign tags to nodes.

Click on it to reveal a field where you can write a tag name. Enter "mob" in the field and click the Add button.



An icon appears in the Scene dock to indicate the node is part of at least one group.



We can now use the group from the code to distinguish collisions with monsters from collisions with the floor.

Coding the squash mechanic

Head back to the Player script to code the squash and bounce.

At the top of the script, we need another property, `bounce_impulse`. When squashing an enemy, we don't necessarily want the character to go as high up as when jumping.

GDScript

```
# Vertical impulse applied to the character upon bouncing over a mob in
# meters per second.
@export var bounce_impulse = 16
```

C#

```
// Don't forget to rebuild the project so the editor knows about the new export variable.

// Vertical impulse applied to the character upon bouncing over a mob in meters per second.
[Export]
public int BounceImpulse { get; set; } = 16;
```

Then, after the Jumping codeblock we added above in `_physics_process()`, add the following loop. With `move_and_slide()`, Godot makes the body move sometimes multiple times in a row to smooth out the character's motion. So we have to loop over all collisions that may have happened.

In every iteration of the loop, we check if we landed on a mob. If so, we kill it and bounce.

With this code, if no collisions occurred on a given frame, the loop won't run.

GDScript

```
func _physics_process(delta):
    #...

    # Iterate through all collisions that occurred this frame
    for index in range(get_slide_collision_count()):
        # We get one of the collisions with the player
        var collision = get_slide_collision(index)

        # If the collision is with ground
        if collision.get_collider() == null:
            continue

        # If the collider is with a mob
        if collision.get_collider().is_in_group("mob"):
```

(continues on next page)

(continued from previous page)

```
var mob = collision.get_collider()
# we check that we are hitting it from above.
if Vector3.UP.dot(collision.get_normal()) > 0.1:
    # If so, we squash it and bounce.
    mob.squash()
    target_velocity.y = bounce_impulse
    # Prevent further duplicate calls.
    break
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...

    // Iterate through all collisions that occurred this frame.
    for (int index = 0; index < GetSlideCollisionCount(); index++)
    {
        // We get one of the collisions with the player.
        KinematicCollision3D collision = GetSlideCollision(index);

        // If the collision is with a mob.
        // With C# we leverage typing and pattern-matching
        // instead of checking for the group we created.
        if (collision.GetCollider() is Mob mob)
        {
            // We check that we are hitting it from above.
            if (Vector3.Up.Dot(collision.GetNormal()) > 0.1f)
            {
                // If so, we squash it and bounce.
                mob.Squash();
                _targetVelocity.Y = BounceImpulse;
                // Prevent further duplicate calls.
                break;
            }
        }
    }
}
```

That's a lot of new functions. Here's some more information about them.

The functions `get_slide_collision_count()` and `get_slide_collision()` both come from the `CharacterBody3D` class and are related to `move_and_slide()`.

`get_slide_collision()` returns a `KinematicCollision3D` object that holds information about where and how the collision occurred. For example, we use its `get_collider` property to check if we collided with a "mob" by calling `is_in_group()` on it: `collision.get_collider().is_in_group("mob")`.

Note: The method `is_in_group()` is available on every `Node`.

To check that we are landing on the monster, we use the vector dot product: `Vector3.UP.dot(collision.get_normal()) > 0.1`. The collision normal is a 3D vector that is perpendicular to the plane where the collision occurred. The dot product allows us to compare it to the up direction.

With dot products, when the result is greater than 0, the two vectors are at an angle of fewer than 90 degrees. A value higher than 0.1 tells us that we are roughly above the monster.

After handling the squash and bounce logic, we terminate the loop early via the `break` statement to prevent further duplicate calls to `mob.squash()`, which may otherwise result in unintended bugs such as counting the score multiple times for one kill.

We are calling one undefined function, `mob.squash()`, so we have to add it to the `Mob` class.

Open the script `Mob.gd` by double-clicking on it in the `FileSystem` dock. At the top of the script, we want to define a new signal named `squashed`. And at the bottom, you can add the `squash` function, where we emit the signal and destroy the mob.

GDScript

```
# Emitted when the player jumped on the mob.
signal squashed

# ...

func squash():
    squashed.emit()
    queue_free()
```

C#

```
// Don't forget to rebuild the project so the editor knows about the new signal.

// Emitted when the player jumped on the mob.
[Signal]
public delegate void SquashedEventHandler();

// ...

public void Squash()
{
    EmitSignal(SignalName.Squashed);
    QueueFree();
}
```

Note: When using C#, Godot will create the appropriate events automatically for all Signals ending with `EventHandler`, see [C# Signals](#).

We will use the signal to add points to the score in the next lesson.

With that, you should be able to kill monsters by jumping on them. You can press `F5` to try the game and set `main.tscn` as your project's main scene.

However, the player won't die yet. We'll work on that in the next part.

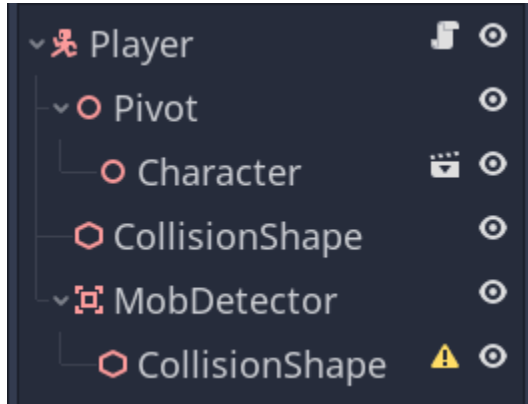
Killing the player

We can kill enemies by jumping on them, but the player still can't die. Let's fix this.

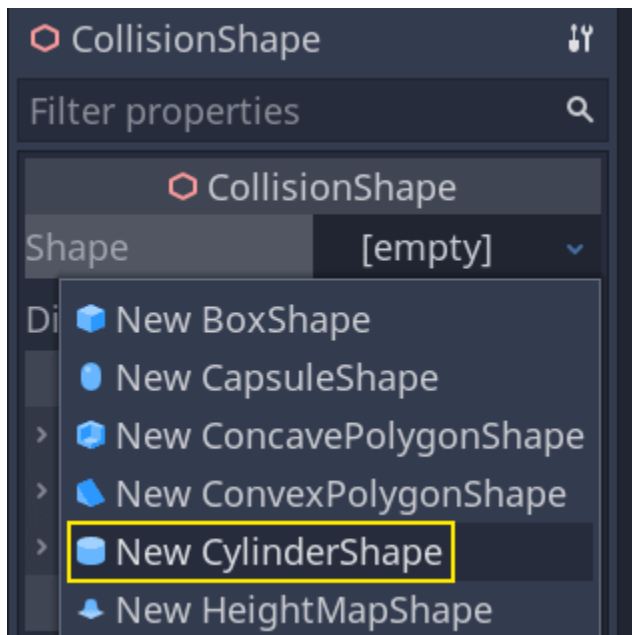
We want to detect being hit by an enemy differently from squashing them. We want the player to die when they're moving on the floor, but not if they're in the air. We could use vector math to distinguish the two kinds of collisions. Instead, though, we will use an Area3D node, which works well for hitboxes.

Hitbox with the Area node

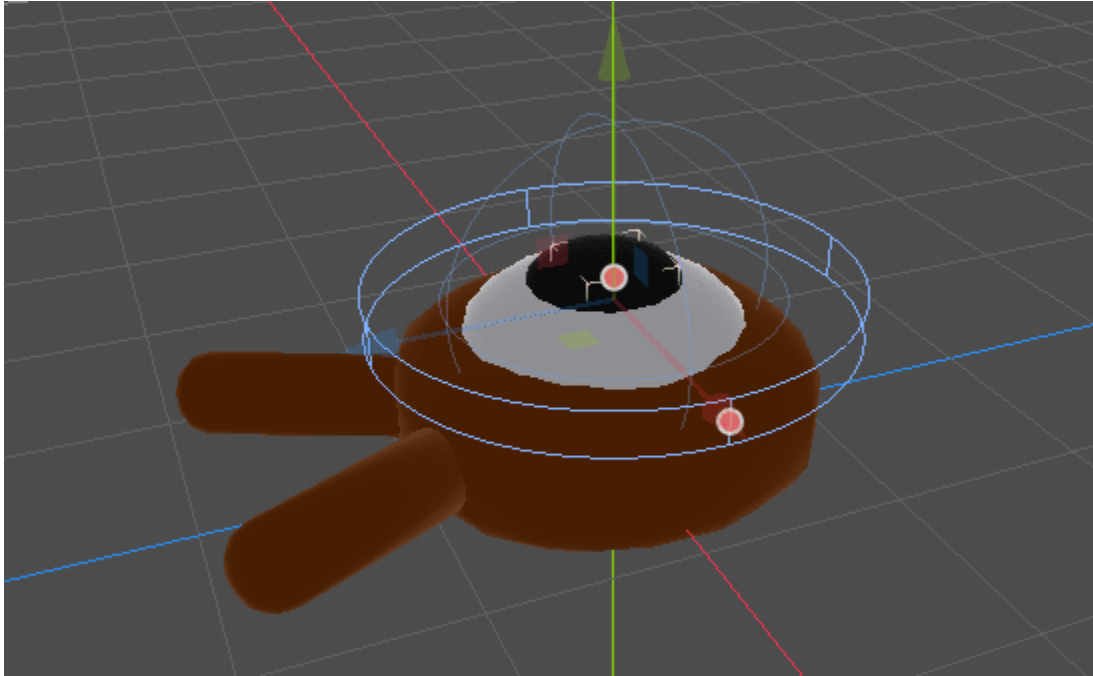
Head back to the player.tscn scene and add a new child node Area3D. Name it MobDetector. Add a CollisionShape3D node as a child of it.



In the Inspector, assign a cylinder shape to it.



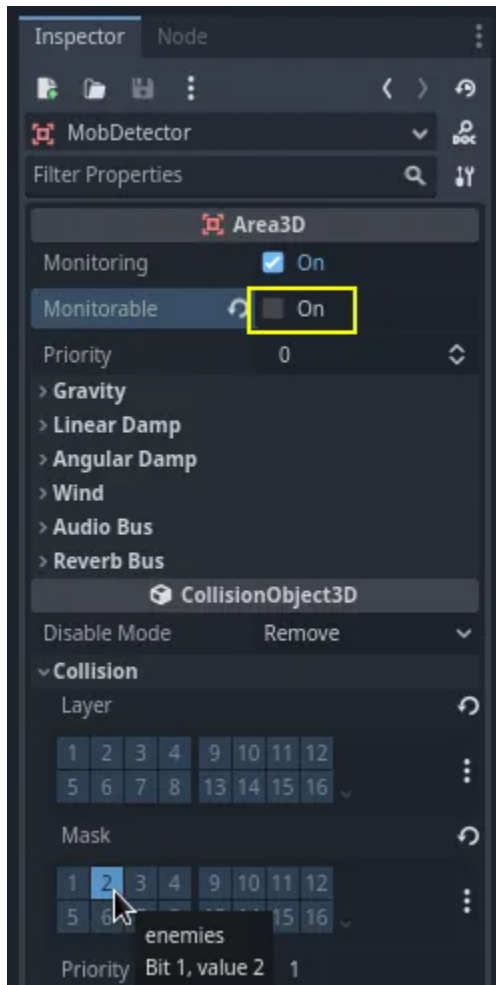
Here is a trick you can use to make the collisions only happen when the player is on the ground or close to it. You can reduce the cylinder's height and move it up to the top of the character. This way, when the player jumps, the shape will be too high up for the enemies to collide with it.



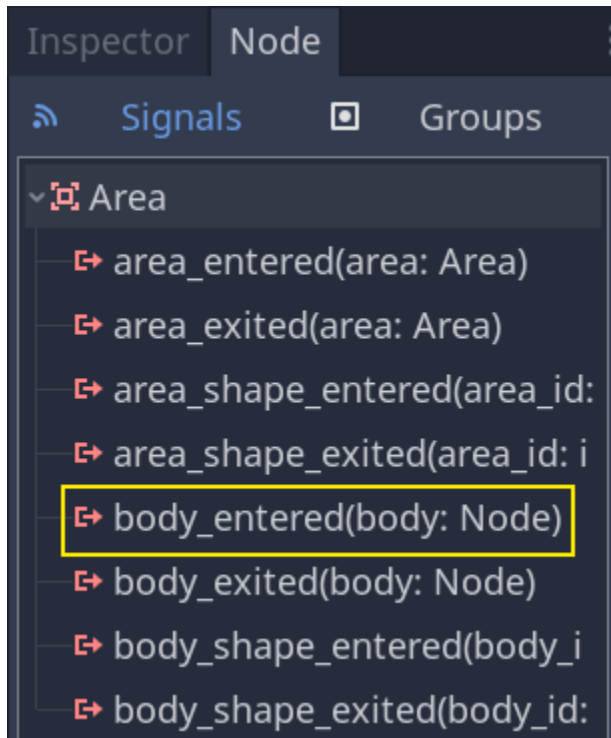
You also want the cylinder to be wider than the sphere. This way, the player gets hit before colliding and being pushed on top of the monster's collision box.

The wider the cylinder, the more easily the player will get killed.

Next, select the MobDetector node again, and in the Inspector, turn off its Monitorable property. This makes it so other physics nodes cannot detect the area. The complementary Monitoring property allows it to detect collisions. Then, remove the Collision -> Layer and set the mask to the "enemies" layer.



When areas detect a collision, they emit signals. We're going to connect one to the Player node. Select MobDetector and go to Inspector's Node tab, double-click the `body_entered` signal and connect it to the Player



The MobDetector will emit `body_entered` when a `CharacterBody3D` or a `RigidBody3D` node enters it. As it only masks the "enemies" physics layers, it will only detect the Mob nodes.

Code-wise, we're going to do two things: emit a signal we'll later use to end the game and destroy the player. We can wrap these operations in a `die()` function that helps us put a descriptive label on the code.

GDScript

```
# Emitted when the player was hit by a mob.
# Put this at the top of the script.
signal hit

# And this function at the bottom.
func die():
    hit.emit()
    queue_free()

func _on_mob_detector_body_entered(body):
    die()
```

C#

```
// Don't forget to rebuild the project so the editor knows about the new signal.

// Emitted when the player was hit by a mob.
[Signal]
public delegate void HitEventHandler();

// ...
```

(continues on next page)

(continued from previous page)

```
private void Die()
{
    EmitSignal(SignalName.Hit);
    QueueFree();
}

// We also specified this function name in PascalCase in the editor's connection window
private void OnMobDetectorBodyEntered(Node3D body)
{
    Die();
}
```

Try the game again by pressing F5. If everything is set up correctly, the character should die when an enemy runs into the collider. Note that without a Player, the following line

GDScript

```
var player_position = $Player.position
```

C#

```
Vector3 playerPosition = GetNode<Player>("Player").Position;
```

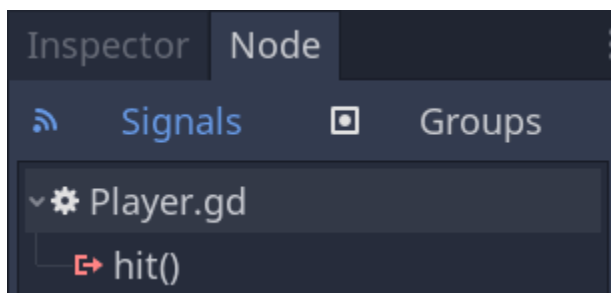
gives error because there is no \$Player!

Also note that the enemy colliding with the player and dying depends on the size and position of the Player and the Mob's collision shapes. You may need to move them and resize them to achieve a tight game feel.

Ending the game

We can use the Player's hit signal to end the game. All we need to do is connect it to the Main node and stop the MobTimer in reaction.

Open main.tscn, select the Player node, and in the Node dock, connect its hit signal to the Main node.



Get the timer, and stop it, in the `_on_player_hit()` function.

GDScript

```
func _on_player_hit():
    $MobTimer.stop()
```

C#

```
// We also specified this function name in PascalCase in the editor's connection window
private void OnPlayerHit()
{
    GetNode<Timer>("MobTimer").Stop();
}
```

If you try the game now, the monsters will stop spawning when you die, and the remaining ones will leave the screen.

You can pat yourself in the back: you prototyped a complete 3D game, even if it's still a bit rough.

From there, we'll add a score, the option to retry the game, and you'll see how you can make the game feel much more alive with minimalistic animations.

Code checkpoint

Here are the complete scripts for the Main, Mob, and Player nodes, for reference. You can use them to compare and check your code.

Starting with main.gd.

GDScript

```
extends Node

@export var mob_scene: PackedScene

func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location = get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position, player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)

func _on_player_hit():
    $MobTimer.stop()
```

C#

```
using Godot;

public partial class Main : Node
{
    [Export]
    public PackedScene MobScene { get; set; }
```

(continues on next page)

(continued from previous page)

```

private void OnMobTimerTimeout()
{
    // Create a new instance of the Mob scene.
    Mob mob = MobScene.Instantiate<Mob>();

    // Choose a random location on the SpawnPath.
    // We store the reference to the SpawnLocation node.
    var mobSpawnLocation = GetNode<PathFollow3D>("SpawnPath/SpawnLocation");
    // And give it a random offset.
    mobSpawnLocation.ProgressRatio = GD.Randf();

    Vector3 playerPosition = GetNode<Player>("Player").Position;
    mob.Initialize(mobSpawnLocation.Position, playerPosition);

    // Spawn the mob by adding it to the Main scene.
    AddChild(mob);
}

private void OnPlayerHit()
{
    GetNode<Timer>("MobTimer").Stop();
}
}

```

Next is Mob.gd.

GDScript

```

extends CharacterBody3D

# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18

# Emitted when the player jumped on the mob
signal squashed

func _physics_process(_delta):
    move_and_slide()

# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the player.
    look_at_from_position(start_position, player_position, Vector3.UP)
    # Rotate this mob randomly within range of -90 and +90 degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))

    # We calculate a random speed (integer)

```

(continues on next page)

(continued from previous page)

```

var random_speed = randi_range(min_speed, max_speed)
# We calculate a forward velocity that represents the speed.
velocity = Vector3.FORWARD * random_speed
# We then rotate the velocity vector based on the mob's Y rotation
# in order to move in the direction the mob is looking.
velocity = velocity.rotated(Vector3.UP, rotation.y)

func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()

func squash():
    squashed.emit()
    queue_free() # Destroy this node

```

C#

```

using Godot;

public partial class Mob : CharacterBody3D
{
    // Emitted when the player jumped on the mob.
    [Signal]
    public delegate void SquashedEventHandler();

    // Minimum speed of the mob in meters per second
    [Export]
    public int MinSpeed { get; set; } = 10;
    // Maximum speed of the mob in meters per second
    [Export]
    public int MaxSpeed { get; set; } = 18;

    public override void _PhysicsProcess(double delta)
    {
        MoveAndSlide();
    }

    // This function will be called from the Main scene.
    public void Initialize(Vector3 startPosition, Vector3 playerPosition)
    {
        // We position the mob by placing it at startPosition
        // and rotate it towards playerPosition, so it looks at the player.
        LookAtFromPosition(startPosition, playerPosition, Vector3.Up);
        // Rotate this mob randomly within range of -90 and +90 degrees,
        // so that it doesn't move directly towards the player.
        RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi / 4.0));

        // We calculate a random speed (integer)
        int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
        // We calculate a forward velocity that represents the speed.
        Velocity = Vector3.Forward * randomSpeed;
        // We then rotate the velocity vector based on the mob's Y rotation
        // in order to move in the direction the mob is looking.
    }
}

```

(continues on next page)

(continued from previous page)

```

    Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);
}

public void Squash()
{
    EmitSignal(SignalName.Squashed);
    QueueFree(); // Destroy this node
}

private void OnVisibilityNotifierScreenExited()
{
    QueueFree();
}
}

```

Finally, the longest script, Player.gd:

GDScript

```

extends CharacterBody3D

signal hit

# How fast the player moves in meters per second
@export var speed = 14
# The downward acceleration while in the air, in meters per second squared.
@export var fall_acceleration = 75
# Vertical impulse applied to the character upon jumping in meters per second.
@export var jump_impulse = 20
# Vertical impulse applied to the character upon bouncing over a mob
# in meters per second.
@export var bounce_impulse = 16

var target_velocity = Vector3.ZERO

func _physics_process(delta):
    # We create a local variable to store the input direction
    var direction = Vector3.ZERO

    # We check for each move input and update the direction accordingly
    if Input.is_action_pressed("move_right"):
        direction.x = direction.x + 1
    if Input.is_action_pressed("move_left"):
        direction.x = direction.x - 1
    if Input.is_action_pressed("move_back"):
        # Notice how we are working with the vector's x and z axes.
        # In 3D, the XZ plane is the ground plane.
        direction.z = direction.z + 1
    if Input.is_action_pressed("move_forward"):
        direction.z = direction.z - 1

```

(continues on next page)

(continued from previous page)

```

# Prevent diagonal moving fast af
if direction != Vector3.ZERO:
    direction = direction.normalized()
    $Pivot.look_at(position + direction, Vector3.UP)

# Ground Velocity
target_velocity.x = direction.x * speed
target_velocity.z = direction.z * speed

# Vertical Velocity
if not is_on_floor(): # If in the air, fall towards the floor. Literally gravity
    target_velocity.y = target_velocity.y - (fall_acceleration * delta)

# Jumping.
if is_on_floor() and Input.is_action_just_pressed("jump"):
    target_velocity.y = jump_impulse

# Iterate through all collisions that occurred this frame
# in C this would be for(int i = 0; i < collisions.Count; i++)
for index in range(get_slide_collision_count()):
    # We get one of the collisions with the player
    var collision = get_slide_collision(index)

    # If the collision is with ground
    if collision.get_collider() == null:
        continue

    # If the collider is with a mob
    if collision.get_collider().is_in_group("mob"):
        var mob = collision.get_collider()
        # we check that we are hitting it from above.
        if Vector3.UP.dot(collision.get_normal()) > 0.1:
            # If so, we squash it and bounce.
            mob.squash()
            target_velocity.y = bounce_impulse
            # Prevent further duplicate calls.
            break

# Moving the Character
velocity = target_velocity
move_and_slide()

# And this function at the bottom.
func die():
    hit.emit()
    queue_free()

func _on_mob_detector_body_entered(body):
    die()

```

C#

```
using Godot;

public partial class Player : CharacterBody3D
{
    // Emitted when the player was hit by a mob.
    [Signal]
    public delegate void HitEventHandler();

    // How fast the player moves in meters per second.
    [Export]
    public int Speed { get; set; } = 14;
    // The downward acceleration when in the air, in meters per second squared.
    [Export]
    public int FallAcceleration { get; set; } = 75;
    // Vertical impulse applied to the character upon jumping in meters per second.
    [Export]
    public int JumpImpulse { get; set; } = 20;
    // Vertical impulse applied to the character upon bouncing over a mob in meters per second.
    [Export]
    public int BounceImpulse { get; set; } = 16;

    private Vector3 _targetVelocity = Vector3.Zero;

    public override void _PhysicsProcess(double delta)
    {
        // We create a local variable to store the input direction.
        var direction = Vector3.Zero;

        // We check for each move input and update the direction accordingly.
        if (Input.IsActionPressed("move_right"))
        {
            direction.X += 1.0f;
        }
        if (Input.IsActionPressed("move_left"))
        {
            direction.X -= 1.0f;
        }
        if (Input.IsActionPressed("move_back"))
        {
            // Notice how we are working with the vector's X and Z axes.
            // In 3D, the XZ plane is the ground plane.
            direction.Z += 1.0f;
        }
        if (Input.IsActionPressed("move_forward"))
        {
            direction.Z -= 1.0f;
        }

        // Prevent diagonal moving fast af
        if (direction != Vector3.Zero)
        {
            direction = direction.Normalized();
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    GetNode<Node3D>("Pivot").LookAt(Position + direction, Vector3.Up);
}

// Ground Velocity
_targetVelocity.X = direction.X * Speed;
_targetVelocity.Z = direction.Z * Speed;

// Vertical Velocity
if (!IsOnFloor()) // If in the air, fall towards the floor. Literally gravity
{
    _targetVelocity.Y -= FallAcceleration * (float)delta;
}

// Jumping.
if (IsOnFloor() && Input.IsActionJustPressed("jump"))
{
    _targetVelocity.Y = JumpImpulse;
}

// Iterate through all collisions that occurred this frame.
for (int index = 0; index < GetSlideCollisionCount(); index++)
{
    // We get one of the collisions with the player.
    KinematicCollision3D collision = GetSlideCollision(index);

    // If the collision is with a mob.
    if (collision.GetCollider() is Mob mob)
    {
        // We check that we are hitting it from above.
        if (Vector3.Up.Dot(collision.GetNormal()) > 0.1f)
        {
            // If so, we squash it and bounce.
            mob.Squash();
            _targetVelocity.Y = BounceImpulse;
            // Prevent further duplicate calls.
            break;
        }
    }
}

// Moving the Character
Velocity = _targetVelocity;
MoveAndSlide();
}

private void Die()
{
    EmitSignal(SignalName.Hit);
    QueueFree();
}

private void OnMobDetectorBodyEntered(Node3D body)

```

(continues on next page)

(continued from previous page)

```
{  
    Die();  
}  
}
```

See you in the next lesson to add the score and the retry option.

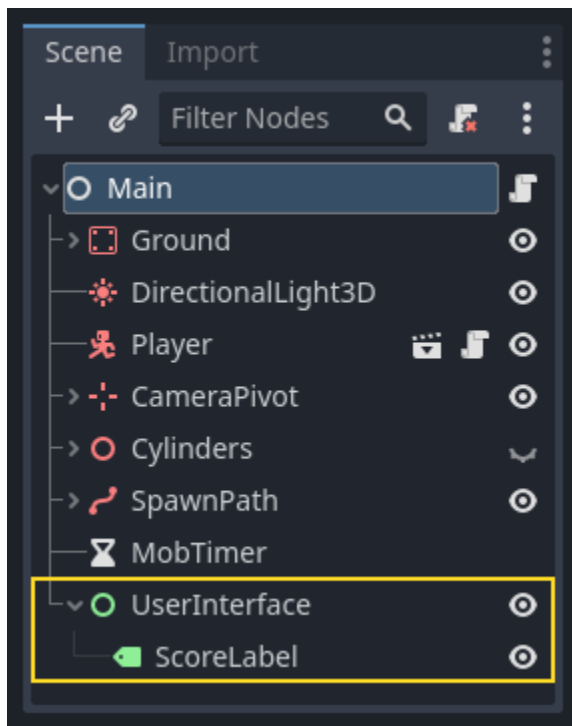
Score and replay

In this part, we'll add the score, music playback, and the ability to restart the game.

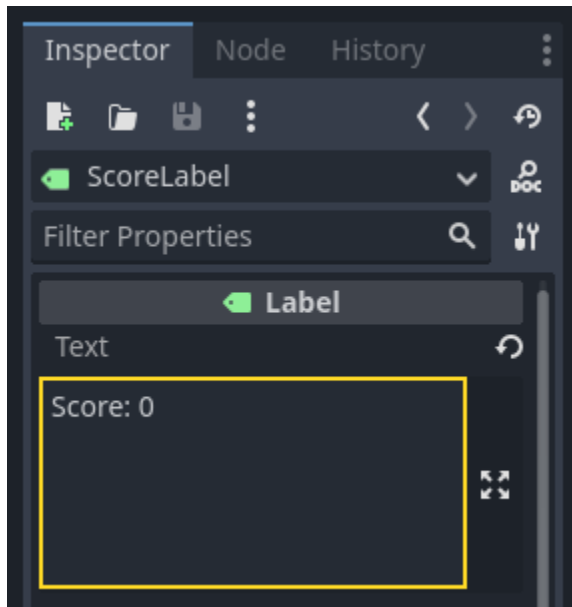
We have to keep track of the current score in a variable and display it on screen using a minimal interface. We will use a text label to do that.

In the main scene, add a new child node Control to Main and name it `UserInterface`. You will automatically be taken to the 2D screen, where you can edit your User Interface (UI).

Add a Label node and name it `ScoreLabel`

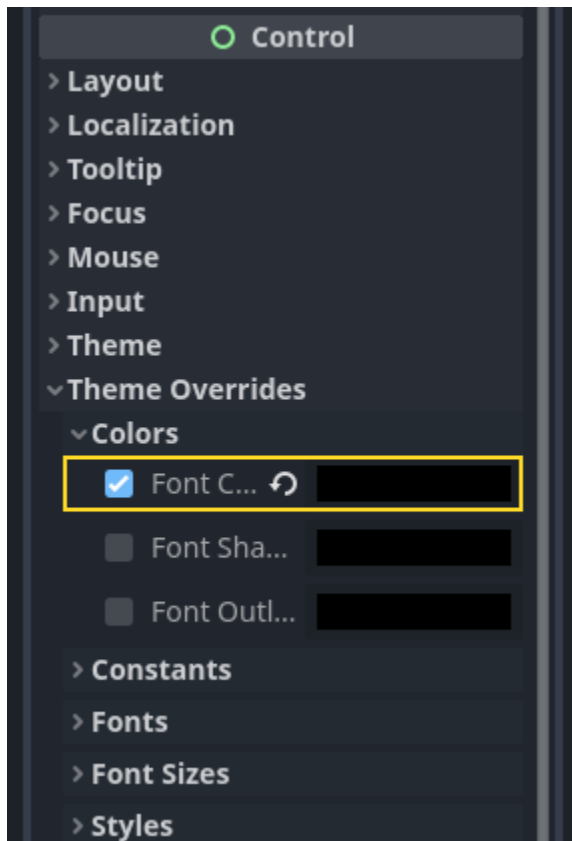


In the Inspector, set the Label's Text to a placeholder like "Score: 0".

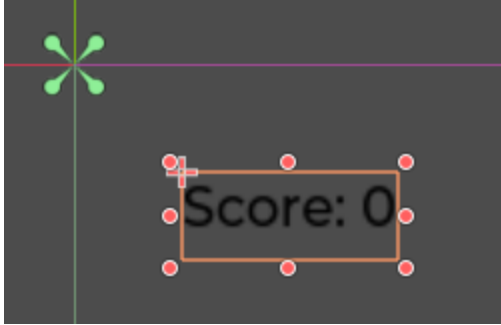


Also, the text is white by default, like our game's background. We need to change its color to see it at runtime.

Scroll down to Theme Overrides, and expand Colors and enable Font Color in order to tint the text to black (which contrasts well with the white 3D scene)



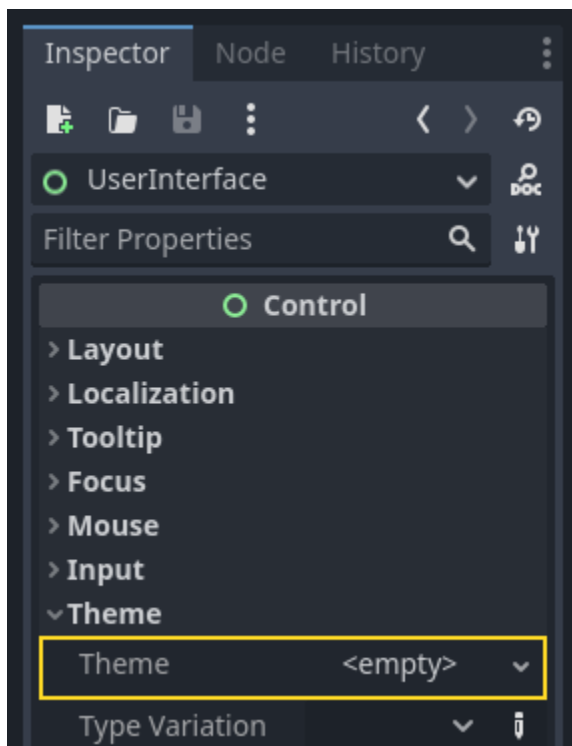
Finally, click and drag on the text in the viewport to move it away from the top-left corner.



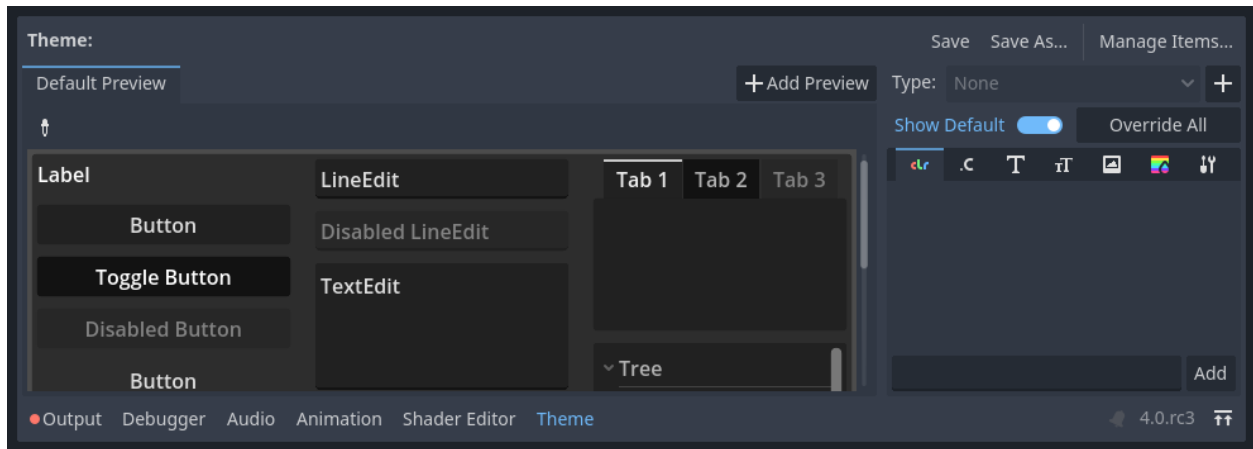
The `UIInterface` node allows us to group our UI in a branch of the scene tree and use a theme resource that will propagate to all its children. We'll use it to set our game's font.

Creating a UI theme

Once again, select the `UIInterface` node. In the Inspector, create a new theme resource in Theme -> Theme.



Click on it to open the theme editor In the bottom panel. It gives you a preview of how all the built-in UI widgets will look with your theme resource.



By default, a theme only has one property, the Default Font.

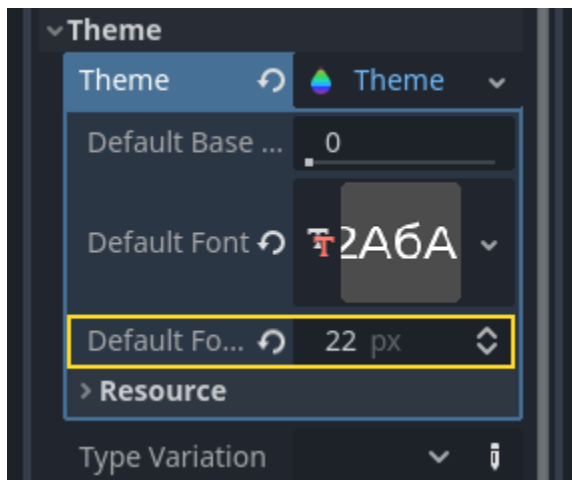
See also:

You can add more properties to the theme resource to design complex user interfaces, but that is beyond the scope of this series. To learn more about creating and editing themes, see [Introduction to GUI skinning](#).

This one expects a font file like the ones you have on your computer. Two common font file formats are TrueType Font (TTF) and OpenType Font (OTF).

In the FileSystem dock, expand the fonts directory and click and drag the Montserrat-Medium.ttf file we included in the project onto the Default Font. The text will reappear in the theme preview.

The text is a bit small. Set the Default Font Size to 22 pixels to increase the text's size.



Keeping track of the score

Let's work on the score next. Attach a new script to the ScoreLabel and define the score variable.

GDScript

```
extends Label

var score = 0
```

C#

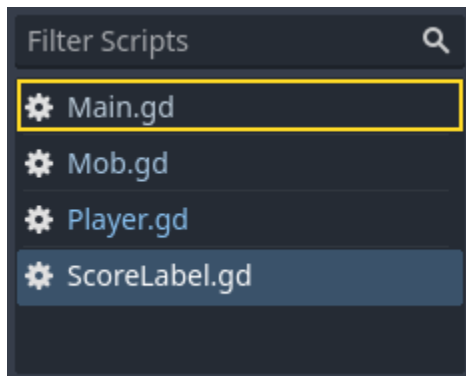
```
using Godot;

public partial class ScoreLabel : Label
{
    private int _score = 0;
}
```

The score should increase by 1 every time we squash a monster. We can use their squashed signal to know when that happens. However, because we instantiate monsters from the code, we cannot connect the mob signal to the ScoreLabel via the editor.

Instead, we have to make the connection from the code every time we spawn a monster.

Open the script main.gd. If it's still open, you can click on its name in the script editor's left column.



Alternatively, you can double-click the main.gd file in the FileSystem dock.

At the bottom of the `_on_mob_timer_timeout()` function, add the following line:

GDScript

```
func _on_mob_timer_timeout():
    #...
    # We connect the mob to the score label to update the score upon squashing one.
    mob.squashed.connect($UserInterface/ScoreLabel._on_mob_squashed.bind())
```

C#

```
private void OnMobTimerTimeout()
{
    // ...
    // We connect the mob to the score label to update the score upon squashing one.
    mob.Squashed += GetNode<ScoreLabel>("UserInterface/ScoreLabel").OnMobSquashed;
}
```

This line means that when the mob emits the squashed signal, the ScoreLabel node will receive it and call the function `_on_mob_squashed()`.

Head back to the ScoreLabel.gd script to define the `_on_mob_squashed()` callback function.

There, we increment the score and update the displayed text.

GDScript

```
func _on_mob_squashed():  
    score += 1  
    text = "Score: %s" % score
```

C#

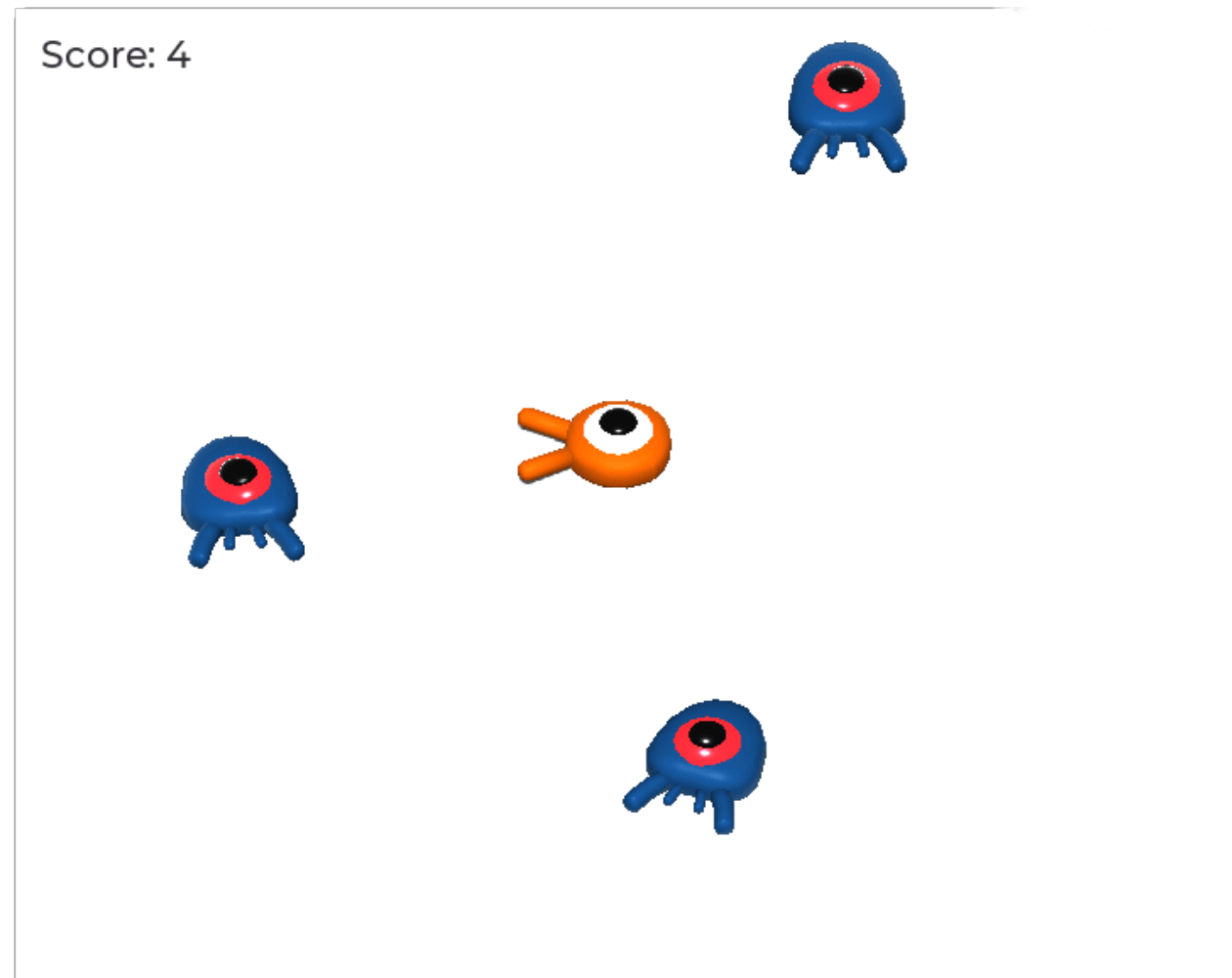
```
public void OnMobSquashed()  
{  
    _score += 1;  
    Text = $"Score: {_score}";  
}
```

The second line uses the value of the score variable to replace the placeholder `%s`. When using this feature, Godot automatically converts values to string text, which is convenient when outputting text in labels or when using the `print()` function.

See also:

You can learn more about string formatting here: [GDScript format strings](#). In C#, consider using [string interpolation](#) with `"$"`.

You can now play the game and squash a few enemies to see the score increase.



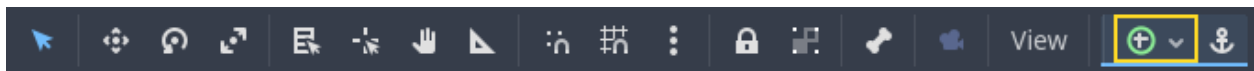
Note: In a complex game, you may want to completely separate your user interface from the game world. In that case, you would not keep track of the score on the label. Instead, you may want to store it in a separate, dedicated object. But when prototyping or when your project is simple, it is fine to keep your code simple. Programming is always a balancing act.

Retrying the game

We'll now add the ability to play again after dying. When the player dies, we'll display a message on the screen and wait for input.

Head back to the main.tscn scene, select the `UIInterface` node, add a child node `ColorRect`, and name it `Retry`. This node fills a rectangle with a uniform color and will serve as an overlay to darken the screen.

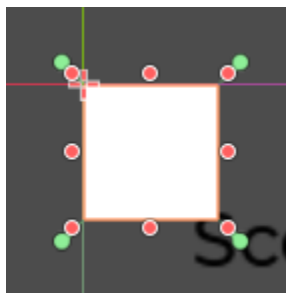
To make it span over the whole viewport, you can use the `Anchor Preset` menu in the toolbar.



Open it and apply the `Full Rect` command.



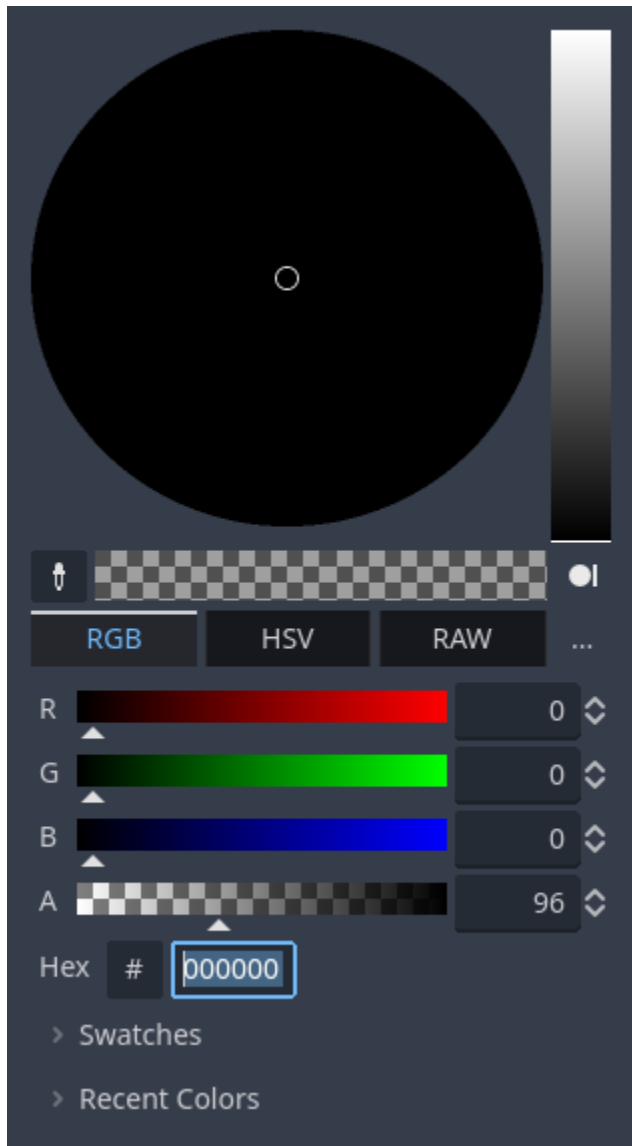
Nothing happens. Well, almost nothing; only the four green pins move to the corners of the selection box.



This is because `UI` nodes (all the ones with a green icon) work with anchors and margins relative to their parent's bounding box. Here, the `UIInterface` node has a small size and the `Retry` one is limited by it.

Select the `UIInterface` and apply `Anchor Preset -> Full Rect` to it as well. The `Retry` node should now span the whole viewport.

Let's change its color so it darkens the game area. Select `Retry` and in the `Inspector`, set its `Color` to something both dark and transparent. To do so, in the color picker, drag the `A` slider to the left. It controls the color's `Alpha` channel, that is to say, its opacity/transparency.



Next, add a Label as a child of Retry and give it the Text "Press Enter to retry." To move it and anchor it in the center of the screen, apply Anchor Preset -> Center to it.



Coding the retry option

We can now head to the code to show and hide the Retry node when the player dies and plays again.

Open the script main.gd. First, we want to hide the overlay at the start of the game. Add this line to the `_ready()` function.

GDScript

```
func _ready():
    $UserInterface/Retry.hide()
```

C#

```
public override void _Ready()
{
    GetNode<Control>("UserInterface/Retry").Hide();
}
```

Then, when the player gets hit, we show the overlay.

GDScript

```
func _on_player_hit():
    #...
    $UserInterface/Retry.show()
```

C#

```
private void OnPlayerHit()
{
    //...
    GetNode<Control>("UserInterface/Retry").Show();
}
```

Finally, when the Retry node is visible, we need to listen to the player's input and restart the game if they press enter. To do this, we use the built-in `_unhandled_input()` callback, which is triggered on any input.

If the player pressed the predefined `ui_accept` input action and Retry is visible, we reload the current scene.

GDScript

```
func _unhandled_input(event):
    if event.is_action_pressed("ui_accept") and $UserInterface/Retry.visible:
        # This restarts the current scene.
        get_tree().reload_current_scene()
```

C#

```
public override void _UnhandledInput(InputEvent @event)
{
    if (@event.IsActionPressed("ui_accept") && GetNode<Control>("UserInterface/Retry").Visible)
    {
        // This restarts the current scene.
        GetTree().ReloadCurrentScene();
    }
}
```

The function `get_tree()` gives us access to the global `SceneTree` object, which allows us to reload and restart the current scene.

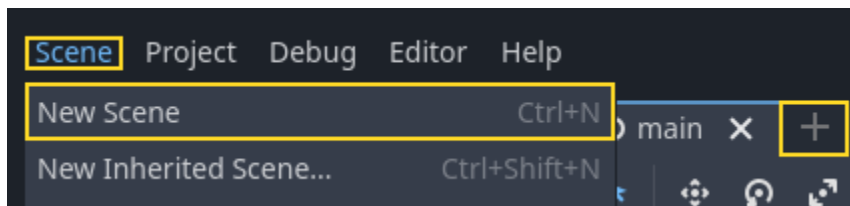
Adding music

To add music that plays continuously in the background, we're going to use another feature in Godot: autoloads.

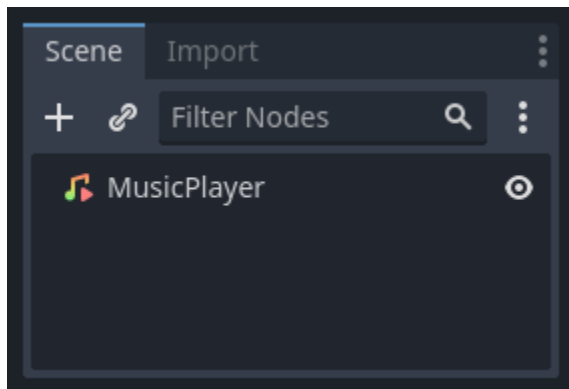
To play audio, all you need to do is add an `AudioStreamPlayer` node to your scene and attach an audio file to it. When you start the scene, it can play automatically. However, when you reload the scene, like we do to play again, the audio nodes are also reset, and the music starts back from the beginning.

You can use the autoload feature to have Godot load a node or a scene automatically at the start of the game, outside the current scene. You can also use it to create globally accessible objects.

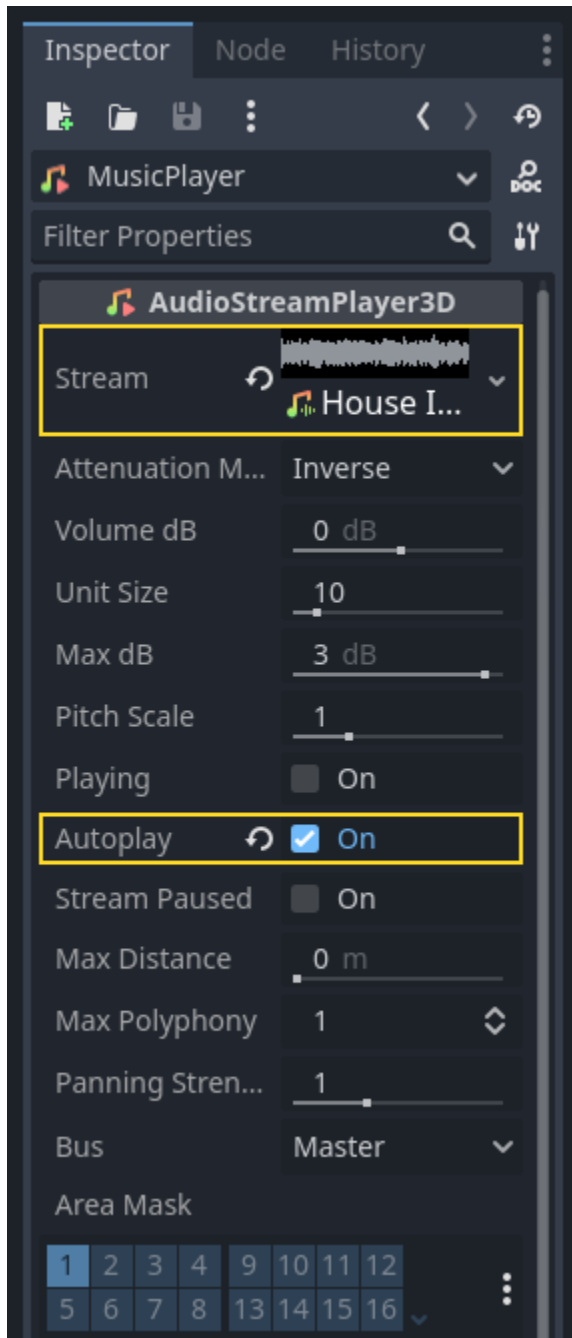
Create a new scene by going to the Scene menu and clicking New Scene or by using the `+` icon next to your currently opened scene.



Click the Other Node button to create an `AudioStreamPlayer` and rename it to `MusicPlayer`.



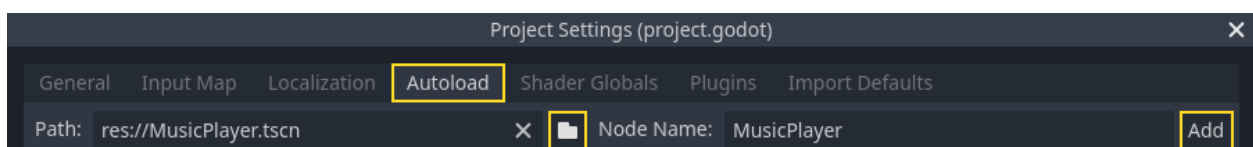
We included a music soundtrack in the `art/` directory, `House In a Forest Loop.ogg`. Click and drag it onto the `Stream` property in the Inspector. Also, turn on `Autoplay` so the music plays automatically at the start of the game.



Save the scene as `MusicPlayer.tscn`.

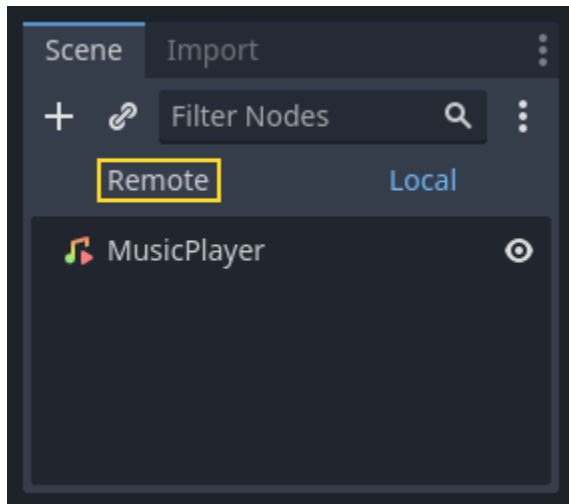
We have to register it as an autoload. Head to the Project -> Project Settings... menu and click on the Autoload tab.

In the Path field, you want to enter the path to your scene. Click the folder icon to open the file browser and double-click on `MusicPlayer.tscn`. Then, click the Add button on the right to register the node.

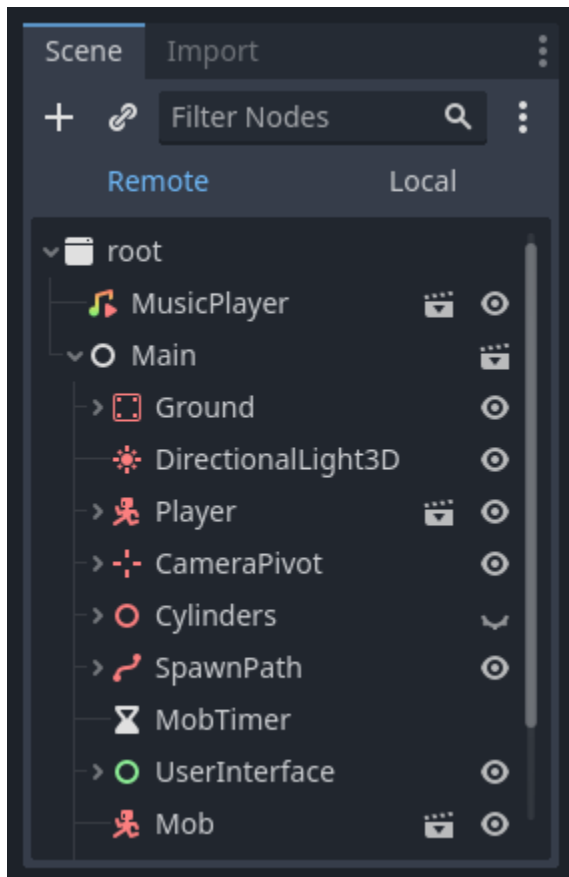


MusicPlayer.tscn now loads into any scene you open or play. So if you run the game now, the music will play automatically in any scene.

Before we wrap up this lesson, here's a quick look at how it works under the hood. When you run the game, your Scene dock changes to give you two tabs: Remote and Local.



The Remote tab allows you to visualize the node tree of your running game. There, you will see the Main node and everything the scene contains and the instantiated mobs at the bottom.



At the top are the autoloaded MusicPlayer and a root node, which is your game's viewport.

And that does it for this lesson. In the next part, we'll add an animation to make the game both look and feel much nicer.

Here is the complete main.gd script for reference.

GDScript

```
extends Node

@export var mob_scene: PackedScene

func _ready():
    $UserInterface/Retry.hide()

func _on_mob_timer_timeout():
    # Create a new instance of the Mob scene.
    var mob = mob_scene.instantiate()

    # Choose a random location on the SpawnPath.
    # We store the reference to the SpawnLocation node.
    var mob_spawn_location = get_node("SpawnPath/SpawnLocation")
    # And give it a random offset.
    mob_spawn_location.progress_ratio = randf()

    var player_position = $Player.position
    mob.initialize(mob_spawn_location.position, player_position)

    # Spawn the mob by adding it to the Main scene.
    add_child(mob)

    # We connect the mob to the score label to update the score upon squashing one.
    mob.squashed.connect($UserInterface/ScoreLabel._on_mob_squashed.bind())

func _on_player_hit():
    $MobTimer.stop()
    $UserInterface/Retry.show()

func _unhandled_input(event):
    if event.is_action_pressed("ui_accept") and $UserInterface/Retry.visible:
        # This restarts the current scene.
        get_tree().reload_current_scene()
```

C#

```
using Godot;

public partial class Main : Node
{
    [Export]
    public PackedScene MobScene { get; set; }

    public override void _Ready()
    {
        GetNode<Control>("UserInterface/Retry").Hide();
```

(continues on next page)

(continued from previous page)

```

}

public override void _UnhandledInput(InputEvent @event)
{
    if (@event.IsActionPressed("ui_accept") && GetNode<Control>("UserInterface/Retry").Visible)
    {
        // This restarts the current scene.
        GetTree().ReloadCurrentScene();
    }
}

private void OnMobTimerTimeout()
{
    // Create a new instance of the Mob scene.
    Mob mob = MobScene.Instantiate<Mob>();

    // Choose a random location on the SpawnPath.
    // We store the reference to the SpawnLocation node.
    var mobSpawnLocation = GetNode<PathFollow3D>("SpawnPath/SpawnLocation");
    // And give it a random offset.
    mobSpawnLocation.ProgressRatio = GD.Randf();

    Vector3 playerPosition = GetNode<Player>("Player").position;
    mob.Initialize(mobSpawnLocation.Position, playerPosition);

    // Spawn the mob by adding it to the Main scene.
    AddChild(mob);

    // We connect the mob to the score label to update the score upon squashing one.
    mob.Squashed += GetNode<ScoreLabel>("UserInterface/ScoreLabel").OnMobSquashed;
}

private void OnPlayerHit()
{
    GetNode<Timer>("MobTimer").Stop();
    GetNode<Control>("UserInterface/Retry").Show();
}
}

```

Character animation

In this final lesson, we'll use Godot's built-in animation tools to make our characters float and flap. You'll learn to design animations in the editor and use code to make your game feel alive.

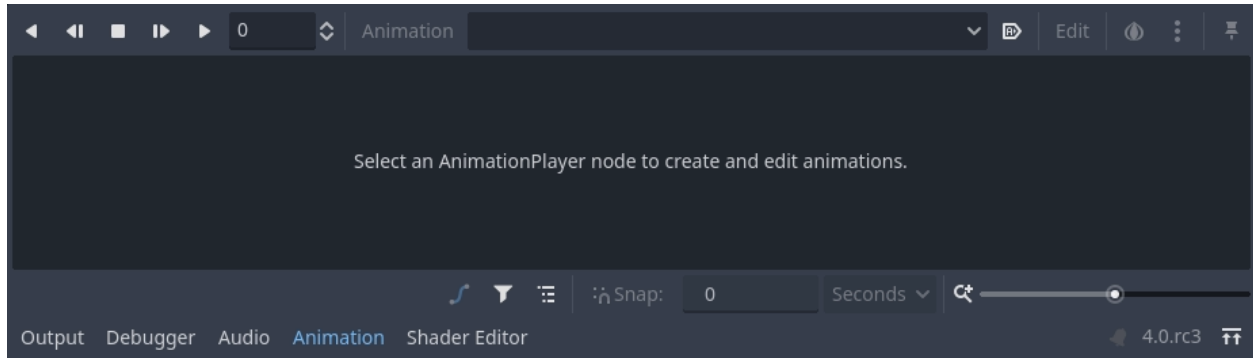
We'll start with an introduction to using the animation editor.

Using the animation editor

The engine comes with tools to author animations in the editor. You can then use the code to play and control them at runtime.

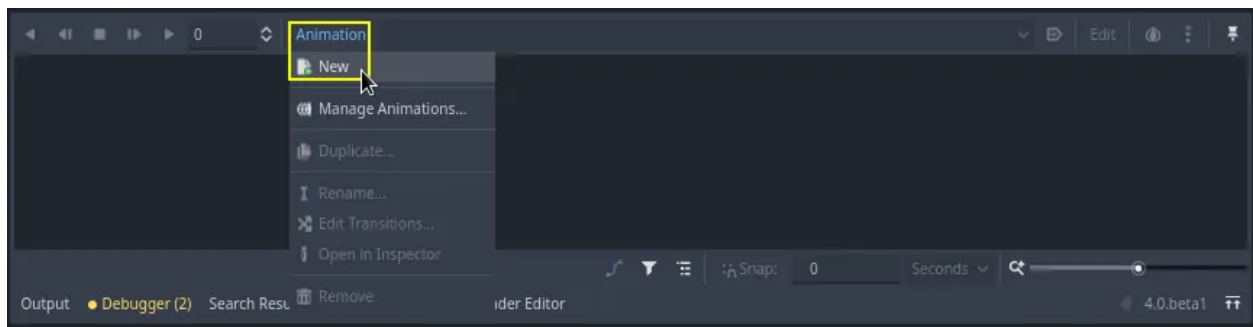
Open the player scene, select the Player node, and add an `AnimationPlayer` node.

The Animation dock appears in the bottom panel.

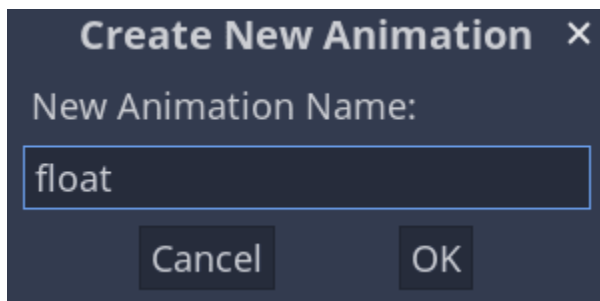


It features a toolbar and the animation drop-down menu at the top, a track editor in the middle that's currently empty, and filter, snap, and zoom options at the bottom.

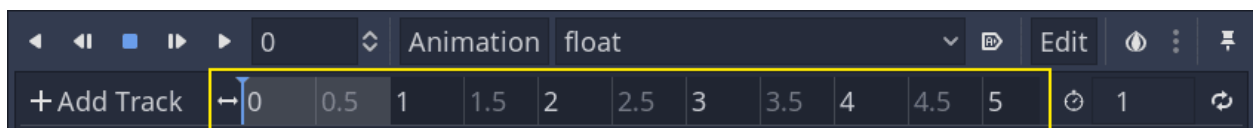
Let's create an animation. Click on Animation -> New.



Name the animation "float".

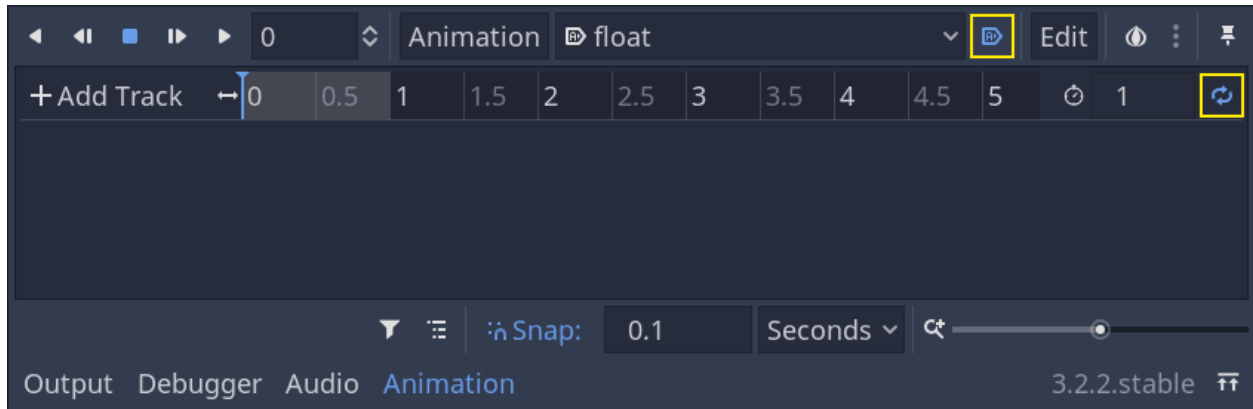


Once you've created the animation, the timeline appears with numbers representing time in seconds.

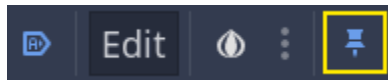


We want the animation to start playback automatically at the start of the game. Also, it should loop.

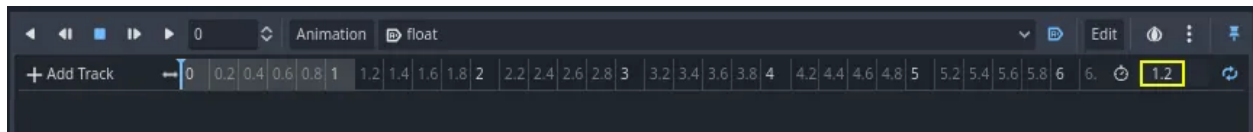
To do so, you can click the button with an "A+" icon in the animation toolbar and the looping arrows, respectively.



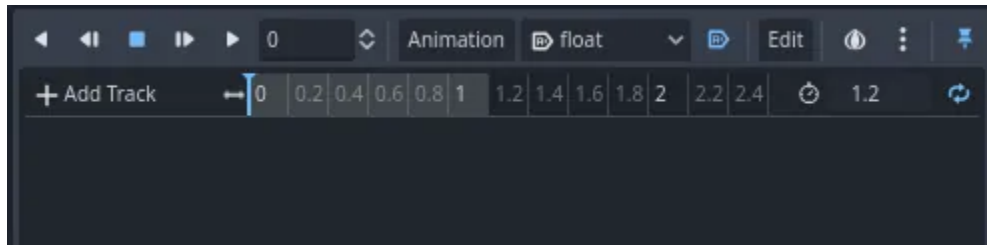
You can also pin the animation editor by clicking the pin icon in the top-right. This prevents it from folding when you click on the viewport and deselect the nodes.



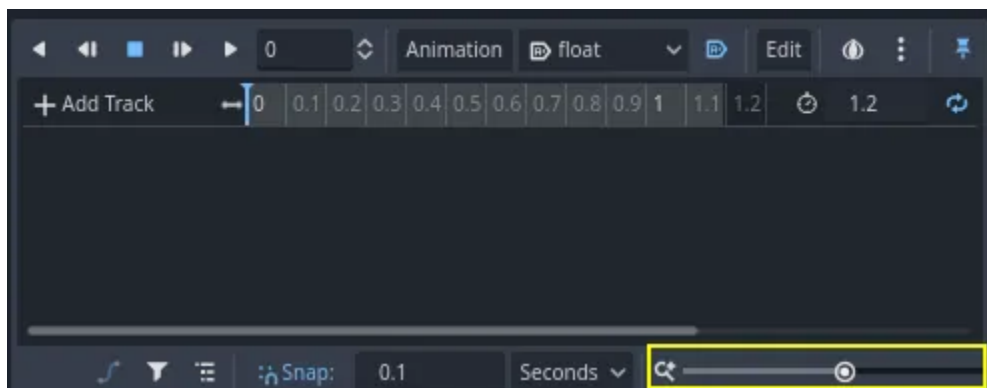
Set the animation duration to 1.2 seconds in the top-right of the dock.



You should see the gray ribbon widen a bit. It shows you the start and end of your animation and the vertical blue line is your time cursor.



You can click and drag the slider in the bottom-right to zoom in and out of the timeline.

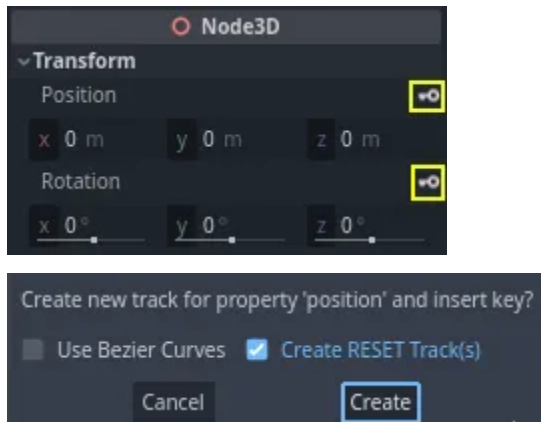


The float animation

With the animation player node, you can animate most properties on as many nodes as you need. Notice the key icon next to properties in the Inspector. You can click any of them to create a keyframe, a time and value pair for the corresponding property. The keyframe gets inserted where your time cursor is in the timeline.

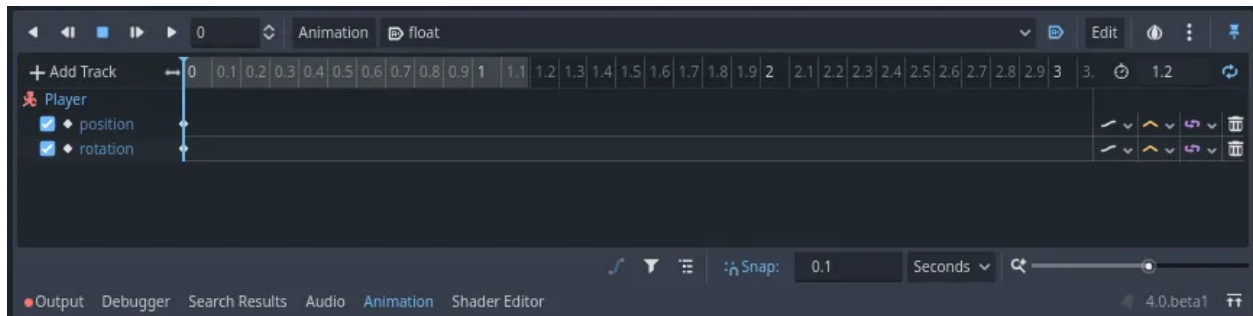
Let's insert our first keys. Here, we will animate both the position and the rotation of the Character node.

Select the Character and in the Inspector expand the Transform section. Click the key icon next to Position, and Rotation.

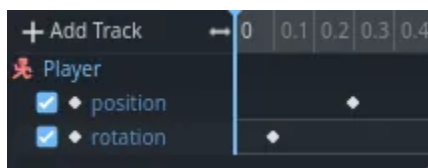


For this tutorial, just create RESET Track(s) which is the default choice

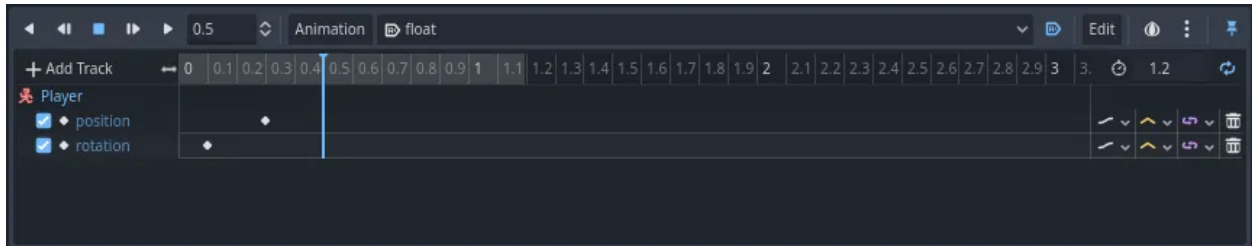
Two tracks appear in the editor with a diamond icon representing each keyframe.



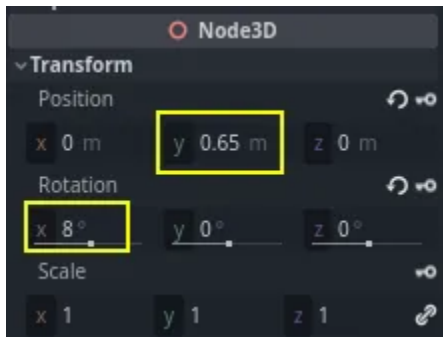
You can click and drag on the diamonds to move them in time. Move the position key to 0.3 seconds and the rotation key to 0.1 seconds.



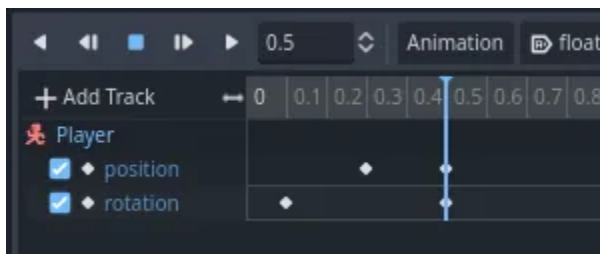
Move the time cursor to 0.5 seconds by clicking and dragging on the gray timeline.



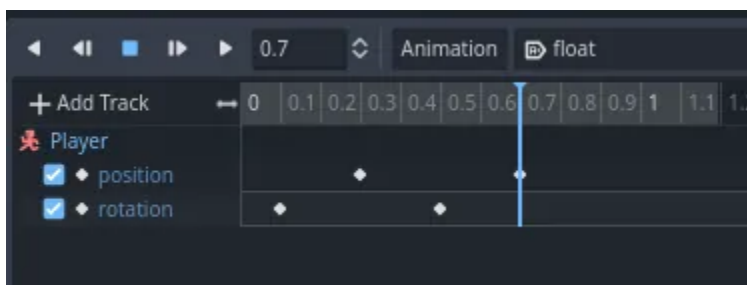
In the Inspector, set the Position's Y axis to 0.65 meters and the Rotation's X axis to 8.



Create a keyframe for both properties

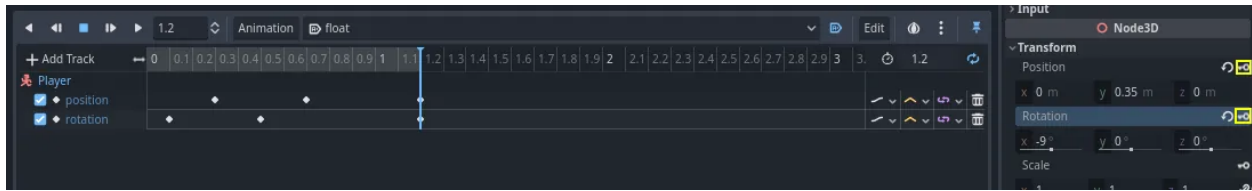


Now, move the position keyframe to 0.7 seconds by dragging it on the timeline.

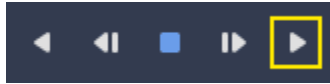


Note: A lecture on the principles of animation is beyond the scope of this tutorial. Just note that you don't want to time and space everything evenly. Instead, animators play with timing and spacing, two core animation principles. You want to offset and contrast in your character's motion to make them feel alive.

Move the time cursor to the end of the animation, at 1.2 seconds. Set the Y position to about 0.35 and the X rotation to -9 degrees. Once again, create a key for both properties.



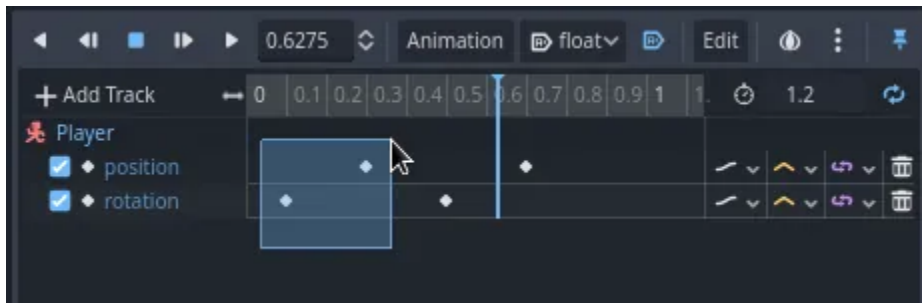
You can preview the result by clicking the play button or pressing Shift + D. Click the stop button or press S to stop playback.



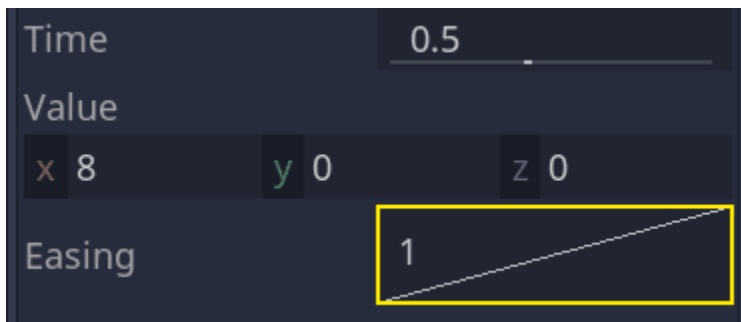
You can see that the engine interpolates between your keyframes to produce a continuous animation. At the moment, though, the motion feels very robotic. This is because the default interpolation is linear, causing constant transitions, unlike how living things move in the real world.

We can control the transition between keyframes using easing curves.

Click and drag around the first two keys in the timeline to box select them.



You can edit the properties of both keys simultaneously in the Inspector, where you can see an Easing property.

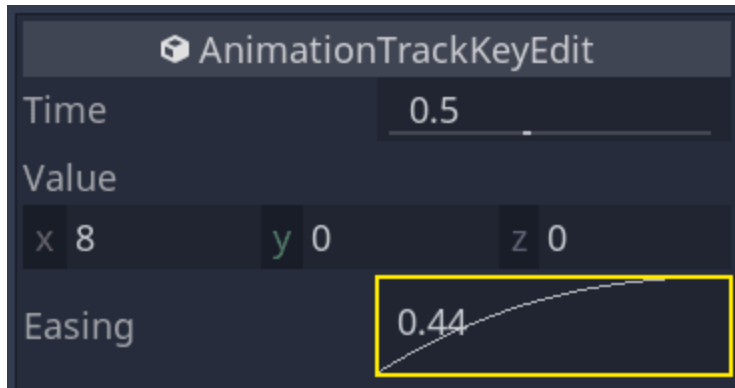


Click and drag on the curve, pulling it towards the left. This will make it ease-out, that is to say, transition fast initially and slow down as the time cursor reaches the next keyframe.

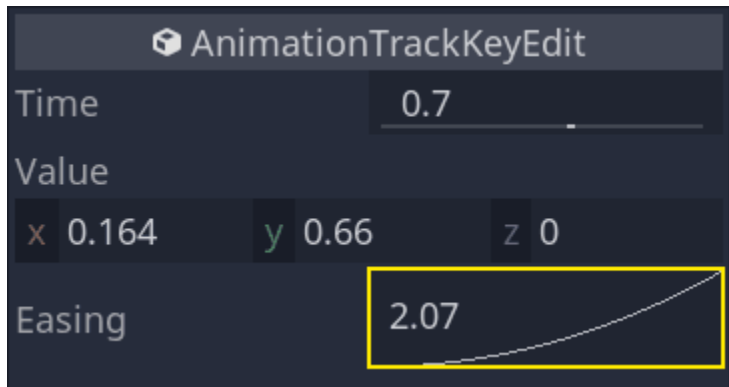


Play the animation again to see the difference. The first half should already feel a bit bouncier.

Apply an ease-out to the second keyframe in the rotation track.



Do the opposite for the second position keyframe, dragging it to the right.



Your animation should look something like this.

Note: Animations update the properties of the animated nodes every frame, overriding initial values. If we directly animated the Player node, it would prevent us from moving it in code. This is where the Pivot node comes in handy: even though we animated the Character, we can still move and rotate the Pivot and layer changes on top of the animation in a script.

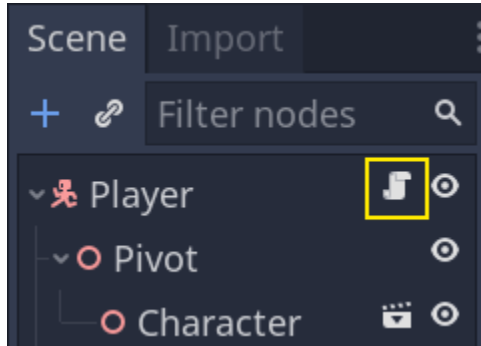
If you play the game, the player's creature will now float!

If the creature is a little too close to the floor, you can move the Pivot up to offset it.

Controlling the animation in code

We can use code to control the animation playback based on the player's input. Let's change the animation speed when the character is moving.

Open the Player's script by clicking the script icon next to it.



In `_physics_process()`, after the line where we check the direction vector, add the following code.

GDScript

```
func _physics_process(delta):
    #...
    if direction != Vector3.ZERO:
        #...
        $AnimationPlayer.speed_scale = 4
    else:
        $AnimationPlayer.speed_scale = 1
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...
    if (direction != Vector3.Zero)
    {
        // ...
        GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale = 4;
    }
    else
    {
        GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale = 1;
    }
}
```

This code makes it so when the player moves, we multiply the playback speed by 4. When they stop, we reset it to normal.

We mentioned that the Pivot could layer transforms on top of the animation. We can make the character arc when jumping using the following line of code. Add it at the end of `_physics_process()`.

GDScript

```
func _physics_process(delta):
    #...
    $Pivot.rotation.x = PI / 6 * velocity.y / jump_impulse
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // ...
    var pivot = GetNode<Node3D>("Pivot");
    pivot.Rotation = new Vector3(Mathf.Pi / 6.0f * Velocity.Y / JumpImpulse, pivot.Rotation.Y, pivot.
    ↪Rotation.Z);
}
```

Animating the mobs

Here's another nice trick with animations in Godot: as long as you use a similar node structure, you can copy them to different scenes.

For example, both the Mob and the Player scenes have a Pivot and a Character node, so we can reuse animations between them.

Open the Player scene, select the AnimationPlayer node and open the "float" animation. Next, click on Animation > Copy. Then open mob.tscn, create an AnimationPlayer child node and select it. Click Animation > Paste and make sure that the button with an "A+" icon (Autoplay on Load) and the looping arrows (Animation looping) are also turned on in the animation editor in the bottom panel. That's it; all monsters will now play the float animation.

We can change the playback speed based on the creature's random_speed. Open the Mob's script and at the end of the initialize() function, add the following line.

GDScript

```
func initialize(start_position, player_position):
    #...
    $AnimationPlayer.speed_scale = random_speed / min_speed
```

C#

```
public void Initialize(Vector3 startPosition, Vector3 playerPosition)
{
    // ...
    GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale = randomSpeed / MinSpeed;
}
```

And with that, you finished coding your first complete 3D game.

Congratulations!

In the next part, we'll quickly recap what you learned and give you some links to keep learning more. But for now, here are the complete Player.gd and Mob.gd so you can check your code against them.

Here's the Player script.

GDScript

```
extends CharacterBody3D

signal hit

# How fast the player moves in meters per second.
@export var speed = 14
# The downward acceleration while in the air, in meters per second squared.
@export var fall_acceleration = 75
# Vertical impulse applied to the character upon jumping in meters per second.
@export var jump_impulse = 20
# Vertical impulse applied to the character upon bouncing over a mob
# in meters per second.
@export var bounce_impulse = 16

var target_velocity = Vector3.ZERO

func _physics_process(delta):
    # We create a local variable to store the input direction
    var direction = Vector3.ZERO

    # We check for each move input and update the direction accordingly
    if Input.is_action_pressed("move_right"):
        direction.x = direction.x + 1
    if Input.is_action_pressed("move_left"):
        direction.x = direction.x - 1
    if Input.is_action_pressed("move_back"):
        # Notice how we are working with the vector's x and z axes.
        # In 3D, the XZ plane is the ground plane.
        direction.z = direction.z + 1
    if Input.is_action_pressed("move_forward"):
        direction.z = direction.z - 1

    # Prevent diagonal movement being very fast
    if direction != Vector3.ZERO:
        direction = direction.normalized()
        $Pivot.look_at(position + direction, Vector3.UP)
        $AnimationPlayer.speed_scale = 4
    else:
        $AnimationPlayer.speed_scale = 1

    # Ground Velocity
    target_velocity.x = direction.x * speed
    target_velocity.z = direction.z * speed

    # Vertical Velocity
    if not is_on_floor(): # If in the air, fall towards the floor
        target_velocity.y = target_velocity.y - (fall_acceleration * delta)

    # Jumping.
    if is_on_floor() and Input.is_action_just_pressed("jump"):
        target_velocity.y = jump_impulse
```

(continues on next page)

(continued from previous page)

```

# Iterate through all collisions that occurred this frame
# in C this would be for(int i = 0; i < collisions.Count; i++)
for index in range(get_slide_collision_count()):
    # We get one of the collisions with the player
    var collision = get_slide_collision(index)

    # If the collision is with ground
    if collision.get_collider() == null:
        continue

    # If the collider is with a mob
    if collision.get_collider().is_in_group("mob"):
        var mob = collision.get_collider()
        # we check that we are hitting it from above.
        if Vector3.UP.dot(collision.get_normal()) > 0.1:
            # If so, we squash it and bounce.
            mob.squash()
            target_velocity.y = bounce_impulse
            # Prevent further duplicate calls.
            break

# Moving the Character
velocity = target_velocity
move_and_slide()

$Pivot.rotation.x = PI / 6 * velocity.y / jump_impulse

# And this function at the bottom.
func die():
    hit.emit()
    queue_free()

func _on_mob_detector_body_entered(body):
    die()

```

C#

```

using Godot;

public partial class Player : CharacterBody3D
{
    // Emitted when the player was hit by a mob.
    [Signal]
    public delegate void HitEventHandler();

    // How fast the player moves in meters per second.
    [Export]
    public int Speed { get; set; } = 14;
    // The downward acceleration when in the air, in meters per second squared.
    [Export]
    public int FallAcceleration { get; set; } = 75;
}

```

(continues on next page)

(continued from previous page)

```

// Vertical impulse applied to the character upon jumping in meters per second.
[Export]
public int JumpImpulse { get; set; } = 20;
// Vertical impulse applied to the character upon bouncing over a mob in meters per second.
[Export]
public int BounceImpulse { get; set; } = 16;

private Vector3 _targetVelocity = Vector3.Zero;

public override void _PhysicsProcess(double delta)
{
    // We create a local variable to store the input direction.
    var direction = Vector3.Zero;

    // We check for each move input and update the direction accordingly.
    if (Input.IsActionPressed("move_right"))
    {
        direction.X += 1.0f;
    }
    if (Input.IsActionPressed("move_left"))
    {
        direction.X -= 1.0f;
    }
    if (Input.IsActionPressed("move_back"))
    {
        // Notice how we are working with the vector's X and Z axes.
        // In 3D, the XZ plane is the ground plane.
        direction.Z += 1.0f;
    }
    if (Input.IsActionPressed("move_forward"))
    {
        direction.Z -= 1.0f;
    }

    // Prevent diagonal movement being very fast.
    if (direction != Vector3.Zero)
    {
        direction = direction.Normalized();
        GetNode<Node3D>("Pivot").LookAt(Position + direction, Vector3.Up);
        GetNode<AnimationPlayer>("AnimationPlayer").PlaybackSpeed = 4;
    }
    else
    {
        GetNode<AnimationPlayer>("AnimationPlayer").PlaybackSpeed = 1;
    }

    // Ground velocity
    _targetVelocity.X = direction.X * Speed;
    _targetVelocity.Z = direction.Z * Speed;

    // Vertical velocity
    if (!IsOnFloor())

```

(continues on next page)

(continued from previous page)

```

{
    _targetVelocity.Y -= FallAcceleration * (float)delta;
}

// Jumping.
if (IsOnFloor() && Input.IsActionJustPressed("jump"))
{
    _targetVelocity.Y += JumpImpulse;
}

// Iterate through all collisions that occurred this frame.
for (int index = 0; index < GetSlideCollisionCount(); index++)
{
    // We get one of the collisions with the player.
    KinematicCollision3D collision = GetSlideCollision(index);

    // If the collision is with a mob.
    if (collision.GetCollider() is Mob mob)
    {
        // We check that we are hitting it from above.
        if (Vector3.Up.Dot(collision.GetNormal()) > 0.1f)
        {
            // If so, we squash it and bounce.
            mob.Squash();
            _targetVelocity.Y = BounceImpulse;
            // Prevent further duplicate calls.
            break;
        }
    }
}

// Moving the character
Velocity = _targetVelocity;
MoveAndSlide();

var pivot = GetNode<Node3D>("Pivot");
pivot.Rotation = new Vector3(Mathf.Pi / 6.0f * Velocity.Y / JumpImpulse, pivot.Rotation.Y, pivot.
↪Rotation.Z);
}

private void Die()
{
    EmitSignal(SignalName.Hit);
    QueueFree();
}

private void OnMobDetectorBodyEntered(Node body)
{
    Die();
}
}

```

And the Mob's script.

GDScript

```
extends CharacterBody3D

# Minimum speed of the mob in meters per second.
@export var min_speed = 10
# Maximum speed of the mob in meters per second.
@export var max_speed = 18

# Emitted when the player jumped on the mob
signal squashed

func _physics_process(_delta):
    move_and_slide()

# This function will be called from the Main scene.
func initialize(start_position, player_position):
    # We position the mob by placing it at start_position
    # and rotate it towards player_position, so it looks at the player.
    look_at_from_position(start_position, player_position, Vector3.UP)
    # Rotate this mob randomly within range of -90 and +90 degrees,
    # so that it doesn't move directly towards the player.
    rotate_y(randf_range(-PI / 4, PI / 4))

    # We calculate a random speed (integer)
    var random_speed = randi_range(min_speed, max_speed)
    # We calculate a forward velocity that represents the speed.
    velocity = Vector3.FORWARD * random_speed
    # We then rotate the velocity vector based on the mob's Y rotation
    # in order to move in the direction the mob is looking.
    velocity = velocity.rotated(Vector3.UP, rotation.y)

    $AnimationPlayer.speed_scale = random_speed / min_speed

func _on_visible_on_screen_notifier_3d_screen_exited():
    queue_free()

func squash():
    squashed.emit()
    queue_free() # Destroy this node
```

C#

```
using Godot;

public partial class Mob : CharacterBody3D
{
    // Emitted when the player jumped on the mob.
    [Signal]
    public delegate void SquashedEventHandler();

    // Minimum speed of the mob in meters per second
    [Export]
    public int MinSpeed { get; set; } = 10;
```

(continues on next page)

(continued from previous page)

```

// Maximum speed of the mob in meters per second
[Export]
public int MaxSpeed { get; set; } = 18;

public override void _PhysicsProcess(double delta)
{
    MoveAndSlide();
}

// This function will be called from the Main scene.
public void Initialize(Vector3 startPosition, Vector3 playerPosition)
{
    // We position the mob by placing it at startPosition
    // and rotate it towards playerPosition, so it looks at the player.
    LookAtFromPosition(startPosition, playerPosition, Vector3.Up);
    // Rotate this mob randomly within range of -90 and +90 degrees,
    // so that it doesn't move directly towards the player.
    RotateY((float)GD.RandRange(-Mathf.Pi / 4.0, Mathf.Pi / 4.0));

    // We calculate a random speed (integer).
    int randomSpeed = GD.RandRange(MinSpeed, MaxSpeed);
    // We calculate a forward velocity that represents the speed.
    Velocity = Vector3.Forward * randomSpeed;
    // We then rotate the velocity vector based on the mob's Y rotation
    // in order to move in the direction the mob is looking.
    Velocity = Velocity.Rotated(Vector3.Up, Rotation.Y);

    GetNode<AnimationPlayer>("AnimationPlayer").SpeedScale = randomSpeed / MinSpeed;
}

public void Squash()
{
    EmitSignal(SignalName.Squashed);
    QueueFree(); // Destroy this node
}

private void OnVisibilityNotifierScreenExited()
{
    QueueFree();
}
}

```

Going further

You can pat yourself on the back for having completed your first 3D game with Godot.

In this series, we went over a wide range of techniques and editor features. Hopefully, you've witnessed how intuitive Godot's scene system can be and learned a few tricks you can apply in your projects.

But we just scratched the surface: Godot has a lot more in store for you to save time creating games. And you can learn all that by browsing the documentation.

Where should you begin? Below, you'll find a few pages to start exploring and build upon what you've learned so far.

But before that, here's a link to download a completed version of the project: <https://github.com/godotengine/godot-3d-dodge-the-creeps>.

Exploring the manual

The manual is your ally whenever you have a doubt or you're curious about a feature. It does not contain tutorials about specific game genres or mechanics. Instead, it explains how Godot works in general. In it, you will find information about 2D, 3D, physics, rendering and performance, and much more.

Here are the sections we recommend you to explore next:

1. Read the Scripting section to learn essential programming features you'll use in every project.
2. The 3D and Physics sections will teach you more about 3D game creation in the engine.
3. Inputs is another important one for any game project.

You can start with these or, if you prefer, look at the sidebar menu on the left and pick your options.

We hope you enjoyed this tutorial series, and we're looking forward to seeing what you achieve using Godot.

2.11 Best practices

2.11.1 Introduction

This series is a collection of best practices to help you work efficiently with Godot.

Godot allows for a great amount of flexibility in how you structure a project's codebase and break it down into scenes. Each approach has its pros and cons, and they can be hard to weigh until you've worked with the engine for long enough.

There are always many ways to structure your code and solve specific programming problems. It would be impossible to cover them all here.

That is why each article starts from a real-world problem. We will break down each problem in fundamental questions, suggest solutions, analyze the pros and cons of each option, and highlight the best course of action for the problem at hand.

You should start by reading [Applying object-oriented principles in Godot](#). It explains how Godot's nodes and scenes relate to classes and objects in other Object-Oriented programming languages. It will help you make sense of the rest of the series.

Note: The best practices in Godot rely on Object-Oriented design principles. We use tools like the [single responsibility](#) principle and [encapsulation](#).

2.11.2 Applying object-oriented principles in Godot

The engine offers two main ways to create reusable objects: scripts and scenes. Neither of these technically define classes under the hood.

Still, many best practices using Godot involve applying object-oriented programming principles to the scripts and scenes that compose your game. That is why it's useful to understand how we can think of them as classes.

This guide briefly explains how scripts and scenes work in the engine's core to help you understand how they work under the hood.

How scripts work in the engine

The engine provides built-in classes like `Node`. You can extend those to create derived types using a script.

These scripts are not technically classes. Instead, they are resources that tell the engine a sequence of initializations to perform on one of the engine's built-in classes.

Godot's internal classes have methods that register a class's data with a `ClassDB`. This database provides runtime access to class information. `ClassDB` contains information about classes like:

- Properties.
- Methods.
- Constants.
- Signals.

This `ClassDB` is what objects check against when performing an operation like accessing a property or calling a method. It checks the database's records and the object's base types' records to see if the object supports the operation.

Attaching a Script to your object extends the methods, properties, and signals available from the `ClassDB`.

Note: Even scripts that don't use the `extends` keyword implicitly inherit from the engine's base `RefCounted` class. As a result, you can instantiate scripts without the `extends` keyword from code. Since they extend `RefCounted` though, you cannot attach them to a `Node`.

Scenes

The behavior of scenes has many similarities to classes, so it can make sense to think of a scene as a class. Scenes are reusable, instantiable, and inheritable groups of nodes. Creating a scene is similar to having a script that creates nodes and adds them as children using `add_child()`.

We often pair a scene with a scripted root node that makes use of the scene's nodes. As such, the script extends the scene by adding behavior through imperative code.

The content of a scene helps to define:

- What nodes are available to the script.
- How they are organized.
- How they are initialized.
- What signal connections they have with each other.

Why is any of this important to scene organization? Because instances of scenes are objects. As a result, many object-oriented principles that apply to written code also apply to scenes: single responsibility, encapsulation, and others.

The scene is always an extension of the script attached to its root node, so you can interpret it as part of a class.

Most of the techniques explained in this best practices series build on this point.

2.11.3 Scene organization

This article covers topics related to the effective organization of scene content. Which nodes should one use? Where should one place them? How should they interact?

How to build relationships effectively

When Godot users begin crafting their own scenes, they often run into the following problem:

They create their first scene and fill it with content only to eventually end up saving branches of their scene into separate scenes as the nagging feeling that they should split things up starts to accumulate. However, they then notice that the hard references they were able to rely on before are no longer possible. Re-using the scene in multiple places creates issues because the node paths do not find their targets and signal connections established in the editor break.

To fix these problems, one must instantiate the sub-scenes without them requiring details about their environment. One needs to be able to trust that the sub-scene will create itself without being picky about how one uses it.

One of the biggest things to consider in OOP is maintaining focused, singular-purpose classes with [loose coupling](#) to other parts of the codebase. This keeps the size of objects small (for maintainability) and improves their reusability.

These OOP best practices have several implications for best practices in scene structure and script usage.

If at all possible, one should design scenes to have no dependencies. That is, one should create scenes that keep everything they need within themselves.

If a scene must interact with an external context, experienced developers recommend the use of [Dependency Injection](#). This technique involves having a high-level API provide the dependencies of the low-level API. Why do this? Because classes which rely on their external environment can inadvertently trigger bugs and unexpected behavior.

To do this, one must expose data and then rely on a parent context to initialize it:

1. Connect to a signal. Extremely safe, but should be used only to "respond" to behavior, not start it. By convention, signal names are usually past-tense verbs like "entered", "skill_activated", or "item_collected".

GDScript

```
# Parent
$Child.signal_name.connect(method_on_the_object)

# Child
signal_name.emit() # Triggers parent-defined behavior.
```

C#

```
// Parent
GetNode("Child").Connect("SignalName", ObjectWithMethod, "MethodOnTheObject");

// Child
EmitSignal("SignalName"); // Triggers parent-defined behavior.
```

2. Call a method. Used to start behavior.

GDScript

```
# Parent
$Child.method_name = "do"

# Child, assuming it has String property 'method_name' and method 'do'.
call(method_name) # Call parent-defined method (which child must own).
```

C#

```
// Parent
GetNode("Child").Set("MethodName", "Do");

// Child
Call(MethodName); // Call parent-defined method (which child must own).
```

3. Initialize a Callable property. Safer than a method as ownership of the method is unnecessary. Used to start behavior.

GDScript

```
# Parent
$Child.func_property = object_with_method.method_on_the_object

# Child
func_property.call() # Call parent-defined method (can come from anywhere).
```

C#

```
// Parent
GetNode("Child").Set("FuncProperty", Callable.From(ObjectWithMethod.MethodOnTheObject));

// Child
FuncProperty.Call(); // Call parent-defined method (can come from anywhere).
```

4. Initialize a Node or other Object reference.

GDScript

```
# Parent
$Child.target = self

# Child
print(target) # Use parent-defined node.
```

C#

```
// Parent
GetNode("Child").Set("Target", this);

// Child
GD.Print(Target); // Use parent-defined node.
```

5. Initialize a NodePath.

GDScript

```
# Parent
$Child.target_path = ".."

# Child
get_node(target_path) # Use parent-defined NodePath.
```

C#

```
// Parent
GetNode("Child").Set("TargetPath", NodePath(".."));

// Child
GetNode(TargetPath); // Use parent-defined NodePath.
```

These options hide the points of access from the child node. This in turn keeps the child loosely coupled to its environment. One can reuse it in another context without any extra changes to its API.

Note: Although the examples above illustrate parent-child relationships, the same principles apply towards all object relations. Nodes which are siblings should only be aware of their hierarchies while an ancestor mediates their communications and references.

GDScript

```
# Parent
$Left.target = $Right.get_node("Receiver")

# Left
var target: Node
func execute():
    # Do something with 'target'.

# Right
func _init():
    var receiver = Receiver.new()
    add_child(receiver)
```

C#

```
// Parent
GetNode<Left>("Left").Target = GetNode("Right/Receiver");

public partial class Left : Node
{
    public Node Target = null;

    public void Execute()
    {
        // Do something with 'Target'.
    }
}

public partial class Right : Node
{
```

(continues on next page)

(continued from previous page)

```
public Node Receiver = null;

public Right()
{
    Receiver = ResourceLoader.Load<Script>("Receiver.cs").New();
    AddChild(Receiver);
}
}
```

The same principles also apply to non-Node objects that maintain dependencies on other objects. Whichever object actually owns the objects should manage the relationships between them.

Warning: One should favor keeping data in-house (internal to a scene) though as placing a dependency on an external context, even a loosely coupled one, still means that the node will expect something in its environment to be true. The project's design philosophies should prevent this from happening. If not, the code's inherent liabilities will force developers to use documentation to keep track of object relations on a microscopic scale; this is otherwise known as development hell. Writing code that relies on external documentation for one to use it safely is error-prone by default.

To avoid creating and maintaining such documentation, one converts the dependent node ("child" above) into a tool script that implements `_get_configuration_warnings()`. Returning a non-empty PackedStringArray from it will make the Scene dock generate a warning icon with the string(s) as a tooltip by the node. This is the same icon that appears for nodes such as the Area2D node when it has no child CollisionShape2D nodes defined. The editor then self-documents the scene through the script code. No content duplication via documentation is necessary.

A GUI like this can better inform project users of critical information about a Node. Does it have external dependencies? Have those dependencies been satisfied? Other programmers, and especially designers and writers, will need clear instructions in the messages telling them what to do to configure it.

So, why does all this complex switcharoo work? Well, because scenes operate best when they operate alone. If unable to work alone, then working with others anonymously (with minimal hard dependencies, i.e. loose coupling) is the next best thing. Inevitably, changes may need to be made to a class and if these changes cause it to interact with other scenes in unforeseen ways, then things will start to break down. The whole point of all this indirection is to avoid ending up in a situation where changing one class results in adversely effecting other classes dependent on it.

Scripts and scenes, as extensions of engine classes, should abide by all OOP principles. Examples include...

- SOLID
- DRY
- KISS
- YAGNI

Choosing a node tree structure

So, a developer starts work on a game only to stop at the vast possibilities before them. They might know what they want to do, what systems they want to have, but where to put them all? Well, how one goes about making their game is always up to them. One can construct node trees in countless ways. But, for those who are unsure, this helpful guide can give them a sample of a decent structure to start with.

A game should always have a sort of "entry point"; somewhere the developer can definitively track where things begin so that they can follow the logic as it continues elsewhere. This place also serves as a bird's eye view of all of the other data and logic in the program. For traditional applications, this would be the "main" function. In this case, it would be a Main node.

- Node "Main" (main.gd)

The main.gd script would then serve as the primary controller of one's game.

Then one has their actual in-game "World" (a 2D or 3D one). This can be a child of Main. In addition, one will need a primary GUI for their game that manages the various menus and widgets the project needs.

- Node "Main" (main.gd)
 - Node2D/Node3D "World" (game_world.gd)
 - Control "GUI" (gui.gd)

When changing levels, one can then swap out the children of the "World" node. Changing scenes manually gives users full control over how their game world transitions.

The next step is to consider what gameplay systems one's project requires. If one has a system that...

1. tracks all of its data internally
2. should be globally accessible
3. should exist in isolation

... then one should create an autoload 'singleton' node.

Note: For smaller games, a simpler alternative with less control would be to have a "Game" singleton that simply calls the `SceneTree.change_scene_to_file()` method to swap out the main scene's content. This structure more or less keeps the "World" as the main game node.

Any GUI would need to also be a singleton; be a transitory part of the "World"; or be manually added as a direct child of the root. Otherwise, the GUI nodes would also delete themselves during scene transitions.

If one has systems that modify other systems' data, one should define those as their own scripts or scenes rather than autoloads. For more information on the reasons, please see the Autoloads versus regular nodes documentation.

Each subsystem within one's game should have its own section within the SceneTree. One should use parent-child relationships only in cases where nodes are effectively elements of their parents. Does removing the parent reasonably mean that one should also remove the children? If not, then it should have its own place in the hierarchy as a sibling or some other relation.

Note: In some cases, one needs these separated nodes to also position themselves relative to each other. One can use the RemoteTransform / RemoteTransform2D nodes for this purpose. They will allow a target node to conditionally inherit selected transform elements from the Remote* node. To assign the target NodePath, use one of the following:

1. A reliable third party, likely a parent node, to mediate the assignment.

2. A group, to easily pull a reference to the desired node (assuming there will only ever be one of the targets).

When should one do this? Well, this is subjective. The dilemma arises when one must micro-manage when a node must move around the SceneTree to preserve itself. For example...

- Add a "player" node to a "room".
- Need to change rooms, so one must delete the current room.
- Before the room can be deleted, one must preserve and/or move the player.

Is memory a concern?

- If not, one can just create the two rooms, move the player and delete the old one. No problem.

If so, one will need to...

- Move the player somewhere else in the tree.
- Delete the room.
- Instantiate and add the new room.
- Re-add the player.

The issue is that the player here is a "special case"; one where the developers must know that they need to handle the player this way for the project. As such, the only way to reliably share this information as a team is to document it. Keeping implementation details in documentation however is dangerous. It's a maintenance burden, strains code readability, and bloats the intellectual content of a project unnecessarily.

In a more complex game with larger assets, it can be a better idea to simply keep the player somewhere else in the SceneTree entirely. This results in:

1. More consistency.
2. No "special cases" that must be documented and maintained somewhere.
3. No opportunity for errors to occur because these details are not accounted for.

In contrast, if one ever needs to have a child node that does not inherit the transform of their parent, one has the following options:

1. The declarative solution: place a Node in between them. As nodes with no transform, Nodes will not pass along such information to their children.
2. The imperative solution: Use the `top_level` property for the CanvasItem or Node3D node. This will make the node ignore its inherited transform.

Note: If building a networked game, keep in mind which nodes and gameplay systems are relevant to all players versus those just pertinent to the authoritative server. For example, users do not all need to have a copy of every players' "PlayerController" logic. Instead, they need only their own. As such, keeping these in a separate branch from the "world" can help simplify the management of game connections and the like.

The key to scene organization is to consider the SceneTree in relational terms rather than spatial terms. Are the nodes dependent on their parent's existence? If not, then they can thrive all by themselves somewhere else. If they are dependent, then it stands to reason that they should be children of that parent (and likely part of that parent's scene if they aren't already).

Does this mean nodes themselves are components? Not at all. Godot's node trees form an aggregation relationship, not one of composition. But while one still has the flexibility to move nodes around, it is still best when such moves are unnecessary by default.

2.11.4 When to use scenes versus scripts

We've already covered how scenes and scripts are different. Scripts define an engine class extension with imperative code, scenes with declarative code.

Each system's capabilities are different as a result. Scenes can define how an extended class initializes, but not what its behavior actually is. Scenes are often used in conjunction with a script, the scene declaring a composition of nodes, and the script adding behaviour with imperative code.

Anonymous types

It is possible to completely define a scenes' contents using a script alone. This is, in essence, what the Godot Editor does, only in the C++ constructor of its objects.

But, choosing which one to use can be a dilemma. Creating script instances is identical to creating in-engine classes whereas handling scenes requires a change in API:

GDScript

```
const MyNode = preload("my_node.gd")
const MyScene = preload("my_scene.tscn")
var node = Node.new()
var my_node = MyNode.new() # Same method call.
var my_scene = MyScene.instantiate() # Different method call.
var my_inherited_scene = MyScene.instantiate(PackedScene.GEN_EDIT_STATE_MAIN) # Create_
↳scene inheriting from MyScene.
```

C#

```
using Godot;

public partial class Game : Node
{
    public static CSharpScript MyNode { get; } =
        GD.Load<CSharpScript>("res://Path/To/MyNode.cs");
    public static PackedScene MyScene { get; } =
        GD.Load<PackedScene>("res://Path/To/MyScene.tscn");
    private Node _node;
    private Node _myNode;
    private Node _myScene;
    private Node _myInheritedScene;

    public Game()
    {
        _node = new Node();
        _myNode = MyNode.New().As<Node>();
        // Different than calling new() or MyNode.New(). Instantiated from a PackedScene.
        _myScene = MyScene.Instantiate();
        // Create scene inheriting from MyScene.
        _myInheritedScene = MyScene.Instantiate(PackedScene.GenEditState.Main);
    }
}
```

Also, scripts will operate a little slower than scenes due to the speed differences between engine and script code. The larger and more complex the node, the more reason there is to build it as a scene.

Named types

Scripts can be registered as a new type within the editor itself. This displays it as a new type in the node or resource creation dialog with an optional icon. This way, the user's ability to use the script is much more streamlined. Rather than having to...

1. Know the base type of the script they would like to use.
2. Create an instance of that base type.
3. Add the script to the node.

With a registered script, the scripted type instead becomes a creation option like the other nodes and resources in the system. The creation dialog even has a search bar to look up the type by name.

There are two systems for registering types:

- Custom Types
 - Editor-only. Typenames are not accessible at runtime.
 - Does not support inherited custom types.
 - An initializer tool. Creates the node with the script. Nothing more.
 - Editor has no type-awareness of the script or its relationship to other engine types or scripts.
 - Allows users to define an icon.
 - Works for all scripting languages because it deals with Script resources in abstract.
 - Set up using `EditorPlugin.add_custom_type`.
- Script Classes
 - Editor and runtime accessible.
 - Displays inheritance relationships in full.
 - Creates the node with the script, but can also change types or extend the type from the editor.
 - Editor is aware of inheritance relationships between scripts, script classes, and engine C++ classes.
 - Allows users to define an icon.
 - Engine developers must add support for languages manually (both name exposure and runtime accessibility).
 - Godot 3.1+ only.
 - The Editor scans project folders and registers any exposed names for all scripting languages. Each scripting language must implement its own support for exposing this information.

Both methodologies add names to the creation dialog, but script classes, in particular, also allow for users to access the typename without loading the script resource. Creating instances and accessing constants or static methods is viable from anywhere.

With features like these, one may wish their type to be a script without a scene due to the ease of use it grants users. Those developing plugins or creating in-house tools for designers to use will find an easier time of things this way.

On the downside, it also means having to use largely imperative programming.

Performance of Script vs PackedScene

One last aspect to consider when choosing scenes and scripts is execution speed.

As the size of objects increases, the scripts' necessary size to create and initialize them grows much larger. Creating node hierarchies demonstrates this. Each Node's logic could be several hundred lines of code in length.

The code example below creates a new Node, changes its name, assigns a script to it, sets its future parent as its owner so it gets saved to disk along with it, and finally adds it as a child of the Main node:

GDScript

```
# main.gd
extends Node

func _init():
    var child = Node.new()
    child.name = "Child"
    child.script = preload("child.gd")
    add_child(child)
    child.owner = self
```

C#

```
using Godot;

public partial class Main : Node
{
    public Node Child { get; set; }

    public Main()
    {
        Child = new Node();
        Child.Name = "Child";
        var childID = Child.GetInstanceId();
        Child.SetScript(GD.Load<Script>("res://Path/To/Child.cs"));
        // SetScript() causes the C# wrapper object to be disposed, so obtain a new
        // wrapper for the Child node using its instance ID before proceeding.
        Child = (Node)GodotObject.InstanceFromId(childID);
        AddChild(Child);
        Child.Owner = this;
    }
}
```

Script code like this is much slower than engine-side C++ code. Each instruction makes a call to the scripting API which leads to many "lookups" on the back-end to find the logic to execute.

Scenes help to avoid this performance issue. PackedScene, the base type that scenes inherit from, defines resources that use serialized data to create objects. The engine can process scenes in batches on the back-end and provide much better performance than scripts.

Conclusion

In the end, the best approach is to consider the following:

- If one wishes to create a basic tool that is going to be re-used in several different projects and which people of all skill levels will likely use (including those who don't label themselves as "programmers"), then chances are that it should probably be a script, likely one with a custom name/icon.
- If one wishes to create a concept that is particular to their game, then it should always be a scene. Scenes are easier to track/edit and provide more security than scripts.
- If one would like to give a name to a scene, then they can still sort of do this by declaring a script class and giving it a scene as a constant. The script becomes, in effect, a namespace:

GDScript

```
# game.gd
class_name Game # extends RefCounted, so it won't show up in the node creation dialog.
extends RefCounted

const MyScene = preload("my_scene.tscn")

# main.gd
extends Node
func _ready():
    add_child(Game.MyScene.instantiate())
```

C#

```
// Game.cs
public partial class Game : RefCounted
{
    public static PackedScene MyScene { get; } =
        GD.Load<PackedScene>("res://Path/To/MyScene.tscn");
}

// Main.cs
public partial class Main : Node
{
    public override void _Ready()
    {
        AddChild(Game.MyScene.Instantiate());
    }
}
```

2.11.5 Autoloads versus regular nodes

Godot offers a feature to automatically load nodes at the root of your project, allowing you to access them globally, that can fulfill the role of a Singleton: Singletons (Autoload). These autoloaded nodes are not freed when you change the scene from code with `SceneTree.change_scene_to_file`.

In this guide, you will learn when to use the Autoload feature, and techniques you can use to avoid it.

The cutting audio issue

Other engines can encourage the use of creating manager classes, singletons that organize a lot of functionality into a globally accessible object. Godot offers many ways to avoid global state thanks to the node tree and signals.

For example, let's say we are building a platformer and want to collect coins that play a sound effect. There's a node for that: the `AudioStreamPlayer`. But if we call the `AudioStreamPlayer` while it is already playing a sound, the new sound interrupts the first.

A solution is to code a global, autoloading sound manager class. It generates a pool of `AudioStreamPlayer` nodes that cycle through as each new request for sound effects comes in. Say we call that class `Sound`, you can use it from anywhere in your project by calling `Sound.play("coin_pickup.ogg")`. This solves the problem in the short term but causes more problems:

1. Global state: one object is now responsible for all objects' data. If the `Sound` class has errors or doesn't have an `AudioStreamPlayer` available, all the nodes calling it can break.
2. Global access: now that any object can call `Sound.play(sound_path)` from anywhere, there's no longer an easy way to find the source of a bug.
3. Global resource allocation: with a pool of `AudioStreamPlayer` nodes stored from the start, you can either have too few and face bugs, or too many and use more memory than you need.

Note: About global access, the problem is that any code anywhere could pass wrong data to the `Sound` autoload in our example. As a result, the domain to explore to fix the bug spans the entire project.

When you keep code inside a scene, only one or two scripts may be involved in audio.

Contrast this with each scene keeping as many `AudioStreamPlayer` nodes as it needs within itself and all these problems go away:

1. Each scene manages its own state information. If there is a problem with the data, it will only cause issues in that one scene.
2. Each scene accesses only its own nodes. Now, if there is a bug, it's easy to find which node is at fault.
3. Each scene allocates exactly the amount of resources it needs.

Managing shared functionality or data

Another reason to use an Autoload can be that you want to reuse the same method or data across many scenes.

In the case of functions, you can create a new type of `Node` that provides that feature for an individual scene using the `class_name` keyword in `GDScript`.

When it comes to data, you can either:

1. Create a new type of `Resource` to share the data.
2. Store the data in an object to which each node has access, for example using the `owner` property to access the scene's root node.

When you should use an Autoload

GDScript supports the creation of static functions using `static func`. When combined with `class_name`, this makes it possible to create libraries of helper functions without having to create an instance to call them. The limitation of static functions is that they can't reference member variables, non-static functions or `self`.

Since Godot 4.1, GDScript also supports static variables using `static var`. This means you can now share a variables across instances of a class without having to create a separate autoload.

Still, autoloaded nodes can simplify your code for systems with a wide scope. If the autoload is managing its own information and not invading the data of other objects, then it's a great way to create systems that handle broad-scoped tasks. For example, a quest or a dialogue system.

Note: An autoload is not necessarily a singleton. Nothing prevents you from instantiating copies of an autoloaded node. An autoload is only a tool that makes a node load automatically as a child of the root of your scene tree, regardless of your game's node structure or which scene you run, e.g. by pressing the F6 key.

As a result, you can get the autoloaded node, for example an autoload called `Sound`, by calling `get_node("/root/Sound")`.

2.11.6 When and how to avoid using nodes for everything

Nodes are cheap to produce, but even they have their limits. A project may have tens of thousands of nodes all doing things. The more complex their behavior though, the larger the strain each one adds to a project's performance.

Godot provides more lightweight objects for creating APIs which nodes use. Be sure to keep these in mind as options when designing how you wish to build your project's features.

1. **Object:** The ultimate lightweight object, the original `Object` must use manual memory management. With that said, it isn't too difficult to create one's own custom data structures, even node structures, that are also lighter than the `Node` class.
 - **Example:** See the `Tree` node. It supports a high level of customization for a table of content with an arbitrary number of rows and columns. The data that it uses to generate its visualization though is actually a tree of `TreeItem` Objects.
 - **Advantages:** Simplifying one's API to smaller scoped objects helps improve its accessibility and improve iteration time. Rather than working with the entire `Node` library, one creates an abbreviated set of Objects from which a node can generate and manage the appropriate sub-nodes.

Note: One should be careful when handling them. One can store an `Object` into a variable, but these references can become invalid without warning. For example, if the object's creator decides to delete it out of nowhere, this would trigger an error state when one next accesses it.

2. **RefCounted:** Only a little more complex than `Object`. They track references to themselves, only deleting loaded memory when no further references to themselves exist. These are useful in the majority of cases where one needs data in a custom class.
 - **Example:** See the `FileAccess` object. It functions just like a regular `Object` except that one need not delete it themselves.
 - **Advantages:** same as the `Object`.
3. **Resource:** Only slightly more complex than `RefCounted`. They have the innate ability to serialize/deserialize (i.e. save and load) their object properties to/from Godot resource files.

- Example: Scripts, PackedScene (for scene files), and other types like each of the AudioEffect classes. Each of these can be save and loaded, therefore they extend from Resource.
- Advantages: Much has already been said on Resource's advantages over traditional data storage methods. In the context of using Resources over Nodes though, their main advantage is in Inspector-compatibility. While nearly as lightweight as Object/RefCounted, they can still display and export properties in the Inspector. This allows them to fulfill a purpose much like sub-Nodes on the usability front, but also improve performance if one plans to have many such Resources/Nodes in their scenes.

2.11.7 Godot interfaces

Often one needs scripts that rely on other objects for features. There are 2 parts to this process:

1. Acquiring a reference to the object that presumably has the features.
2. Accessing the data or logic from the object.

The rest of this tutorial outlines the various ways of doing all this.

Acquiring object references

For all Objects, the most basic way of referencing them is to get a reference to an existing object from another acquired instance.

GDScript

```
var obj = node.object # Property access.  
var obj = node.get_object() # Method access.
```

C#

```
GodotObject obj = node.Object; // Property access.  
GodotObject obj = node.GetObject(); // Method access.
```

The same principle applies for RefCounted objects. While users often access Node and Resource this way, alternative measures are available.

Instead of property or method access, one can get Resources by load access.

GDScript

```
# If you need an "export const var" (which doesn't exist), use a conditional  
# setter for a tool script that checks if it's executing in the editor.  
# The `@tool` annotation must be placed at the top of the script.  
@tool  
  
# Load resource during scene load.  
var preres = preload(path)  
# Load resource when program reaches statement.  
var res = load(path)  
  
# Note that users load scenes and scripts, by convention, with PascalCase  
# names (like typename), often into constants.  
const MyScene = preload("my_scene.tscn") # Static load  
const MyScript = preload("my_script.gd")  
  
# This type's value varies, i.e. it is a variable, so it uses snake_case.
```

(continues on next page)

(continued from previous page)

```

@export var script_type: Script

# Must configure from the editor, defaults to null.
@export var const_script: Script:
    set(value):
        if Engine.is_editor_hint():
            const_script = value

# Warn users if the value hasn't been set.
func _get_configuration_warnings():
    if not const_script:
        return ["Must initialize property 'const_script'."]

    return []

```

C#

```

// Tool script added for the sake of the "const [Export]" example.
[Tool]
public MyType
{
    // Property initializations load during Script instancing, i.e. .new().
    // No "preload" loads during scene load exists in C#.

    // Initialize with a value. Editable at runtime.
    public Script MyScript = GD.Load<Script>("res://Path/To/MyScript.cs");

    // Initialize with same value. Value cannot be changed.
    public readonly Script MyConstScript = GD.Load<Script>("res://Path/To/MyScript.cs");

    // Like 'readonly' due to inaccessible setter.
    // But, value can be set during constructor, i.e. MyType().
    public Script MyNoSetScript { get; } = GD.Load<Script>("res://Path/To/MyScript.cs");

    // If need a "const [Export]" (which doesn't exist), use a
    // conditional setter for a tool script that checks if it's executing
    // in the editor.
    private PackedScene _enemyScn;

    [Export]
    public PackedScene EnemyScn
    {
        get { return _enemyScn; }
        set
        {
            if (Engine.IsEditorHint())
            {
                _enemyScn = value;
            }
        }
    }
};

```

(continues on next page)

(continued from previous page)

```
// Warn users if the value hasn't been set.
public string[] _GetConfigurationWarnings()
{
    if (EnemyScn == null)
    {
        return new string[] { "Must initialize property 'EnemyScn'." };
    }
    return Array.Empty<string>();
}
```

Note the following:

1. There are many ways in which a language can load such resources.
2. When designing how objects will access data, don't forget that one can pass resources around as references as well.
3. Keep in mind that loading a resource fetches the cached resource instance maintained by the engine. To get a new object, one must duplicate an existing reference or instantiate one from scratch with `new()`.

Nodes likewise have an alternative access point: the `SceneTree`.

GDScript

```
extends Node

# Slow.
func dynamic_lookup_with_dynamic_nodepath():
    print(get_node("Child"))

# Faster. GDScript only.
func dynamic_lookup_with_cached_nodepath():
    print($Child)

# Fastest. Doesn't break if node moves later.
# Note that `@onready` annotation is GDScript-only.
# Other languages must do...
#     var child
#     func _ready():
#         child = get_node("Child")
@onready var child = $Child
func lookup_and_cache_for_future_access():
    print(child)

# Fastest. Doesn't break if node is moved in the Scene tree dock.
# Node must be selected in the inspector as it's an exported property.
@export var child: Node
func lookup_and_cache_for_future_access():
    print(child)

# Delegate reference assignment to an external source.
# Con: need to perform a validation check.
```

(continues on next page)

(continued from previous page)

```

# Pro: node makes no requirements of its external structure.
# 'prop' can come from anywhere.
var prop
func call_me_after_prop_is_initialized_by_parent():
    # Validate prop in one of three ways.

    # Fail with no notification.
    if not prop:
        return

    # Fail with an error message.
    if not prop:
        printerr("'prop' wasn't initialized")
        return

    # Fail and terminate.
    # NOTE: Scripts run from a release export template don't run `assert`s.
    assert(prop, "'prop' wasn't initialized")

# Use an autoload.
# Dangerous for typical nodes, but useful for true singleton nodes
# that manage their own data and don't interfere with other objects.
func reference_a_global_autoloaded_variable():
    print(globals)
    print(globals.prop)
    print(globals.my_getter())

```

C#

```

using Godot;
using System;
using System.Diagnostics;

public class MyNode : Node
{
    // Slow
    public void DynamicLookupWithDynamicNodePath()
    {
        GD.Print(GetNode("Child"));
    }

    // Fastest. Lookup node and cache for future access.
    // Doesn't break if node moves later.
    private Node _child;
    public void _Ready()
    {
        _child = GetNode("Child");
    }
    public void LookupAndCacheForFutureAccess()
    {
        GD.Print(_child);
    }
}

```

(continues on next page)

(continued from previous page)

```
// Delegate reference assignment to an external source.
// Con: need to perform a validation check.
// Pro: node makes no requirements of its external structure.
//      'prop' can come from anywhere.
public object Prop { get; set; }
public void CallMeAfterPropIsInitializedByParent()
{
    // Validate prop in one of three ways.

    // Fail with no notification.
    if (prop == null)
    {
        return;
    }

    // Fail with an error message.
    if (prop == null)
    {
        GD.PrintErr("'Prop' wasn't initialized");
        return;
    }

    // Fail with an exception.
    if (prop == null)
    {
        throw new InvalidOperationException("'Prop' wasn't initialized.");
    }

    // Fail and terminate.
    // Note: Scripts run from a release export template don't run `Debug.Assert`s.
    Debug.Assert(Prop, "'Prop' wasn't initialized");
}

// Use an autoload.
// Dangerous for typical nodes, but useful for true singleton nodes
// that manage their own data and don't interfere with other objects.
public void ReferenceAGlobalAutoloadedVariable()
{
    MyNode globals = GetNode<MyNode>("/root/Globals");
    GD.Print(globals);
    GD.Print(globals.Prop);
    GD.Print(globals.MyGetter());
}
};
```

Accessing data or logic from an object

Godot's scripting API is duck-typed. This means that if a script executes an operation, Godot doesn't validate that it supports the operation by type. It instead checks that the object implements the individual method.

For example, the `CanvasItem` class has a `visible` property. All properties exposed to the scripting API are in fact a setter and getter pair bound to a name. If one tried to access `CanvasItem.visible`, then Godot would do the following checks, in order:

- If the object has a script attached, it will attempt to set the property through the script. This leaves open the opportunity for scripts to override a property defined on a base object by overriding the setter method for the property.
- If the script does not have the property, it performs a `HashMap` lookup in the `ClassDB` for the "visible" property against the `CanvasItem` class and all of its inherited types. If found, it will call the bound setter or getter. For more information about `HashMap`s, see the data preferences docs.
- If not found, it does an explicit check to see if the user wants to access the "script" or "meta" properties.
- If not, it checks for a `_set/_get` implementation (depending on type of access) in the `CanvasItem` and its inherited types. These methods can execute logic that gives the impression that the Object has a property. This is also the case with the `_get_property_list` method.
 - Note that this happens even for non-legal symbol names, such as names starting with a digit or containing a slash.

As a result, this duck-typed system can locate a property either in the script, the object's class, or any class that object inherits, but only for things which extend `Object`.

Godot provides a variety of options for performing runtime checks on these accesses:

- A duck-typed property access. These will be property checks (as described above). If the operation isn't supported by the object, execution will halt.

GDScript

```
# All Objects have duck-typed get, set, and call wrapper methods.
get_parent().set("visible", false)

# Using a symbol accessor, rather than a string in the method call,
# will implicitly call the `set` method which, in turn, calls the
# setter method bound to the property through the property lookup
# sequence.
get_parent().visible = false

# Note that if one defines a _set and _get that describe a property's
# existence, but the property isn't recognized in any _get_property_list
# method, then the set() and get() methods will work, but the symbol
# access will claim it can't find the property.
```

C#

```
// All Objects have duck-typed Get, Set, and Call wrapper methods.
GetParent().Set("visible", false);

// C# is a static language, so it has no dynamic symbol access, e.g.
// `GetParent().Visible = false` won't work.
```

- A method check. In the case of `CanvasItem.visible`, one can access the methods, `set_visible` and `is_visible` like any other method.

GDScript

```
var child = get_child(0)

# Dynamic lookup.
child.call("set_visible", false)

# Symbol-based dynamic lookup.
# GDScript aliases this into a 'call' method behind the scenes.
child.set_visible(false)

# Dynamic lookup, checks for method existence first.
if child.has_method("set_visible"):
    child.set_visible(false)

# Cast check, followed by dynamic lookup.
# Useful when you make multiple "safe" calls knowing that the class
# implements them all. No need for repeated checks.
# Tricky if one executes a cast check for a user-defined type as it
# forces more dependencies.
if child is CanvasItem:
    child.set_visible(false)
    child.show_on_top = true

# If one does not wish to fail these checks without notifying users,
# one can use an assert instead. These will trigger runtime errors
# immediately if not true.
assert(child.has_method("set_visible"))
assert(child.is_in_group("offer"))
assert(child is CanvasItem)

# Can also use object labels to imply an interface, i.e. assume it
# implements certain methods.
# There are two types, both of which only exist for Nodes: Names and
# Groups.

# Assuming...
# A "Quest" object exists and 1) that it can "complete" or "fail" and
# that it will have text available before and after each state...

# 1. Use a name.
var quest = $Quest
print(quest.text)
quest.complete() # or quest.fail()
print(quest.text) # implied new text content

# 2. Use a group.
for a_child in get_children():
    if a_child.is_in_group("quest"):
        print(quest.text)
        quest.complete() # or quest.fail()
```

(continues on next page)

(continued from previous page)

```

    print(quest.text) # implied new text content

# Note that these interfaces are project-specific conventions the team
# defines (which means documentation! But maybe worth it?).
# Any script that conforms to the documented "interface" of the name or
# group can fill in for it.

```

C#

```

Node child = GetChild(0);

// Dynamic lookup.
child.Call("SetVisible", false);

// Dynamic lookup, checks for method existence first.
if (child.HasMethod("SetVisible"))
{
    child.Call("SetVisible", false);
}

// Use a group as if it were an "interface", i.e. assume it implements
// certain methods.
// Requires good documentation for the project to keep it reliable
// (unless you make editor tools to enforce it at editor time).
// Note, this is generally not as good as using an actual interface in
// C#, but you can't set C# interfaces from the editor since they are
// language-level features.
if (child.IsInGroup("Offer"))
{
    child.Call("Accept");
    child.Call("Reject");
}

// Cast check, followed by static lookup.
CanvasItem ci = GetParent() as CanvasItem;
if (ci != null)
{
    ci.SetVisible(false);

    // useful when you need to make multiple safe calls to the class
    ci.ShowOnTop = true;
}

// If one does not wish to fail these checks without notifying users,
// one can use an assert instead. These will trigger runtime errors
// immediately if not true.
Debug.Assert(child.HasMethod("set_visible"));
Debug.Assert(child.IsInGroup("offer"));
Debug.Assert(CanvasItem.InstanceHas(child));

// Can also use object labels to imply an interface, i.e. assume it
// implements certain methods.

```

(continues on next page)

(continued from previous page)

```
// There are two types, both of which only exist for Nodes: Names and
// Groups.

// Assuming...
// A "Quest" object exists and 1) that it can "Complete" or "Fail" and
// that it will have Text available before and after each state...

// 1. Use a name.
Node quest = GetNode("Quest");
GD.Print(quest.Get("Text"));
quest.Call("Complete"); // or "Fail".
GD.Print(quest.Get("Text")); // Implied new text content.

// 2. Use a group.
foreach (Node AChild in GetChildren())
{
    if (AChild.IsInGroup("quest"))
    {
        GD.Print(quest.Get("Text"));
        quest.Call("Complete"); // or "Fail".
        GD.Print(quest.Get("Text")); // Implied new text content.
    }
}

// Note that these interfaces are project-specific conventions the team
// defines (which means documentation! But maybe worth it?).
// Any script that conforms to the documented "interface" of the
// name or group can fill in for it. Also note that in C#, these methods
// will be slower than static accesses with traditional interfaces.
```

- Outsource the access to a Callable. These may be useful in cases where one needs the max level of freedom from dependencies. In this case, one relies on an external context to setup the method.

GDScript

```
# child.gd
extends Node
var fn = null

func my_method():
    if fn:
        fn.call()

# parent.gd
extends Node

@onready var child = $Child

func _ready():
    child.fn = print_me
    child.my_method()
```

(continues on next page)

(continued from previous page)

```
func print_me():
    print(name)
```

C#

```
// Child.cs
using Godot;

public partial class Child : Node
{
    public Callable? Callable { get; set; }

    public void MyMethod()
    {
        Callable?.Call();
    }
}

// Parent.cs
using Godot;

public partial class Parent : Node
{
    private Child _child;

    public void _Ready()
    {
        _child = GetNode<Child>("Child");
        _child.Callable = Callable.From(PrintMe);
        _child.MyMethod();
    }

    public void PrintMe()
    {
        GD.Print(Name);
    }
}
```

These strategies contribute to Godot's flexible design. Between them, users have a breadth of tools to meet their specific needs.

2.11.8 Godot notifications

Every Object in Godot implements a `_notification` method. Its purpose is to allow the Object to respond to a variety of engine-level callbacks that may relate to it. For example, if the engine tells a `CanvasItem` to "draw", it will call `_notification(NOTIFICATION_DRAW)`.

Some of these notifications, like draw, are useful to override in scripts. So much so that Godot exposes many of them with dedicated functions:

- `_ready()`: `NOTIFICATION_READY`
- `_enter_tree()`: `NOTIFICATION_ENTER_TREE`
- `_exit_tree()`: `NOTIFICATION_EXIT_TREE`

- `_process(delta)`: `NOTIFICATION_PROCESS`
- `_physics_process(delta)`: `NOTIFICATION_PHYSICS_PROCESS`
- `_draw()`: `NOTIFICATION_DRAW`

What users might not realize is that notifications exist for types other than `Node` alone, for example:

- `Object::NOTIFICATION_POSTINITIALIZE`: a callback that triggers during object initialization. Not accessible to scripts.
- `Object::NOTIFICATION_PREDELETE`: a callback that triggers before the engine deletes an `Object`, i.e. a "destructor".

And many of the callbacks that do exist in `Nodes` don't have any dedicated methods, but are still quite useful.

- `Node::NOTIFICATION_PARENTED`: a callback that triggers anytime one adds a child node to another node.
- `Node::NOTIFICATION_UNPARENTED`: a callback that triggers anytime one removes a child node from another node.

One can access all these custom notifications from the universal `_notification()` method.

Note: Methods in the documentation labeled as "virtual" are also intended to be overridden by scripts.

A classic example is the `_init` method in `Object`. While it has no `NOTIFICATION_*` equivalent, the engine still calls the method. Most languages (except `C#`) rely on it as a constructor.

So, in which situation should one use each of these notifications or virtual functions?

`_process` vs. `_physics_process` vs. `*_input`

Use `_process()` when one needs a framerate-dependent delta time between frames. If code that updates object data needs to update as often as possible, this is the right place. Recurring logic checks and data caching often execute here, but it comes down to the frequency at which one needs the evaluations to update. If they don't need to execute every frame, then implementing a `Timer`-timeout loop is another option.

GDScript

```
# Allows for recurring operations that don't trigger script logic
# every frame (or even every fixed frame).
func _ready():
    var timer = Timer.new()
    timer.autostart = true
    timer.wait_time = 0.5
    add_child(timer)
    timer.timeout.connect(func():
        print("This block runs every 0.5 seconds")
    )
```

C#

```
using Godot;

public partial class MyNode : Node
{
```

(continues on next page)

(continued from previous page)

```
// Allows for recurring operations that don't trigger script logic
// every frame (or even every fixed frame).
public override void _Ready()
{
    var timer = new Timer();
    timer.Autostart = true;
    timer.WaitTime = 0.5;
    AddChild(timer);
    timer.Timeout += () => GD.Print("This block runs every 0.5 seconds");
}
}
```

Use `_physics_process()` when one needs a framerate-independent delta time between frames. If code needs consistent updates over time, regardless of how fast or slow time advances, this is the right place. Recurring kinematic and object transform operations should execute here.

While it is possible, to achieve the best performance, one should avoid making input checks during these callbacks. `_process()` and `_physics_process()` will trigger at every opportunity (they do not "rest" by default). In contrast, `*_input()` callbacks will trigger only on frames in which the engine has actually detected the input.

One can check for input actions within the input callbacks just the same. If one wants to use delta time, one can fetch it from the related delta time methods as needed.

GDScript

```
# Called every frame, even when the engine detects no input.
func _process(delta):
    if Input.is_action_just_pressed("ui_select"):
        print(delta)

# Called during every input event.
func _unhandled_input(event):
    match event.get_class():
        "InputEventKey":
            if Input.is_action_just_pressed("ui_accept"):
                print(get_process_delta_time())
```

C#

```
using Godot;

public partial class MyNode : Node
{
    // Called every frame, even when the engine detects no input.
    public void _Process(double delta)
    {
        if (Input.IsActionJustPressed("ui_select"))
            GD.Print(delta);
    }

    // Called during every input event. Equally true for _input().
    public void _UnhandledInput(InputEvent @event)
```

(continues on next page)

(continued from previous page)

```

{
    switch (@event)
    {
        case InputEventKey:
            if (Input.IsActionJustPressed("ui_accept"))
                GD.Print(GetProcessDeltaTime());
            break;
    }
}

```

`_init` vs. initialization vs. `export`

If the script initializes its own node subtree, without a scene, that code should execute in `_init()`. Other property or SceneTree-independent initializations should also run here.

Note: The C# equivalent to GDScript's `_init()` method is the constructor.

`_init()` triggers before `_enter_tree()` or `_ready()`, but after a script creates and initializes its properties. When instantiating a scene, property values will set up according to the following sequence:

1. Initial value assignment: the property is assigned its initialization value, or its default value if one is not specified. If a setter exists, it is not used.
2. ```_init()``` assignment: the property's value is replaced by any assignments made in `_init()`, triggering the setter.
3. Exported value assignment: an exported property's value is again replaced by any value set in the Inspector, triggering the setter.

GDScript

```

# test is initialized to "one", without triggering the setter.
@export var test: String = "one":
    set(value):
        test = value + "!"

func _init():
    # Triggers the setter, changing test's value from "one" to "two!".
    test = "two"

# If someone sets test to "three" from the Inspector, it would trigger
# the setter, changing test's value from "two!" to "three!".

```

C#

```

using Godot;

public partial class MyNode : Node
{
    private string _test = "one";

```

(continues on next page)

(continued from previous page)

```

[Export]
public string Test
{
    get { return _test; }
    set { _test = $"{value}!"; }
}

public MyNode()
{
    // Triggers the setter, changing _test's value from "one" to "two!".
    Test = "two";
}

// If someone sets Test to "three" in the Inspector, it would trigger
// the setter, changing _test's value from "two!" to "three!".
}

```

As a result, instantiating a script versus a scene may affect both the initialization and the number of times the engine calls the setter.

`_ready` vs. `_enter_tree` vs. `NOTIFICATION_PARENTED`

When instantiating a scene connected to the first executed scene, Godot will instantiate nodes down the tree (making `_init()` calls) and build the tree going downwards from the root. This causes `_enter_tree()` calls to cascade down the tree. Once the tree is complete, leaf nodes call `_ready`. A node will call this method once all child nodes have finished calling theirs. This then causes a reverse cascade going up back to the tree's root.

When instantiating a script or a standalone scene, nodes are not added to the `SceneTree` upon creation, so no `_enter_tree()` callbacks trigger. Instead, only the `_init()` call occurs. When the scene is added to the `SceneTree`, the `_enter_tree()` and `_ready()` calls occur.

If one needs to trigger behavior that occurs as nodes parent to another, regardless of whether it occurs as part of the main/active scene or not, one can use the `PARENTED` notification. For example, here is a snippet that connects a node's method to a custom signal on the parent node without failing. Useful on data-centric nodes that one might create at runtime.

GDScript

```

extends Node

var parent_cache

func connection_check():
    return parent_cache.has_user_signal("interacted_with")

func _notification(what):
    match what:
        NOTIFICATION_PARENTED:
            parent_cache = get_parent()
            if connection_check():
                parent_cache.interacted_with.connect(_on_parent_interacted_with)
        NOTIFICATION_UNPARENTED:
            if connection_check():

```

(continues on next page)

(continued from previous page)

```
parent_cache.interacted_with.disconnect(_on_parent_interacted_with)

func _on_parent_interacted_with():
    print("I'm reacting to my parent's interaction!")
```

C#

```
using Godot;

public partial class MyNode : Node
{
    private Node _parentCache;

    public void ConnectionCheck()
    {
        return _parentCache.HasUserSignal("InteractedWith");
    }

    public void _Notification(int what)
    {
        switch (what)
        {
            case NotificationParented:
                _parentCache = GetParent();
                if (ConnectionCheck())
                {
                    _parentCache.Connect("InteractedWith", Callable.From(OnParentInteractedWith));
                }
                break;
            case NotificationUnparented:
                if (ConnectionCheck())
                {
                    _parentCache.Disconnect("InteractedWith", Callable.From(OnParentInteractedWith));
                }
                break;
        }
    }

    private void OnParentInteractedWith()
    {
        GD.Print("I'm reacting to my parent's interaction!");
    }
}
```

2.11.9 Data preferences

Ever wondered whether one should approach problem X with data structure Y or Z? This article covers a variety of topics related to these dilemmas.

Note: This article makes references to "[something]-time" operations. This terminology comes from algorithm analysis' [Big O Notation](#).

Long-story short, it describes the worst-case scenario of runtime length. In laymen's terms:

"As the size of a problem domain increases, the runtime length of the algorithm..."

- Constant-time, $O(1)$: "...does not increase."
- Logarithmic-time, $O(\log n)$: "...increases at a slow rate."
- Linear-time, $O(n)$: "...increases at the same rate."
- Etc.

Imagine if one had to process 3 million data points within a single frame. It would be impossible to craft the feature with a linear-time algorithm since the sheer size of the data would increase the runtime far beyond the time allotted. In comparison, using a constant-time algorithm could handle the operation without issue.

By and large, developers want to avoid engaging in linear-time operations as much as possible. But, if one keeps the scale of a linear-time operation small, and if one does not need to perform the operation often, then it may be acceptable. Balancing these requirements and choosing the right algorithm / data structure for the job is part of what makes programmers' skills valuable.

Array vs. Dictionary vs. Object

Godot stores all variables in the scripting API in the Variant class. Variants can store Variant-compatible data structures such as Array and Dictionary as well as Objects.

Godot implements Array as a `Vector<Variant>`. The engine stores the Array contents in a contiguous section of memory, i.e. they are in a row adjacent to each other.

Note: For those unfamiliar with C++, a Vector is the name of the array object in traditional C++ libraries. It is a "templated" type, meaning that its records can only contain a particular type (denoted by angled brackets). So, for example, a `PackedStringArray` would be something like a `Vector<String>`.

Contiguous memory stores imply the following operation performance:

- Iterate: Fastest. Great for loops.
 - Op: All it does is increment a counter to get to the next record.
- Insert, Erase, Move: Position-dependent. Generally slow.
 - Op: Adding/removing/moving content involves moving the adjacent records over (to make room / fill space).
 - Fast add/remove from the end.
 - Slow add/remove from an arbitrary position.
 - Slowest add/remove from the front.
 - If doing many inserts/removals from the front, then...
 1. invert the array.

2. do a loop which executes the Array changes at the end.
3. re-invert the array.

This makes only 2 copies of the array (still constant time, but slow) versus copying roughly 1/2 of the array, on average, N times (linear time).

- Get, Set: Fastest by position. E.g. can request 0th, 2nd, 10th record, etc. but cannot specify which record you want.
 - Op: 1 addition operation from array start position up to desired index.
- Find: Slowest. Identifies the index/position of a value.
 - Op: Must iterate through array and compare values until one finds a match.
 - * Performance is also dependent on whether one needs an exhaustive search.
 - If kept ordered, custom search operations can bring it to logarithmic time (relatively fast). Laymen users won't be comfortable with this though. Done by re-sorting the Array after every edit and writing an ordered-aware search algorithm.

Godot implements Dictionary as an `OrderedHashMap<Variant, Variant>`. The engine stores a small array (initialized to 2^3 or 8 records) of key-value pairs. When one attempts to access a value, they provide it a key. It then hashes the key, i.e. converts it into a number. The "hash" is used to calculate the index into the array. As an array, the OHM then has a quick lookup within the "table" of keys mapped to values. When the HashMap becomes too full, it increases to the next power of 2 (so, 16 records, then 32, etc.) and rebuilds the structure.

Hashes are to reduce the chance of a key collision. If one occurs, the table must recalculate another index for the value that takes the previous position into account. In all, this results in constant-time access to all records at the expense of memory and some minor operational efficiency.

1. Hashing every key an arbitrary number of times.
 - Hash operations are constant-time, so even if an algorithm must do more than one, as long as the number of hash calculations doesn't become too dependent on the density of the table, things will stay fast. Which leads to...
2. Maintaining an ever-growing size for the table.
 - HashMaps maintain gaps of unused memory interspersed in the table on purpose to reduce hash collisions and maintain the speed of accesses. This is why it constantly increases in size quadratically by powers of 2.

As one might be able to tell, Dictionaries specialize in tasks that Arrays do not. An overview of their operational details is as follows:

- Iterate: Fast.
 - Op: Iterate over the map's internal vector of hashes. Return each key. Afterwards, users then use the key to jump to and return the desired value.
- Insert, Erase, Move: Fastest.
 - Op: Hash the given key. Do 1 addition operation to look up the appropriate value (array start + offset). Move is two of these (one insert, one erase). The map must do some maintenance to preserve its capabilities:
 - * update ordered List of records.
 - * determine if table density mandates a need to expand table capacity.
 - The Dictionary remembers in what order users inserted its keys. This enables it to execute reliable iterations.

- Get, Set: Fastest. Same as a lookup by key.
 - Op: Same as insert/erase/move.
- Find: Slowest. Identifies the key of a value.
 - Op: Must iterate through records and compare the value until a match is found.
 - Note that Godot does not provide this feature out-of-the-box (because they aren't meant for this task).

Godot implements Objects as stupid, but dynamic containers of data content. Objects query data sources when posed questions. For example, to answer the question, "do you have a property called, 'position'?", it might ask its script or the ClassDB. One can find more information about what objects are and how they work in the Applying object-oriented principles in Godot article.

The important detail here is the complexity of the Object's task. Every time it performs one of these multi-source queries, it runs through several iteration loops and HashMap lookups. What's more, the queries are linear-time operations dependent on the Object's inheritance hierarchy size. If the class the Object queries (its current class) doesn't find anything, the request defers to the next base class, all the way up until the original Object class. While these are each fast operations in isolation, the fact that it must make so many checks is what makes them slower than both of the alternatives for looking up data.

Note: When developers mention how slow the scripting API is, it is this chain of queries they refer to. Compared to compiled C++ code where the application knows exactly where to go to find anything, it is inevitable that scripting API operations will take much longer. They must locate the source of any relevant data before they can attempt to access it.

The reason GDScript is slow is because every operation it performs passes through this system.

C# can process some content at higher speeds via more optimized bytecode. But, if the C# script calls into an engine class' content or if the script tries to access something external to it, it will go through this pipeline.

NativeScript C++ goes even further and keeps everything internal by default. Calls into external structures will go through the scripting API. In NativeScript C++, registering methods to expose them to the scripting API is a manual task. It is at this point that external, non-C++ classes will use the API to locate them.

So, assuming one extends from Reference to create a data structure, like an Array or Dictionary, why choose an Object over the other two options?

1. Control: With objects comes the ability to create more sophisticated structures. One can layer abstractions over the data to ensure the external API doesn't change in response to internal data structure changes. What's more, Objects can have signals, allowing for reactive behavior.
2. Clarity: Objects are a reliable data source when it comes to the data that scripts and engine classes define for them. Properties may not hold the values one expects, but one doesn't need to worry about whether the property exists in the first place.
3. Convenience: If one already has a similar data structure in mind, then extending from an existing class makes the task of building the data structure much easier. In comparison, Arrays and Dictionaries don't fulfill all use cases one might have.

Objects also give users the opportunity to create even more specialized data structures. With it, one can design their own List, Binary Search Tree, Heap, Splay Tree, Graph, Disjoint Set, and any host of other options.

"Why not use Node for tree structures?" one might ask. Well, the Node class contains things that won't be relevant to one's custom data structure. As such, it can be helpful to construct one's own node type when building tree structures.

GDScript

```
extends Object
class_name TreeNode

var _parent: TreeNode = null
var _children: = [] setget

func _notification(p_what):
    match p_what:
        NOTIFICATION_PREDELETE:
            # Destructor.
            for a_child in _children:
                a_child.free()
```

C#

```
using Godot;
using System.Collections.Generic;

// Can decide whether to expose getters/setters for properties later
public partial class TreeNode : GodotObject
{
    private TreeNode _parent = null;

    private List<TreeNode> _children = new();

    public override void _Notification(int what)
    {
        switch (what)
        {
            case NotificationPredelete:
                foreach (TreeNode child in _children)
                {
                    node.Free();
                }
                break;
        }
    }
}
```

From here, one can then create their own structures with specific features, limited only by their imagination.

Enumerations: int vs. string

Most languages offer an enumeration type option. GDScript is no different, but unlike most other languages, it allows one to use either integers or strings for the enum values (the latter only when using the `export` keyword in GDScript). The question then arises, "which should one use?"

The short answer is, "whichever you are more comfortable with." This is a feature specific to GDScript and not Godot scripting in general; The languages prioritizes usability over performance.

On a technical level, integer comparisons (constant-time) will happen faster than string comparisons (linear-time). If one wants to keep up other languages' conventions though, then one should use integers.

The primary issue with using integers comes up when one wants to print an enum value. As integers,

attempting to print `MY_ENUM` will print 5 or what-have-you, rather than something like "MyEnum". To print an integer enum, one would have to write a Dictionary that maps the corresponding string value for each enum.

If the primary purpose of using an enum is for printing values and one wishes to group them together as related concepts, then it makes sense to use them as strings. That way, a separate data structure to execute on the printing is unnecessary.

AnimatedTexture vs. AnimatedSprite2D vs. AnimationPlayer vs. AnimationTree

Under what circumstances should one use each of Godot's animation classes? The answer may not be immediately clear to new Godot users.

`AnimatedTexture` is a texture that the engine draws as an animated loop rather than a static image. Users can manipulate...

1. the rate at which it moves across each section of the texture (FPS).
2. the number of regions contained within the texture (frames).

Godot's `RenderingServer` then draws the regions in sequence at the prescribed rate. The good news is that this involves no extra logic on the part of the engine. The bad news is that users have very little control.

Also note that `AnimatedTexture` is a `Resource` unlike the other `Node` objects discussed here. One might create a `Sprite2D` node that uses `AnimatedTexture` as its texture. Or (something the others can't do) one could add `AnimatedTextures` as tiles in a `TileSet` and integrate it with a `TileMap` for many auto-animating backgrounds that all render in a single batched draw call.

The `AnimatedSprite2D` node, in combination with the `SpriteFrames` resource, allows one to create a variety of animation sequences through spritesheets, flip between animations, and control their speed, regional offset, and orientation. This makes them well-suited to controlling 2D frame-based animations.

If one needs trigger other effects in relation to animation changes (for example, create particle effects, call functions, or manipulate other peripheral elements besides the frame-based animation), then will need to use an `AnimationPlayer` node in conjunction with the `AnimatedSprite2D`.

`AnimationPlayers` are also the tool one will need to use if they wish to design more complex 2D animation systems, such as...

1. Cut-out animations: editing sprites' transforms at runtime.
2. 2D Mesh animations: defining a region for the sprite's texture and rigging a skeleton to it. Then one animates the bones which stretch and bend the texture in proportion to the bones' relationships to each other.
3. A mix of the above.

While one needs an `AnimationPlayer` to design each of the individual animation sequences for a game, it can also be useful to combine animations for blending, i.e. enabling smooth transitions between these animations. There may also be a hierarchical structure between animations that one plans out for their object. These are the cases where the `AnimationTree` shines. One can find an in-depth guide on using the `AnimationTree` [here](#).

2.11.10 Logic preferences

Ever wondered whether one should approach problem X with strategy Y or Z? This article covers a variety of topics related to these dilemmas.

Adding nodes and changing properties: which first?

When initializing nodes from a script at runtime, you may need to change properties such as the node's name or position. A common dilemma is, when should you change those values?

It is the best practice to change values on a node before adding it to the scene tree. Some property's setters have code to update other corresponding values, and that code can be slow! For most cases, this code has no impact on your game's performance, but in heavy use cases such as procedural generation, it can bring your game to a crawl.

For these reasons, it is always a best practice to set the initial values of a node before adding it to the scene tree.

Loading vs. preloading

In GDScript, there exists the global preload method. It loads resources as early as possible to front-load the "loading" operations and avoid loading resources while in the middle of performance-sensitive code.

Its counterpart, the load method, loads a resource only when it reaches the load statement. That is, it will load a resource in-place which can cause slowdowns when it occurs in the middle of sensitive processes. The load() function is also an alias for ResourceLoader.load(path) which is accessible to all scripting languages.

So, when exactly does preloading occur versus loading, and when should one use either? Let's see an example:

GDScript

```
# my_buildings.gd
extends Node

# Note how constant scripts/scenes have a different naming scheme than
# their property variants.

# This value is a constant, so it spawns when the Script object loads.
# The script is preloading the value. The advantage here is that the editor
# can offer autocompletion since it must be a static path.
const BuildingScn = preload("res://building.tscn")

# 1. The script preloads the value, so it will load as a dependency
#    of the 'my_buildings.gd' script file. But, because this is a
#    property rather than a constant, the object won't copy the preloaded
#    PackedScene resource into the property until the script instantiates
#    with .new().
#
# 2. The preloaded value is inaccessible from the Script object alone. As
#    such, preloading the value here actually does not benefit anyone.
#
# 3. Because the user exports the value, if this script stored on
#    a node in a scene file, the scene instantiation code will overwrite the
#    preloaded initial value anyway (wasting it). It's usually better to
#    provide null, empty, or otherwise invalid default values for exports.
#
# 4. It is when one instantiates this script on its own with .new() that
```

(continues on next page)

(continued from previous page)

```
# one will load "office.tscn" rather than the exported value.
export(PackedScene) var a_building = preload("office.tscn")

# Uh oh! This results in an error!
# One must assign constant values to constants. Because `load` performs a
# runtime lookup by its very nature, one cannot use it to initialize a
# constant.
const OfficeScn = load("res://office.tscn")

# Successfully loads and only when one instantiates the script! Yay!
var office_scn = load("res://office.tscn")
```

C#

```
using Godot;

// C# and other languages have no concept of "preloading".
public partial class MyBuildings : Node
{
    //This is a read-only field, it can only be assigned when it's declared or during a constructor.
    public readonly PackedScene Building = ResourceLoader.Load<PackedScene>("res://building.tscn");

    public PackedScene ABuilding;

    public override void _Ready()
    {
        // Can assign the value during initialization.
        ABuilding = GD.Load<PackedScene>("res://Office.tscn");
    }
}
```

Preloading allows the script to handle all the loading the moment one loads the script. Preloading is useful, but there are also times when one doesn't wish for it. To distinguish these situations, there are a few things one can consider:

1. If one cannot determine when the script might load, then preloading a resource, especially a scene or script, could result in further loads one does not expect. This could lead to unintentional, variable-length load times on top of the original script's load operations.
2. If something else could replace the value (like a scene's exported initialization), then preloading the value has no meaning. This point isn't a significant factor if one intends to always create the script on its own.
3. If one wishes only to 'import' another class resource (script or scene), then using a preloaded constant is often the best course of action. However, in exceptional cases, one may wish not to do this:
 1. If the 'imported' class is liable to change, then it should be a property instead, initialized either using an export or a load() (and perhaps not even initialized until later).
 2. If the script requires a great many dependencies, and one does not wish to consume so much memory, then one may wish to, load and unload various dependencies at runtime as circumstances change. If one preloads resources into constants, then the only way to unload these resources would be to unload the entire script. If they are instead loaded properties, then one can set them to null and remove all references to the resource entirely (which, as a RefCounted-extending type, will cause the resources to delete themselves from memory).

Large levels: static vs. dynamic

If one is creating a large level, which circumstances are most appropriate? Should they create the level as one static space? Or should they load the level in pieces and shift the world's content as needed?

Well, the simple answer is, "when the performance requires it." The dilemma associated with the two options is one of the age-old programming choices: does one optimize memory over speed, or vice versa?

The naive answer is to use a static level that loads everything at once. But, depending on the project, this could consume a large amount of memory. Wasting users' RAM leads to programs running slow or outright crashing from everything else the computer tries to do at the same time.

No matter what, one should break larger scenes into smaller ones (to aid in reusability of assets). Developers can then design a node that manages the creation/loading and deletion/unloading of resources and nodes in real-time. Games with large and varied environments or procedurally generated elements often implement these strategies to avoid wasting memory.

On the flip side, coding a dynamic system is more complex, i.e. uses more programmed logic, which results in opportunities for errors and bugs. If one isn't careful, they can develop a system that bloats the technical debt of the application.

As such, the best options would be...

1. To use a static level for smaller games.
2. If one has the time/resources on a medium/large game, create a library or plugin that can code the management of nodes and resources. If refined over time, so as to improve usability and stability, then it could evolve into a reliable tool across projects.
3. Code the dynamic logic for a medium/large game because one has the coding skills, but not the time or resources to refine the code (game's gotta get done). Could potentially refactor later to outsource the code into a plugin.

For an example of the various ways one can swap scenes around at runtime, please see the "Change scenes manually" documentation.

2.11.11 Project organization

Introduction

Since Godot has no restrictions on project structure or filesystem usage, organizing files when learning the engine can seem challenging. This tutorial suggests a workflow which should be a good starting point. We will also cover using version control with Godot.

Organization

Godot is scene-based in nature, and uses the filesystem as-is, without metadata or an asset database.

Unlike other engines, many resources are contained within the scene itself, so the amount of files in the filesystem is considerably lower.

Considering that, the most common approach is to group assets as close to scenes as possible; when a project grows, it makes it more maintainable.

As an example, one can usually place into a single folder their basic assets, such as sprite images, 3D model meshes, materials, and music, etc. They can then use a separate folder to store built levels that use them.

```
/project.godot
/docs/.gdignore # See "Ignoring specific folders" below
/docs/learning.html
```

(continues on next page)

(continued from previous page)

```
/models/town/house/house.dae
/models/town/house/window.png
/models/town/house/door.png
/characters/player/cubio.dae
/characters/player/cubio.png
/characters/enemies/goblin/goblin.dae
/characters/enemies/goblin/goblin.png
/characters/npcs/suzanne/suzanne.dae
/characters/npcs/suzanne/suzanne.png
/levels/riverdale/riverdale.scn
```

Style guide

For consistency across projects, we recommend following these guidelines:

- Use snake_case for folder and file names (with the exception of C# scripts). This sidesteps case sensitivity issues that can crop up after exporting a project on Windows. C# scripts are an exception to this rule, as the convention is to name them after the class name which should be in PascalCase.
- Use PascalCase for node names, as this matches built-in node casing.
- In general, keep third-party resources in a top-level addons/ folder, even if they aren't editor plugins. This makes it easier to track which files are third-party. There are some exceptions to this rule; for instance, if you use third-party game assets for a character, it makes more sense to include them within the same folder as the character scenes and scripts.

Importing

Godot versions prior to 3.0 did the import process from files outside the project. While this can be useful in large projects, it resulted in an organization hassle for most developers.

Because of this, assets are now transparently imported from within the project folder.

Ignoring specific folders

To prevent Godot from importing files contained in a specific folder, create an empty file called .gdignore in the folder (the leading . is required). This can be useful to speed up the initial project importing.

Note: To create a file whose name starts with a dot on Windows, place a dot at both the beginning and end of the filename (".gdignore."). Windows will automatically remove the trailing dot when you confirm the name.

Alternatively, you can use a text editor such as Notepad++ or use the following command in a command prompt: `type nul > .gdignore`

Once the folder is ignored, resources in that folder can't be loaded anymore using the `load()` and `preload()` methods. Ignoring a folder will also automatically hide it from the FileSystem dock, which can be useful to reduce clutter.

Note that the .gdignore file's contents are ignored, which is why the file should be empty. It does not support patterns like .gitignore files do.

Case sensitivity

Windows and recent macOS versions use case-insensitive filesystems by default, whereas Linux distributions use a case-sensitive filesystem by default. This can cause issues after exporting a project, since Godot's PCK virtual filesystem is case-sensitive. To avoid this, it's recommended to stick to snake_case naming for all files in the project (and lowercase characters in general).

Note: You can break this rule when style guides say otherwise (such as the C# style guide). Still, be consistent to avoid mistakes.

On Windows 10, to further avoid mistakes related to case sensitivity, you can also make the project folder case-sensitive. After enabling the Windows Subsystem for Linux feature, run the following command in a PowerShell window:

```
# To enable case-sensitivity:
fsutil file setcasesensitiveinfo <path to project folder> enable

# To disable case-sensitivity:
fsutil file setcasesensitiveinfo <path to project folder> disable
```

If you haven't enabled the Windows Subsystem for Linux, you can enter the following line in a PowerShell window running as Administrator then reboot when asked:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

2.11.12 Version control systems

Introduction

Godot aims to be VCS-friendly and generate mostly readable and mergeable files.

Version control plugins

Godot also supports the use of version control systems in the editor itself. However, version control in the editor requires a plugin for the specific VCS you're using.

As of July 2023, there is only a Git plugin available, but the community may create additional VCS plugins.

Official Git plugin

Warning: As of July 2023, the Git plugin hasn't been updated to work with Godot 4.1 and later yet.

Using Git from inside the editor is supported with an official plugin. You can find the latest releases on [GitHub](#).

Documentation on how to use the Git plugin can be found on its [wiki](#).

Files to exclude from VCS

Note: This lists files and folders that should be ignored from version control in Godot 4.1 and later.

The list of files of folders that should be ignored from version control in Godot 3.x and Godot 4.0 is entirely different. This is important, as Godot 3.x and 4.0 may store sensitive credentials in `export_presets.cfg` (unlike Godot 4.1 and later).

If you are using Godot 3, check the 3.5 version of this documentation page instead.

There are some files and folders Godot automatically creates when opening a project in the editor for the first time. To avoid bloating your version control repository with generated data, you should add them to your VCS ignore:

- `.godot/`: This folder stores various project cache data.
- `*.translation`: These files are binary imported translations generated from CSV files.

You can make the Godot project manager generate version control metadata for you automatically when creating a project. When choosing the Git option, this creates `.gitignore` and `.gitattributes` files in the project root:

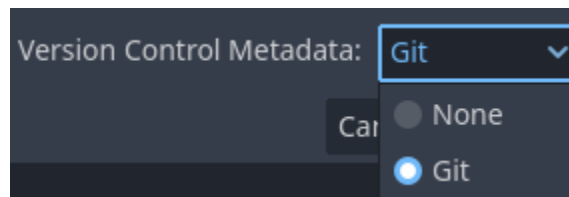


Fig. 1: Creating version control metadata in the project manager's New Project dialog

In existing projects, select the Project menu at the top of the editor, then choose Version Control > Generate Version Control Metadata. This creates the same files as if the operation was performed in the project manager.

Working with Git on Windows

Most Git for Windows clients are configured with the `core.autocrlf` set to `true`. This can lead to files unnecessarily being marked as modified by Git due to their line endings being converted from LF to CRLF automatically.

It is better to set this option as:

```
git config --global core.autocrlf input
```

Creating version control metadata using the project manager or editor will automatically enforce LF line endings using the `.gitattributes` file. In this case, you don't need to change your Git configuration.

2.12 Troubleshooting

This page lists common issues encountered when using Godot and possible solutions.

See also:

See Using the Web editor for caveats specific to the Web version of the Godot editor.

2.12.1 The editor runs slowly and uses all my CPU and GPU resources, making my computer noisy

This is a known issue, especially on macOS since most Macs have Retina displays. Due to Retina displays' higher pixel density, everything has to be rendered at a higher resolution. This increases the load on the GPU and decreases perceived performance.

There are several ways to improve performance and battery life:

- In 3D, click the Perspective button in the top left corner and enable Half Resolution. The 3D viewport will now be rendered at half resolution, which can be up to 4 times faster.
- Open the Editor Settings and increase the value of Low Processor Mode Sleep (μsec) to 33000 (30 FPS). This value determines the amount of microseconds between frames to render. Higher values will make the editor feel less reactive but will help decrease CPU and GPU usage significantly.
- If you have a node that causes the editor to redraw continuously (such as particles), hide it and show it using a script in the `_ready()` method. This way, it will be hidden in the editor but will still be visible in the running project.

2.12.2 The editor stutters and flickers on my variable refresh rate monitor (G-Sync/FreeSync)

This is a known issue. Variable refresh rate monitors need to adjust their gamma curves continuously to emit a consistent amount of light over time. This can cause flicker to appear in dark areas of the image when the refresh rate varies a lot, which occurs as the Godot editor only redraws when necessary.

There are several workarounds for this:

- Enable Interface > Editor > Update Continuously in the Editor Settings. Keep in mind this will increase power usage and heat/noise emissions since the editor will now be rendering constantly, even if nothing has changed on screen. To alleviate this, you can increase Low Processor Mode Sleep (μsec) to 33000 (30 FPS) in the Editor Settings. This value determines the amount of microseconds between frames to render. Higher values will make the editor feel less reactive but will help decrease CPU and GPU usage significantly.
- Alternatively, disable variable refresh rate on your monitor or in the graphics driver.
- VRR flicker can be reduced on some displays using the VRR Control or Fine Tune Dark Areas options in your monitor's OSD. These options may increase input lag or result in crushed blacks.
- If using an OLED display, use the Black (OLED) editor theme preset in the Editor Settings. This hides VRR flicker thanks to OLED's perfect black levels.

2.12.3 The editor or project takes a very long time to start

When using one of the Vulkan-based renderers (Forward+ or Forward Mobile), the first startup is expected to be relatively long. This is because shaders need to be compiled before they can be cached. Shaders also need to be cached again after updating Godot, after updating graphics drivers or after switching graphics cards.

If the issue persists after the first startup, this is a [known bug](#) on Windows when you have specific USB peripherals connected. In particular, Corsair's iCUE software seems to cause this bug. Try updating your USB peripherals' drivers to their latest version. If the bug persists, you need to disconnect the specific peripheral before opening the editor. You can then connect the peripheral again.

Firewall software such as Portmaster may also cause the debug port to be blocked. This causes the project to take a long time to start, while being unable to use debugging features in the editor (such as viewing `print()` output). You can work this around by changing the debug port used by the project in the Editor Settings (Network > Debug > Remote Port). The default is 6007; try another value that is greater than 1024, such as 7007.

2.12.4 The Godot editor appears frozen after clicking the system console

When running Godot on Windows with the system console enabled, you can accidentally enable selection mode by clicking inside the command window. This Windows-specific behavior pauses the application to let you select text inside the system console. Godot cannot override this system-specific behavior.

To solve this, select the system console window and press Enter to leave selection mode.

2.12.5 The Godot editor's macOS dock icon gets duplicated every time it is manually moved

If you open the Godot editor and manually change the position of the dock icon, then restart the editor, you will get a duplicate dock icon all the way to the right of the dock.

This is due to a design limitation of the macOS dock. The only known way to resolve this would be to merge the project manager and editor into a single process, which means the project manager would no longer spawn a separate process when starting the editor. While using a single process instance would bring several benefits, it isn't planned to be done in the near future due to the complexity of the task.

To avoid this issue, keep the Godot editor's dock icon at its default location as created by macOS.

2.12.6 Some text such as "NO DC" appears in the top-left corner of the Project Manager and editor window

This is caused by the NVIDIA graphics driver injecting an overlay to display information.

To disable this overlay on Windows, restore your graphics driver settings to the default values in the NVIDIA Control Panel.

To disable this overlay on Linux, open `nvidia-settings`, go to X Screen 0 > OpenGL Settings then uncheck Enable Graphics API Visual Indicator.

2.12.7 The editor or project appears overly sharp or blurry

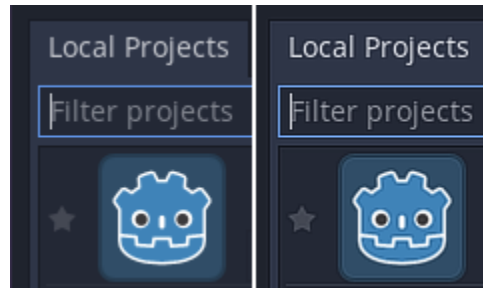


Fig. 2: Correct appearance (left), oversharpened appearance due to graphics driver sharpening (right)

If the editor or project appears overly sharp, this is likely due to image sharpening being forced on all Vulkan or OpenGL applications by your graphics driver. You can disable this behavior in the graphics driver's control panel:

- **NVIDIA (Windows):** Open the start menu and choose NVIDIA Control Panel. Open the Manage 3D settings tab on the left. In the list in the middle, scroll to Image Sharpening and set it to Sharpening Off.
- **AMD (Windows):** Open the start menu and choose AMD Software. Click the settings "cog" icon in the top-right corner. Go to the Graphics tab then disable Radeon Image Sharpening.

If the editor or project appears overly blurry, this is likely due to FXAA (Fast Approximate AntiAliasing) being forced on all Vulkan or OpenGL applications by your graphics driver.

- **NVIDIA (Windows):** Open the start menu and choose NVIDIA Control Panel. Open the Manage 3D settings tab on the left. In the list in the middle, scroll to Fast Approximate Antialiasing and set it to Application Controlled.
- **NVIDIA (Linux):** Open the applications menu and choose NVIDIA X Server Settings. Select to Antialiasing Settings on the left, then uncheck Enable FXAA.
- **AMD (Windows):** Open the start menu and choose AMD Software. Click the settings "cog" icon in the top-right corner. Go to the Graphics tab, scroll to the bottom and click Advanced to unfold its settings. Disable Morphological Anti-Aliasing.

Third-party vendor-independent utilities such as vkBasalt may also force sharpening or FXAA on all Vulkan applications. You may want to check their configuration as well.

After changing options in the graphics driver or third-party utilities, restart Godot to make the changes effective.

If you still wish to force sharpening or FXAA on other applications, it's recommended to do so on a per-application basis using the application profiles system provided by graphics drivers' control panels.

2.12.8 The editor or project appears to have washed out colors

On Windows, this is usually caused by incorrect OS or monitor settings, as Godot currently does not support HDR (High Dynamic Range) output (even though it may internally render in HDR).

As [most displays are not designed to display SDR content in HDR mode](#), it is recommended to disable HDR in the Windows settings when not running applications that use HDR output. On Windows 11, this can be done by pressing Windows + Alt + B (this shortcut is part of the Xbox Game Bar app). To toggle HDR automatically based on applications currently running, you can use [AutoActions](#).

If you insist on leaving HDR enabled, it is possible to somewhat improve the result by ensuring the display is configured to use HGIG (HDR Gaming Interest Group) tonemapping (as opposed to DTM (Dynamic Tone

Mapping)), then using the [Windows HDR calibration app](#). It is also strongly recommended to use Windows 11 instead of Windows 10 when using HDR. The end result will still likely be inferior to disabling HDR on the display, though.

Support for HDR output is planned in a future release.

2.12.9 The editor/project freezes or displays glitched visuals after resuming the PC from suspend

This is a known issue on Linux with NVIDIA graphics when using the proprietary driver. There is no definitive fix yet, as suspend on Linux + NVIDIA is often buggy when OpenGL or Vulkan is involved. The Compatibility rendering method (which uses OpenGL) is generally less prone to suspend-related issues compared to the Forward+ and Forward Mobile rendering methods (which use Vulkan).

The NVIDIA driver offers an experimental [option to preserve video memory after suspend](#) which may resolve this issue. This option has been reported to work better with more recent NVIDIA driver versions.

To avoid losing work, save scenes in the editor before putting the PC to sleep.

2.12.10 The project works when run from the editor, but fails to load some files when running from an exported copy

This is usually caused by forgetting to specify a filter for non-resource files in the Export dialog. By default, Godot will only include actual resources into the PCK file. Some files commonly used, such as JSON files, are not considered resources. For example, if you load test.json in the exported project, you need to specify *.json in the non-resource export filter. See [Resource options](#) for more information.

Also, note that files and folders whose names begin with a period will never be included in the exported project. This is done to prevent version control folders like .git from being included in the exported PCK file.

On Windows, this can also be due to case sensitivity issues. If you reference a resource in your script with a different case than on the filesystem, loading will fail once you export the project. This is because the virtual PCK filesystem is case-sensitive, while Windows's filesystem is case-insensitive by default.

2.13 Editor introduction

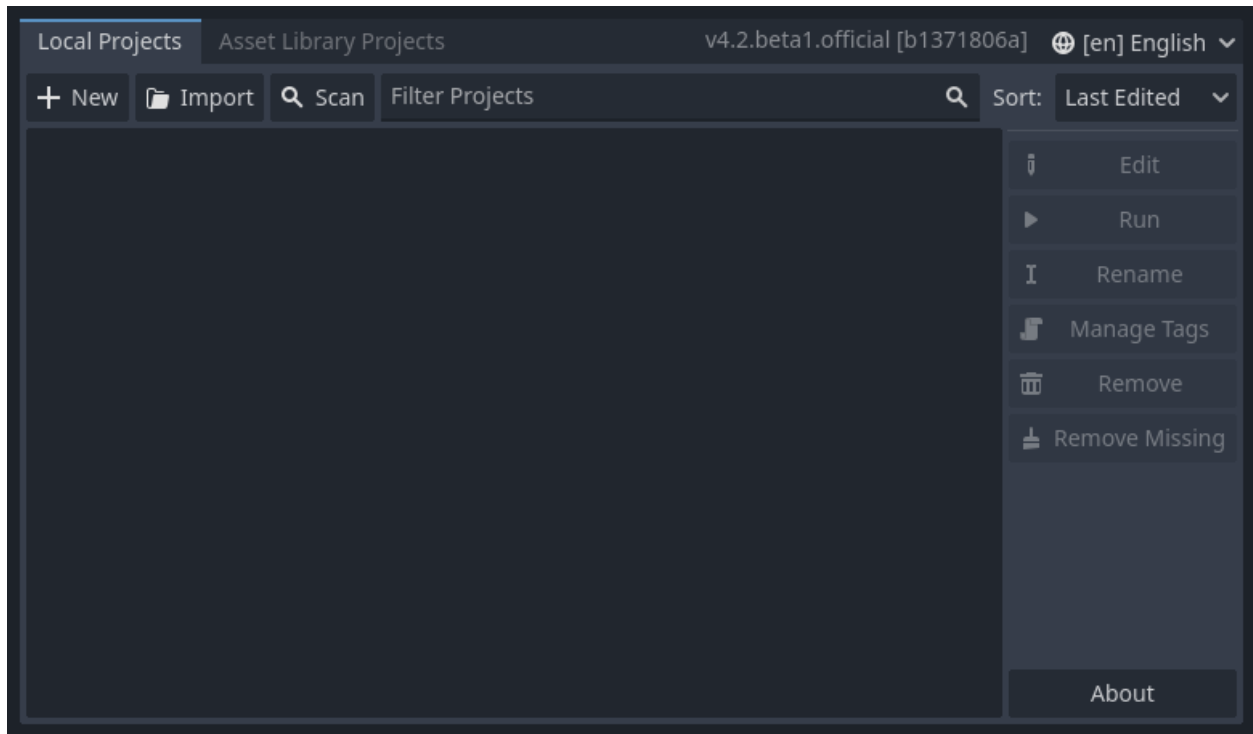
In this section, we cover the Godot editor in general, from its interface to using it with the command line.

2.13.1 Editor's interface

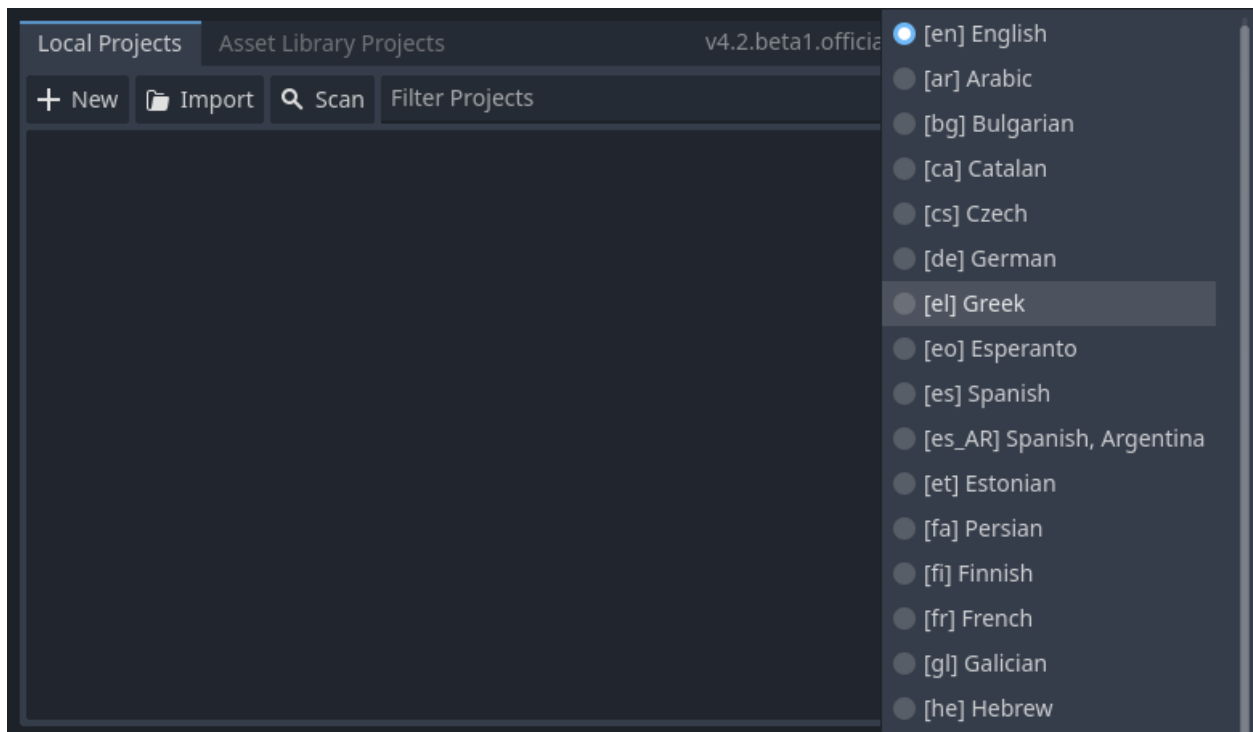
The following pages explain how to use the various windows, workspaces, and docks that make up the Godot editor. We cover some specific editors' interface in other sections where appropriate. For example, the animation editor.

Using the Project Manager

When you launch Godot, the first window you see is the Project Manager. It lets you create, remove, import, or play game projects.



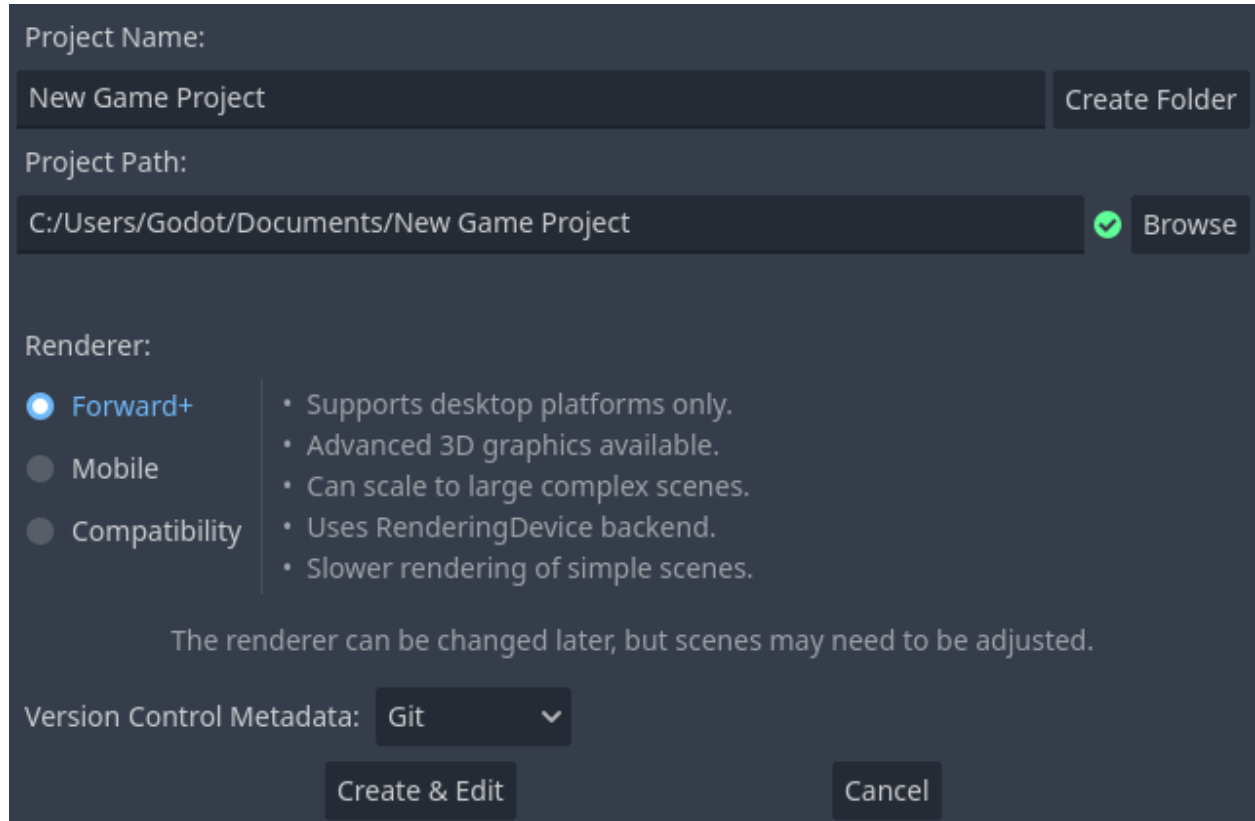
In the window's top-right corner, a drop-down menu allows you to change the editor's language.



Creating and importing projects

To create a new project:

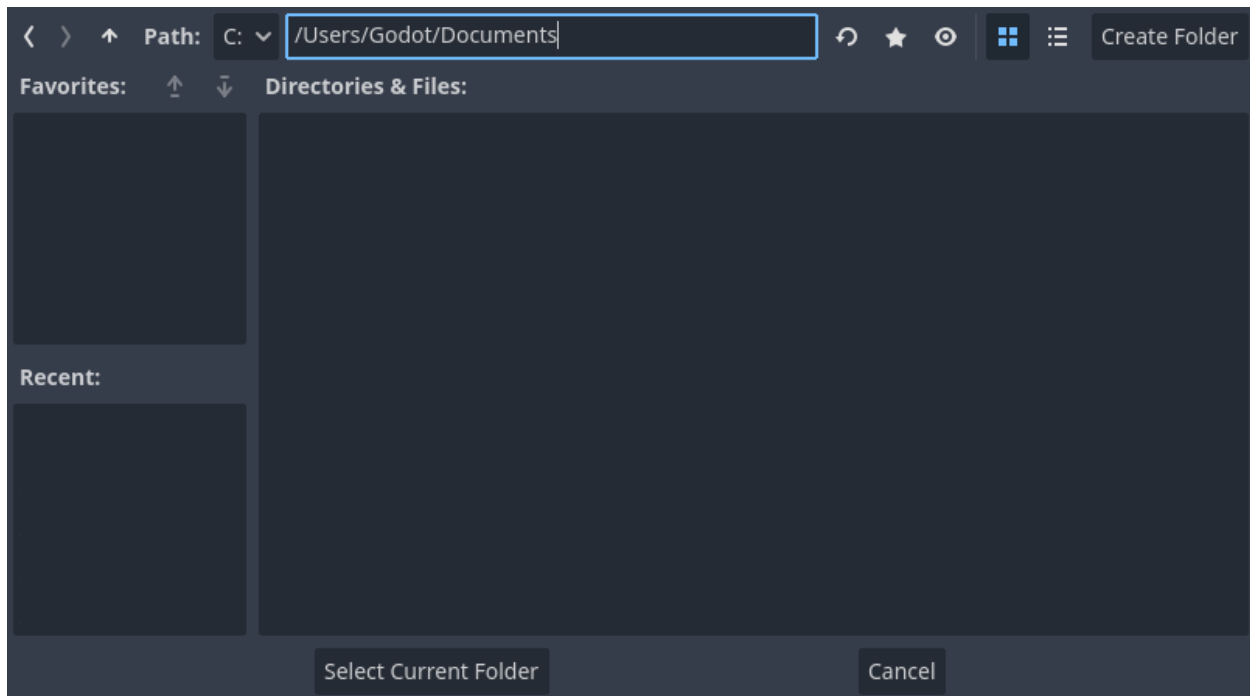
1. Click the New button on the top-left of the window.
2. Give the project a name, choose an empty folder on your computer to save the files, and select a rendering backend.
3. Click the Create & Edit button to create the project folder and open it in the editor.



The screenshot shows the 'New Project' dialog box in Godot. It has a dark theme. At the top, it says 'Project Name:' followed by a text field containing 'New Game Project' and a 'Create Folder' button. Below that, it says 'Project Path:' followed by a text field containing 'C:/Users/Godot/Documents/New Game Project', a green checkmark icon, and a 'Browse' button. Underneath is the 'Renderer:' section with three radio buttons: 'Forward+' (selected), 'Mobile', and 'Compatibility'. To the right of these buttons is a list of features for the selected renderer: 'Supports desktop platforms only.', 'Advanced 3D graphics available.', 'Can scale to large complex scenes.', 'Uses RenderingDevice backend.', and 'Slower rendering of simple scenes.' Below the renderer section is a note: 'The renderer can be changed later, but scenes may need to be adjusted.' At the bottom, there is a 'Version Control Metadata:' label, a dropdown menu showing 'Git', and two buttons: 'Create & Edit' and 'Cancel'.

Using the file browser

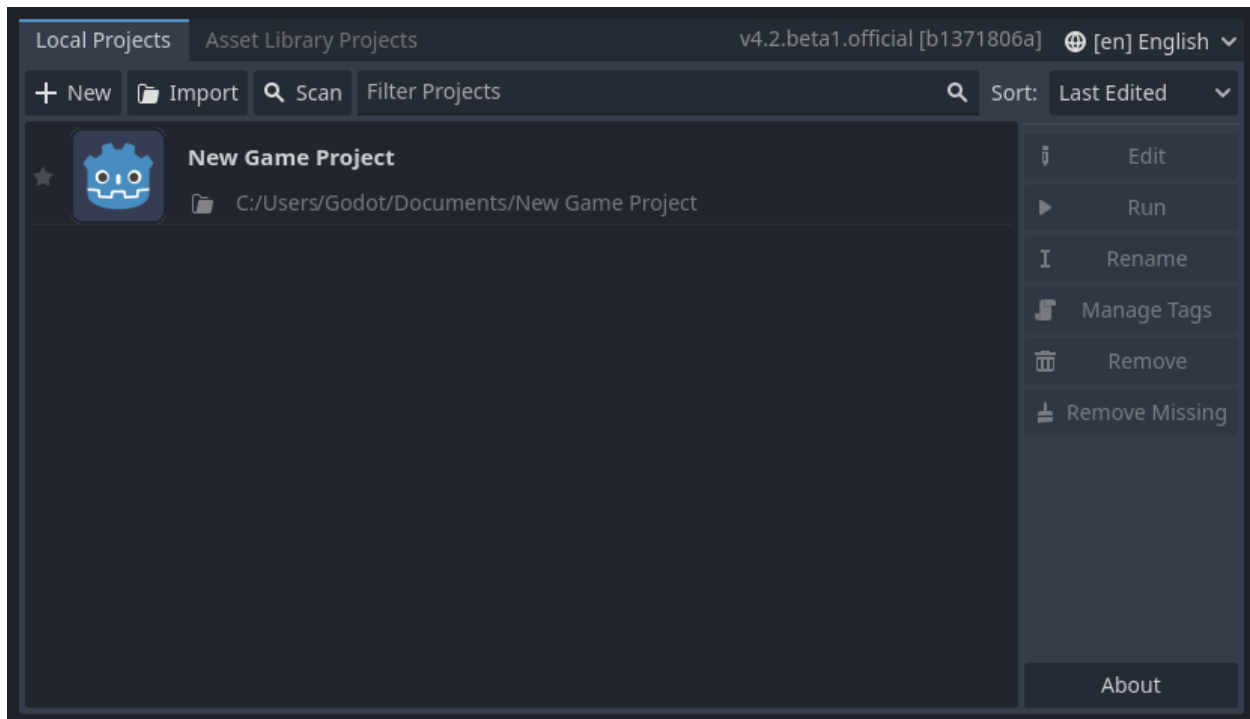
Click the Browse button to open Godot's file browser and pick a location or type the folder's path in the Project Path field.



When you see the green tick on the right, it means the engine detects an empty folder. You can also click the Create Folder button to create an empty folder based on your project's name.

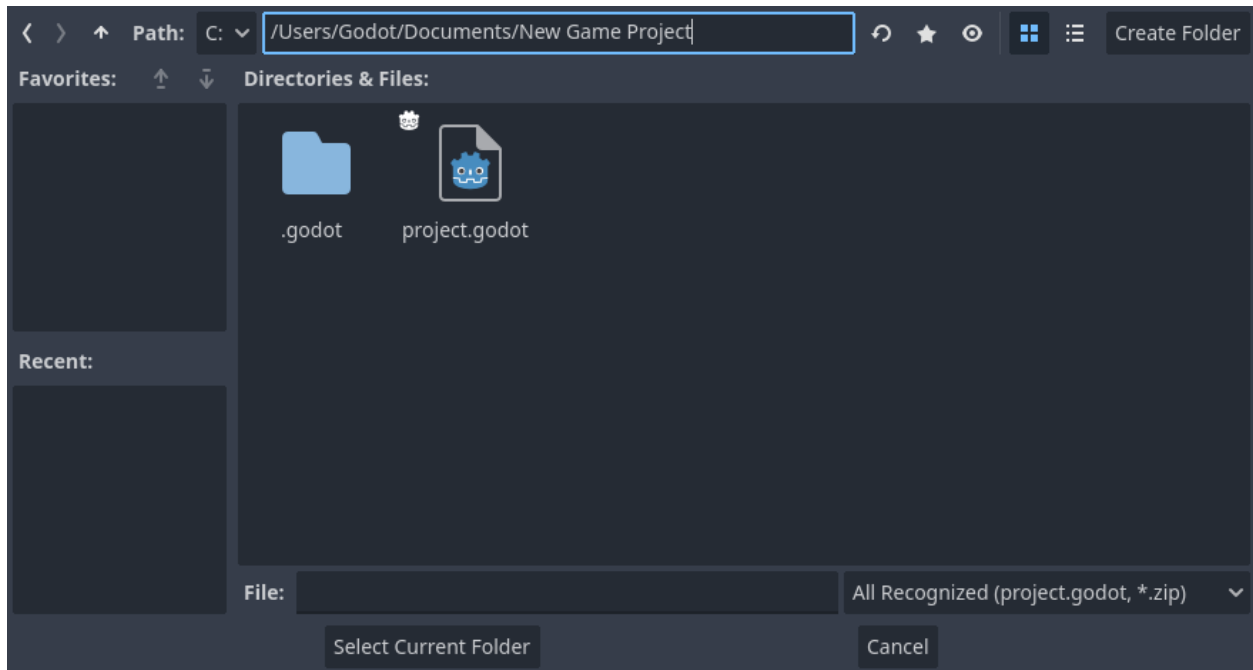
Opening and importing projects

The next time you open the Project Manager, you'll see your new project in the list. Double click on it to open it in the editor.

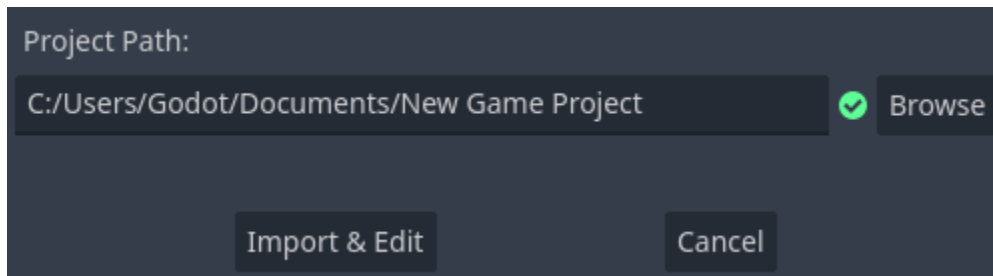


You can similarly import existing projects using the Import button. Locate the folder that contains the

project or the project.godot file to import and edit it.



When the folder path is correct, you'll see a green checkmark.

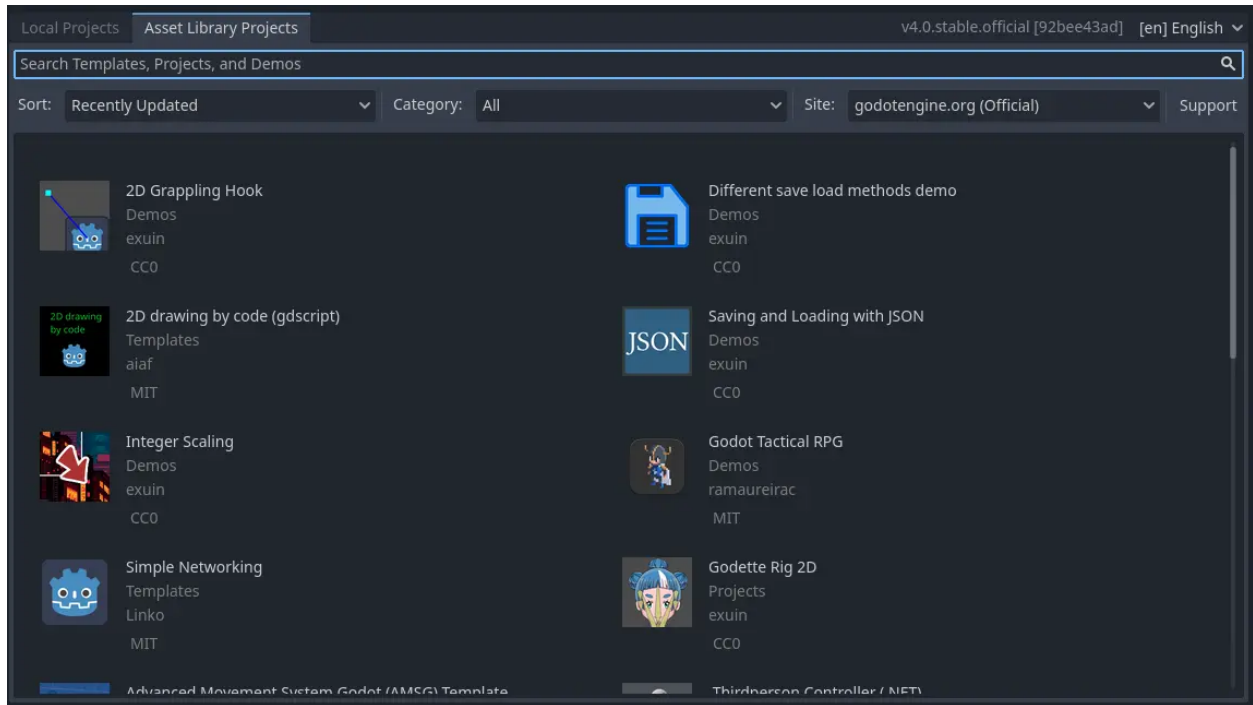


Downloading demos and templates

From the Asset Library Projects tab you can download open source project templates and demos from the Asset Library to help you get started faster.

To download a demo or template:

1. Click on its title.
2. On the page that opens, click the download button.
3. Once it finished downloading, click install and choose where you want to save the project.



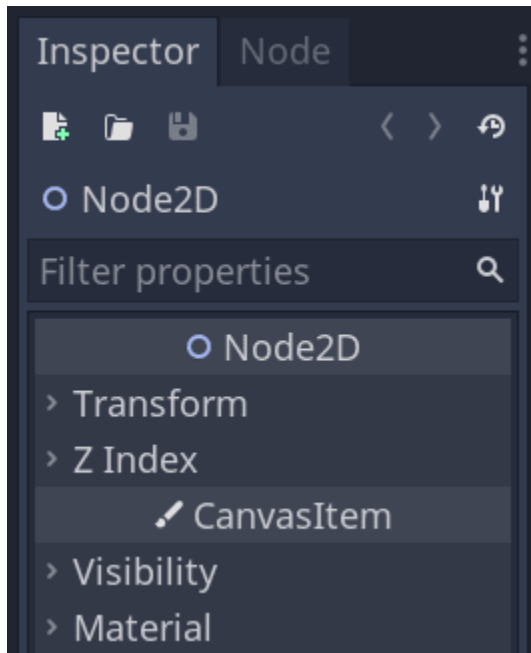
The Inspector

This page explains how the Inspector dock works in-depth. You will learn how to edit properties, fold and unfold areas, use the search bar, and more.

Warning: This page is a work-in-progress.

Overview of the interface

Let's start by looking at the dock's main parts.



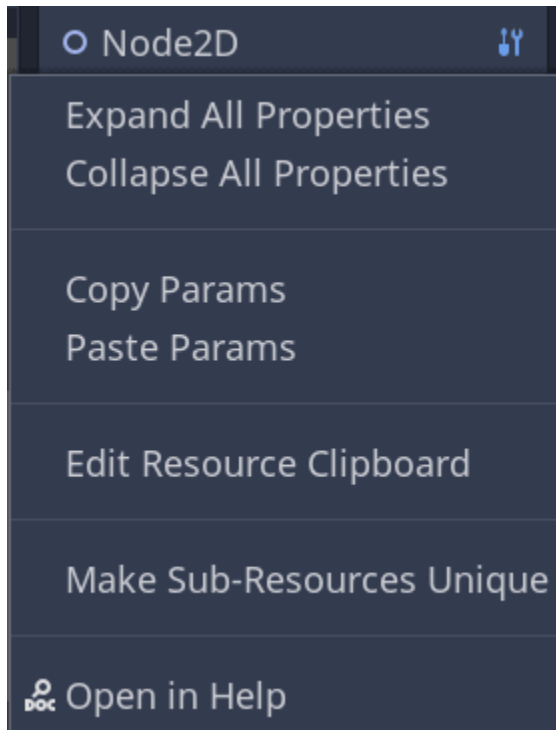
At the top are the file and navigation buttons.



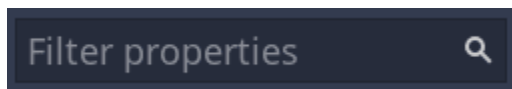
Below it, you can find the selected node's name, its type, and the tools menu on the right side.



If you click the tool menu icon, a drop-down menu offers some view and edit options.



Then comes the search bar. Type anything in it to filter displayed properties. Delete the text to clear the search.



Project Settings

This page explains how to use the Project Settings window. If you would like to access and modify project settings via code, see [ProjectSettings](#).

Godot stores the project settings in a `project.godot` file, a plain text file in INI format. There are dozens of settings you can change to control a project's execution. To simplify this process, Godot provides a project settings dialog, which acts as a front-end to editing a `project.godot` file.

To access that dialog, select `Project -> Project Settings`.

Once the window opens, let's select a main scene. Locate the `Application/Run/Main Scene` property and click on it to select `'hello.tscn'`.

The project settings dialog provides a lot of options that can be saved to a `project.godot` file and shows their default values. If you change a value, a tick appears to the left of its name. This means that the property will be saved in the `project.godot` file and remembered.

Default editor shortcuts

Many of Godot Editor functions can be executed with keyboard shortcuts. This page lists functions which have associated shortcuts by default, but many others are available for customization in editor settings as well. To change keys associated with these and other actions navigate to Editor -> Editor Settings -> Shortcuts.

While some actions are universal, a lot of shortcuts are specific to individual tools. For this reason it is possible for some key combinations to be assigned to more than one function. The correct action will be performed depending on the context.

Note: While Windows and Linux builds of the editor share most of the default settings, some shortcuts may differ for macOS version. This is done for better integration of the editor into macOS ecosystem. Users fluent with standard shortcuts on that OS should find Godot Editor's default key mapping intuitive.

General Editor Actions

Action name	Windows, Linux	macOS	Editor setting
Open 2D Editor	Ctrl + F1	Alt + 1	editor/editor_2d
Open 3D Editor	Ctrl + F2	Alt + 2	editor/editor_3d
Open Script Editor	Ctrl + F3	Alt + 3	editor/editor_script
Search Help	F1	Alt + Space	editor/editor_help
Distraction Free Mode	Ctrl + Shift + F11	Cmd + Ctrl + D	editor/distraction_free_mode
Next tab	Ctrl + Tab	Cmd + Tab	editor/next_tab
Previous tab	Ctrl + Shift + Tab	Cmd + Shift + Tab	editor/prev_tab
Filter Files	Ctrl + Alt + P	Cmd + Alt + P	editor/filter_files
Open Scene	Ctrl + O	Cmd + O	editor/open_scene
Close Scene	Ctrl + Shift + W	Cmd + Shift + W	editor/close_scene
Reopen Closed Scene	Ctrl + Shift + T	Cmd + Shift + T	editor/reopen_closed_scene
Save Scene	Ctrl + S	Cmd + S	editor/save_scene
Save Scene As	Ctrl + Shift + S	Cmd + Shift + S	editor/save_scene_as
Save All Scenes	Ctrl + Shift + Alt + S	Cmd + Shift + Alt + S	editor/save_all_scenes
Quick Open	Shift + Alt + O	Shift + Alt + O	editor/quick_open
Quick Open Scene	Ctrl + Shift + O	Cmd + Shift + O	editor/quick_open_scene
Quick Open Script	Ctrl + Alt + O	Cmd + Alt + O	editor/quick_open_script
Undo	Ctrl + Z	Cmd + Z	editor/undo
Redo	Ctrl + Shift + Z	Cmd + Shift + Z	editor/redo
Quit	Ctrl + Q	Cmd + Q	editor/file_quit
Quit to Project List	Ctrl + Shift + Q	Shift + Alt + Q	editor/quit_to_project_list
Take Screenshot	Ctrl + F12	Cmd + F12	editor/take_screenshot
Toggle Fullscreen	Shift + F11	Cmd + Ctrl + F	editor/fullscreen_mode
Play	F5	Cmd + B	editor/play
Pause Scene	F7	Cmd + Ctrl + Y	editor/pause_scene
Stop	F8	Cmd + .	editor/stop
Play Scene	F6	Cmd + R	editor/play_scene
Play Custom Scene	Ctrl + Shift + F5	Cmd + Shift + R	editor/play_custom_scene
Expand Bottom Panel	Shift + F12	Shift + F12	editor/bottom_panel_expand

2D / Canvas Item Editor

Action name	Windows, Linux	macOS	Editor setting
Zoom In	Ctrl + =	Cmd + =	canvas_item_editor/zoom_plus
Zoom Out	Ctrl + -	Cmd + -	canvas_item_editor/zoom_minus
Zoom Reset	Ctrl + 0	Cmd + 0	canvas_item_editor/zoom_reset
Pan View	Space	Space	canvas_item_editor/pan_view
Select Mode	Q	Q	canvas_item_editor/select_mode
Move Mode	W	W	canvas_item_editor/move_mode
Rotate Mode	E	E	canvas_item_editor/rotate_mode
Scale Mode	S	S	canvas_item_editor/scale_mode
Ruler Mode	R	R	canvas_item_editor/ruler_mode
Use Smart Snap	Shift + S	Shift + S	canvas_item_editor/use_smart_snap
Use Grid Snap	Shift + G	Shift + G	canvas_item_editor/use_grid_snap
Multiply grid step by 2	Num *	Num *	canvas_item_editor/multiply_grid_step
Divide grid step by 2	Num /	Num /	canvas_item_editor/divide_grid_step
Always Show Grid	G	G	canvas_item_editor/show_grid
Show Helpers	H	H	canvas_item_editor/show_helpers
Show Guides	Y	Y	canvas_item_editor/show_guides
Center Selection	F	F	canvas_item_editor/center_selection
Frame Selection	Shift + F	Shift + F	canvas_item_editor/frame_selection
Preview Canvas Scale	Ctrl + Shift + P	Cmd + Shift + P	canvas_item_editor/preview_canvas_scale
Insert Key	Ins	Ins	canvas_item_editor/anim_insert_key
Insert Key (Existing Tracks)	Ctrl + Ins	Cmd + Ins	canvas_item_editor/anim_insert_key_existing_tracks
Make Custom Bones from Nodes	Ctrl + Shift + B	Cmd + Shift + B	canvas_item_editor/skeleton_make_bones
Clear Pose	Shift + K	Shift + K	canvas_item_editor/anim_clear_pose

3D / Spatial Editor

Action name	Windows, Linux	macOS	Editor setting
Toggle Freeload	Shift + F	Shift + F	spatial_editor/freeload_toggle
Freeload Left	A	A	spatial_editor/freeload_left
Freeload Right	D	D	spatial_editor/freeload_right
Freeload Forward	W	W	spatial_editor/freeload_forward
Freeload Backwards	S	S	spatial_editor/freeload_backwards
Freeload Up	E	E	spatial_editor/freeload_up
Freeload Down	Q	Q	spatial_editor/freeload_down
Freeload Speed Modifier	Shift	Shift	spatial_editor/freeload_speed_modifier
Freeload Slow Modifier	Alt	Alt	spatial_editor/freeload_slow_modifier
Select Mode	Q	Q	spatial_editor/tool_select
Move Mode	W	W	spatial_editor/tool_move
Rotate Mode	E	E	spatial_editor/tool_rotate
Scale Mode	R	R	spatial_editor/tool_scale
Use Local Space	T	T	spatial_editor/local_coords
Use Snap	Y	Y	spatial_editor/snap
Snap Object to Floor	PgDown	PgDown	spatial_editor/snap_to_floor

continues on next page

Table 1 – continued from previous page

Action name	Windows, Linux	macOS	Editor setting
Top View	Num 7	Num 7	spatial_editor/top_view
Bottom View	Alt + Num 7	Alt + Num 7	spatial_editor/bottom_view
Front View	Num 1	Num 1	spatial_editor/front_view
Rear View	Alt + Num 1	Alt + Num 1	spatial_editor/rear_view
Right View	Num 3	Num 3	spatial_editor/right_view
Left View	Alt + Num 3	Alt + Num 3	spatial_editor/left_view
Switch Perspective/Orthogonal View	Num 5	Num 5	spatial_editor/switch_perspective_orthogo
Insert Animation Key	K	K	spatial_editor/insert_anim_key
Focus Origin	O	O	spatial_editor/focus_origin
Focus Selection	F	F	spatial_editor/focus_selection
Align Transform with View	Ctrl + Alt + M	Cmd + Alt + M	spatial_editor/align_transform_with_view
Align Rotation with View	Ctrl + Alt + F	Cmd + Alt + F	spatial_editor/align_rotation_with_view
1 Viewport	Ctrl + 1	Cmd + 1	spatial_editor/1_viewport
2 Viewports	Ctrl + 2	Cmd + 2	spatial_editor/2_viewports
2 Viewports (Alt)	Ctrl + Alt + 2	Cmd + Alt + 2	spatial_editor/2_viewports_alt
3 Viewports	Ctrl + 3	Cmd + 3	spatial_editor/3_viewports
3 Viewports (Alt)	Ctrl + Alt + 3	Cmd + Alt + 3	spatial_editor/3_viewports_alt
4 Viewports	Ctrl + 4	Cmd + 4	spatial_editor/4_viewports

Text Editor

Action name	Windows, Linux	macOS	Editor setting
Cut	Ctrl + X	Cmd + X	script_text_editor/cut
Copy	Ctrl + C	Cmd + C	script_text_editor/copy
Paste	Ctrl + V	Cmd + V	script_text_editor/paste
Select All	Ctrl + A	Cmd + A	script_text_editor/select_all
Find	Ctrl + F	Cmd + F	script_text_editor/find
Find Next	F3	Cmd + G	script_text_editor/find_next
Find Previous	Shift + F3	Cmd + Shift + G	script_text_editor/find_previous
Find in Files	Ctrl + Shift + F	Cmd + Shift + F	script_text_editor/find_in_files
Replace	Ctrl + R	Alt + Cmd + F	script_text_editor/replace
Replace in Files	Ctrl + Shift + R	Cmd + Shift + R	script_text_editor/replace_in_files
Undo	Ctrl + Z	Cmd + Z	script_text_editor/undo
Redo	Ctrl + Y	Cmd + Y	script_text_editor/redo
Move Up	Alt + Up Arrow	Alt + Up Arrow	script_text_editor/move_up
Move Down	Alt + Down Arrow	Alt + Down Arrow	script_text_editor/move_down
Delete Line	Ctrl + Shift + K	Cmd + Shift + K	script_text_editor/delete_line
Toggle Comment	Ctrl + K	Cmd + K	script_text_editor/toggle_comment
Fold/Unfold Line	Alt + F	Ctrl + Cmd + F	script_text_editor/toggle_fold_line
Duplicate Selection	Ctrl + Shift + D	Cmd + Shift + C	script_text_editor/duplicate_selection
Complete Symbol	Ctrl + Space	Ctrl + Space	script_text_editor/complete_symbol
Evaluate Selection	Ctrl + Shift + E	Cmd + Shift + E	script_text_editor/evaluate_selection
Trim Trailing Whitespace	Ctrl + Alt + T	Cmd + Alt + T	script_text_editor/trim_trailing_whitespace
Uppercase	Shift + F4	Shift + F4	script_text_editor/convert_to_uppercase
Lowercase	Shift + F5	Shift + F5	script_text_editor/convert_to_lowercase
Capitalize	Shift + F6	Shift + F6	script_text_editor/capitalize
Convert Indent to Spaces	Ctrl + Shift + Y	Cmd + Shift + Y	script_text_editor/convert_indent_to_spaces
Convert Indent to Tabs	Ctrl + Shift + I	Cmd + Shift + I	script_text_editor/convert_indent_to_tabs

continues on next page

Table 2 – continued from previous page

Action name	Windows, Linux	macOS	Editor setting
Auto Indent	Ctrl + I	Cmd + I	script_text_editor/auto_indent
Toggle Bookmark	Ctrl + Alt + B	Cmd + Alt + B	script_text_editor/toggle_bookmark
Go to Next Bookmark	Ctrl + B	Cmd + B	script_text_editor/goto_next_bookmark
Go to Previous Bookmark	Ctrl + Shift + B	Cmd + Shift + B	script_text_editor/goto_previous_bookmark
Go to Function	Ctrl + Alt + F	Ctrl + Cmd + J	script_text_editor/goto_function
Go to Line	Ctrl + L	Cmd + L	script_text_editor/goto_line
Toggle Breakpoint	F9	Cmd + Shift + B	script_text_editor/toggle_breakpoint
Remove All Breakpoints	Ctrl + Shift + F9	Cmd + Shift + F9	script_text_editor/remove_all_breakpoints
Go to Next Breakpoint	Ctrl + .	Cmd + .	script_text_editor/goto_next_breakpoint
Go to Previous Breakpoint	Ctrl + ,	Cmd + ,	script_text_editor/goto_previous_breakpoint
Contextual Help	Alt + F1	Alt + Shift + Space	script_text_editor/contextual_help

Script Editor

Action name	Windows, Linux	macOS	Editor setting
Find	Ctrl + F	Cmd + F	script_editor/find
Find Next	F3	F3	script_editor/find_next
Find Previous	Shift + F3	Shift + F3	script_editor/find_previous
Find in Files	Ctrl + Shift + F	Cmd + Shift + F	script_editor/find_in_files
Move Up	Shift + Alt + Up Arrow	Shift + Alt + Up Arrow	script_editor/window_move_up
Move Down	Shift + Alt + Down Arrow	Shift + Alt + Down Arrow	script_editor/window_move_down
Next Script	Ctrl + Shift + .	Cmd + Shift + .	script_editor/next_script
Previous Script	Ctrl + Shift + ,	Cmd + Shift + ,	script_editor/prev_script
Reopen Closed Script	Ctrl + Shift + T	Cmd + Shift + T	script_editor/reopen_closed_script
Save	Ctrl + Alt + S	Cmd + Alt + S	script_editor/save
Save All	Ctrl + Shift + Alt + S	Cmd + Shift + Alt + S	script_editor/save_all
Soft Reload Script	Ctrl + Shift + R	Cmd + Shift + R	script_editor/reload_script_soft
History Previous	Alt + Left Arrow	Alt + Left Arrow	script_editor/history_previous
History Next	Alt + Right Arrow	Alt + Right Arrow	script_editor/history_next
Close	Ctrl + W	Cmd + W	script_editor/close_file
Run	Ctrl + Shift + X	Cmd + Shift + X	script_editor/run_file
Toggle Scripts Panel	Ctrl + \	Cmd + \	script_editor/toggle_scripts_panel
Zoom In	Ctrl + =	Cmd + =	script_editor/zoom_in
Zoom Out	Ctrl + -	Cmd + -	script_editor/zoom_out
Reset Zoom	Ctrl + 0	Cmd + 0	script_editor/reset_zoom

Editor Output

Action name	Windows, Linux	macOS	Editor setting
Copy Selection	Ctrl + C	Cmd + C	editor/copy_output
Clear Output	Ctrl + Shift + K	Cmd + Shift + K	editor/clear_output

Debugger

Action name	Windows, Linux	macOS	Editor setting
Step Into	F11	F11	debugger/step_into
Step Over	F10	F10	debugger/step_over
Continue	F12	F12	debugger/continue

File Dialog

Action name	Windows, Linux	macOS	Editor setting
Go Back	Alt + Left Arrow	Alt + Left Arrow	file_dialog/go_back
Go Forward	Alt + Right Arrow	Alt + Right Arrow	file_dialog/go_forward
Go Up	Alt + Up Arrow	Alt + Up Arrow	file_dialog/go_up
Refresh	F5	F5	file_dialog/refresh
Toggle Hidden Files	Ctrl + H	Cmd + H	file_dialog/toggle_hidden_files
Toggle Favorite	Alt + F	Alt + F	file_dialog/toggle_favorite
Toggle Mode	Alt + V	Alt + V	file_dialog/toggle_mode
Create Folder	Ctrl + N	Cmd + N	file_dialog/create_folder
Delete	Del	Cmd + BkSp	file_dialog/delete
Focus Path	Ctrl + D	Cmd + D	file_dialog/focus_path
Move Favorite Up	Ctrl + Up Arrow	Cmd + Up Arrow	file_dialog/move_favorite_up
Move Favorite Down	Ctrl + Down Arrow	Cmd + Down Arrow	file_dialog/move_favorite_down

FileSystem Dock

Action name	Windows, Linux	macOS	Editor setting
Copy Path	Ctrl + C	Cmd + C	filesystem_dock/copy_path
Duplicate	Ctrl + D	Cmd + D	filesystem_dock/duplicate
Delete	Del	Cmd + BkSp	filesystem_dock/delete

Scene Tree Dock

Action name	Windows, Linux	macOS	Editor setting
Add Child Node	Ctrl + A	Cmd + A	scene_tree/add_child_node
Batch Rename	Ctrl + F2	Cmd + F2	scene_tree/batch_rename
Copy Node Path	Ctrl + Shift + C	Cmd + Shift + C	scene_tree/copy_node_path
Delete	Del	Cmd + BkSp	scene_tree/delete
Force Delete	Shift + Del	Shift + Del	scene_tree/delete_no_confirm
Duplicate	Ctrl + D	Cmd + D	scene_tree/duplicate
Move Up	Ctrl + Up Arrow	Cmd + Up Arrow	scene_tree/move_up
Move Down	Ctrl + Down Arrow	Cmd + Down Arrow	scene_tree/move_down

Animation Track Editor

Action name	Windows, Linux	macOS	Editor setting
Duplicate Selection	Ctrl + D	Cmd + D	animation_editor/duplicate_selection
Duplicate Transposed	Ctrl + Shift + D	Cmd + Shift + D	animation_editor/duplicate_selection_transposed
Delete Selection	Del	Cmd + BkSp	animation_editor/delete_selection
Go to Next Step	Ctrl + Right Arrow	Cmd + Right Arrow	animation_editor/goto_next_step
Go to Previous Step	Ctrl + Left Arrow	Cmd + Left Arrow	animation_editor/goto_prev_step

Tile Map Editor

Action name	Windows, Linux	macOS	Editor setting
Find Tile	Ctrl + F	Cmd + F	tile_map_editor/find_tile
Pick Tile	I	I	tile_map_editor/pick_tile
Paint Tile	P	P	tile_map_editor/paint_tile
Bucket Fill	G	G	tile_map_editor/bucket_fill
Transpose	T	T	tile_map_editor/transpose
Flip Horizontally	X	X	tile_map_editor/flip_horizontal
Flip Vertically	Z	Z	tile_map_editor/flip_vertical
Rotate Left	A	A	tile_map_editor/rotate_left
Rotate Right	S	S	tile_map_editor/rotate_right
Clear Transform	W	W	tile_map_editor/clear_transform
Select	M	M	tile_map_editor/select
Cut Selection	Ctrl + X	Cmd + X	tile_map_editor/cut_selection
Copy Selection	Ctrl + C	Cmd + C	tile_map_editor/copy_selection
Erase Selection	Del	Cmd + BkSp	tile_map_editor/erase_selection

Tileset Editor

Action name	Windows, Linux	macOS	Editor setting
Next Coordinate	PgDown	PgDown	tileset_editor/next_shape
Previous Coordinate	PgUp	PgUp	tileset_editor/previous_shape
Region Mode	1	1	tileset_editor/editmode_region
Collision Mode	2	2	tileset_editor/editmode_collision
Occlusion Mode	3	3	tileset_editor/editmode_occlusion
Navigation Mode	4	4	tileset_editor/editmode_navigation
Bitmask Mode	5	5	tileset_editor/editmode_bitmask
Priority Mode	6	6	tileset_editor/editmode_priority
Icon Mode	7	7	tileset_editor/editmode_icon
Z Index Mode	8	8	tileset_editor/editmode_z_index

Customizing the interface

Godot's interface lives in a single window by default. Since Godot 4.0, you can split several elements to separate windows to better make use of multi-monitor setups.

Moving and resizing docks

Click and drag on the edge of any dock or panel to resize it horizontally or vertically:



Fig. 3: Resizing a dock in the editor

Click the "3 vertical dots" icon at the top of any dock to change its location, or split it to a separate window by choosing Make Floating in the submenu that appears:



Fig. 4: Moving a dock in the editor

To move a floating dock back to the editor window, close the dock window using the × button in the top-right corner of the window (or in the top-left corner on macOS). Alternatively, you can press `Alt + F4` while the split window is focused.

Splitting the script or shader editor to its own window

Note: This feature is only available on platforms that support spawning multiple windows: Windows, macOS and Linux.

This feature is also not available if Single Window Mode is enabled in the Editor Settings.

Since Godot 4.1, you can split the script or shader editor to its own window.

To split the script editor to its own window, click the corresponding button in the top-right corner of the script editor:

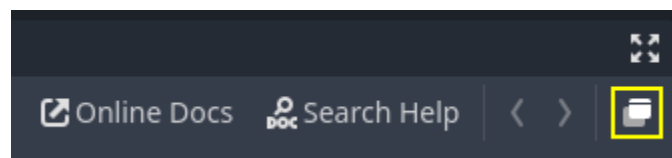


Fig. 5: Splitting the script editor to its own window

To split the shader editor to its own window, click the corresponding button in the top-right corner of the script editor:

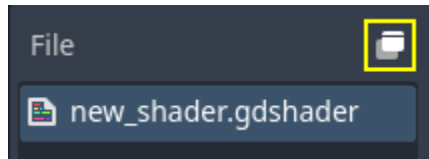


Fig. 6: Splitting the shader editor to its own window

To go back to the previous state (with the script/shader editor embedded in the editor window), close the split window using the \times button in the top-right corner of the window (or in the top-left corner on macOS). Alternatively, you can press $\text{Alt} + \text{F4}$ while the split window is focused.

Customizing editor layouts

You may want to save and load a dock configuration depending on the kind of task you're working on. For instance, when working on animating a character, it may be more convenient to have docks laid out in a different fashion compared to when you're designing a level.

For this purpose, Godot provides a way to save and restore editor layouts. Before saving a layout, make changes to the docks you'd like to save. The following changes are persisted to the saved layout:

- Moving a dock.
- Resizing a dock.
- Making a dock floating.
- Changing a floating dock's position or size.
- FileSystem dock properties: split mode, display mode, sorting order, file list display mode, selected paths and unfolded paths.

Note: Splitting the script or shader editor to its own window is not persisted as part of a layout.

After making changes, open the Editor menu at the top of the editor then choose Editor Layouts > Save. Enter a name for the layout, then click Save. If you've already saved an editor layout, you can choose to override an existing layout using the list.

After making changes, open the Editor menu at the top of the editor then choose Editor Layouts. In the dropdown list, you will see a list of saved editor layouts, plus Default which is a hardcoded editor layout that can't be removed. The default layout matches a fresh Godot installation with no changes made to the docks' position and size, and no floating docks.

You can remove a layout using the Delete option in the Editor Layouts dropdown.

Tip: If you name the saved layout Default (case-sensitive), the default editor layout will be overwritten. Note that the Default does not appear in the list of layouts to overwrite until you overwrite it once, but you can still write its name manually.

You can go back to the standard default layout by removing the Default layout after overriding it. (This option does not appear if you haven't overridden the default layout yet.)

Editor layouts are saved to a file named `editor_layouts.cfg` in the configuration path of the Editor data paths.

Customizing editor settings

In the Editor menu at the top of the editor, you can find an Editor Settings option. This opens a window similar to the Project Settings, but with settings used by the editor. These settings are shared across all projects and are not saved in the project files.

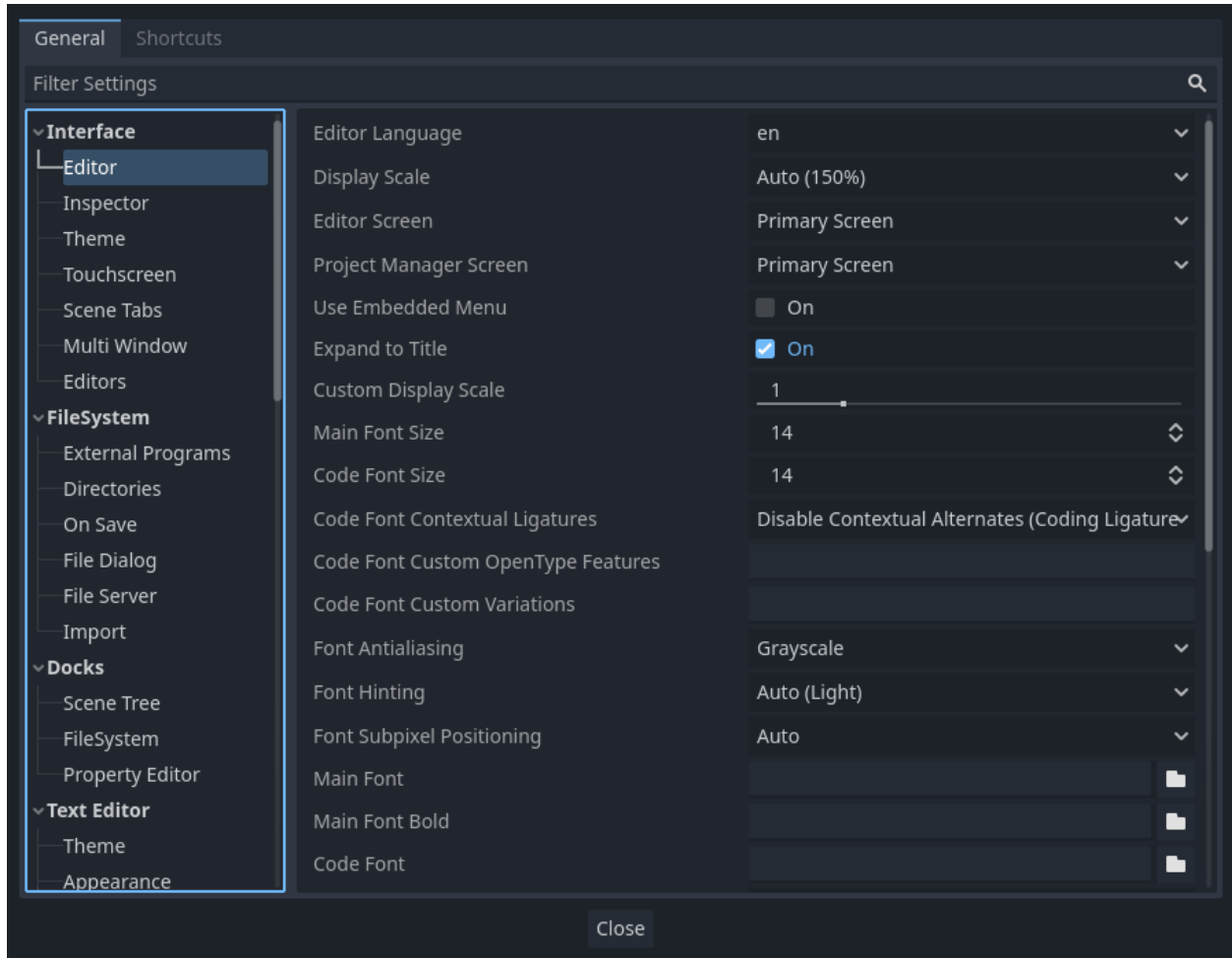


Fig. 7: The Editor Settings window

Some commonly changed settings are:

- **Interface > Editor > Editor Language:** Controls the language the editor displays in. To make English tutorials easier to follow, you may want to change this to English so that menu names are identical to names referred to by tutorials. The language can also be changed in the top-right corner of the project manager.
- **Interface > Editor > Display Scale:** Controls how large UI elements display on screen. The default Auto setting finds a suitable value based on your display's DPI and resolution. Due to engine limitations, it only takes the display-provided scaling factor on macOS, not on Windows or Linux.
- **Interface > Editor > Single Window Mode:** If enabled, this forces the editor to use a single window. This disables certain features such as splitting the script/shaders editor to their own window. Single-window mode can be more stable, especially on Linux when using Wayland.
- **Interface > Theme > Preset:** The editor theme preset to use. The Light theme preset may be easier to read if you're outdoors or in a room with sunlight. The Black (OLED) preset can reduce power

consumption on OLED displays, which are increasingly common in laptops and phones/tablets.

- **FileSystem > Directories > Autoscan Project Path:** This can be set to a folder path that will be automatically scanned for projects in the project manager every time it starts.
- **FileSystem > Directories > Default Project Path:** Controls the default location where new projects are created in the project manager.
- **Editors > 3D > Emulate Numpad:** This allows using the top row 0-9 keys in the 3D editor as their equivalent numpad keys. It's recommended to enable this option if you don't have a number pad on your keyboard.
- **Editors > 3D > Emulate 3 Button Mouse:** This allows using the pan, zoom and orbit modifiers in the 3D editor even when not holding down any mouse button. It's recommended to enable this option if you're using a trackpad.

See the `EditorSettings` class reference for a complete description of most editor settings. You can also hover an editor setting's name with the mouse in the Editor Settings to show its description.

2.13.2 Android editor

Godot offers a native port of the editor running entirely on Android devices. The Android port can be downloaded from the [Android Downloads page](#). While we strive for feature parity with the Desktop version of the editor, the Android port has a certain amount of caveats you should be aware of.

Using the Android editor

In 2023, we added a [Android port of the editor](#) that can be used to work on new or existing projects on Android devices.

Note: The Android editor is in beta testing stage, while we continue to refine the experience, and bring it up to parity with the Desktop version of the editor. See [Required Permissions](#) below.

Android devices support

The Android editor requires devices running Android 5 Lollipop or higher, with at least OpenGL 3 support. This includes (not exhaustive):

- Android tablets, foldables and large phones
- Android-powered netbooks
- Chromebooks supporting Android apps

Required Permissions

The Android editor requires the [All files access permission](#). The permission allows the editor to create / import / read project files from any file locations on the device. Without the permission, the editor is still functional, but has limited access to the device's files and directories.

Limitations & known issues

Here are the known limitations and issues of the Android editor:

- No C#/Mono support
- No support for external script editors
- While available, the Vulkan Forward+ renderer is not recommended due to severe performance issues
- No support for building and exporting an Android APK binary. As a workaround, you can generate and export a [Godot PCK or ZIP file](#)
- No support for building and exporting binaries for other platforms
- Performance and stability issues when using the Vulkan Mobile renderer for a project
- UX not optimized for Android phones form-factor
- [Android Go devices](#) lacks the All files access permission required for device read/write access. As a workaround, when using a Android Go device, it's recommended to create new projects only in the Android Documents or Downloads directories.
- The editor doesn't properly resume when Don't keep activities is enabled in the Developer Options

See also:

See the [list of open issues on GitHub related to the Android editor](#) for a list of known bugs.

2.13.3 Web editor

Godot offers an HTML5 version of the editor running entirely in your browser. No download is required to use it, but it has a certain amount of caveats you should be aware of.

Using the Web editor

Since Godot 3.3, there is a [Web editor](#) you can use to work on new or existing projects.

Note: The web editor is in a preliminary stage. While its feature set may be sufficient for educational purposes, it is currently not recommended for production work. See Limitations below.

Browser support

The Web editor requires support for WebAssembly's SharedArrayBuffer. This is in turn required to support threading in the browser. The following desktop browsers support WebAssembly threading and can therefore run the web editor:

- Chrome 68 or later
- Firefox 79 or later
- Edge 79 or later

Opera and Safari are not supported yet. Safari may work in the future once proper threading support is added.

Mobile browsers are currently not supported.

The web editor only supports the Compatibility rendering method, as there is no stable way to run Vulkan applications on the web yet.

Note: If you use Linux, due to [poor Firefox WebGL performance](#), it's recommended to use a Chromium-based browser instead of Firefox.

Limitations

Due to limitations on the Godot or Web platform side, the following features are currently missing:

- No C#/Mono support.
- No GDExtension support.
- No debugging support. This means GDScript debugging/profiling, live scene editing, the Remote Scene tree dock and other features that rely on the debugger protocol will not work.
- No project exporting. As a workaround, you can download the project source using Project > Tools > Download Project Source and export it using a [native version of the Godot editor](#).
- The editor won't warn you when closing the tab with unsaved changes.
- No lightmap baking support. You can still use existing lightmaps if they were baked with a native version of the Godot editor (e.g. by importing an existing project).

The following features are unlikely to be supported due to inherent limitations of the Web platform:

- No support for external script editors.
- No support for Android one-click deploy.

See also:

See the [list of open issues on GitHub related to the web editor](#) for a list of known bugs.

Importing a project

To import an existing project, the current process is as follows:

- Specify a ZIP file to preload on the HTML5 filesystem using the Preload project ZIP input.
- Run the editor by clicking Start Godot editor. The Godot Project Manager should appear after 10-20 seconds. On slower machines or connections, loading may take up to a minute.
- In the dialog that appears at the middle of the window, specify a name for the folder to create then click the Create Folder button (it doesn't have to match the ZIP archive's name).
- Click Install & Edit and the project will open in the editor.

Attention: It's important to place the project folder somewhere in `/home/web_user/`. If your project folder is placed outside `/home/web_user/`, you will lose your project when closing the editor!

When you follow the steps described above, the project folder will always be located in `/home/web_user/projects`, keeping it safe.

Editing and running a project

Unlike the native version of Godot, the web editor is constrained to a single window. Therefore, it cannot open a new window when running the project. Instead, when you run the project by clicking the Run button or pressing F5, it will appear to "replace" the editor window.

The web editor offers an alternative way to deal with the editor and game windows (which are now "tabs"). You can switch between the Editor and Game tabs using the buttons on the top. You can also close the running game or editor by clicking the × button next to those tabs.

Where are my project files?

Due to browser security limitations, the editor will save the project files to the browser's IndexedDB storage. This storage isn't accessible as a regular folder on your machine, but is abstracted away in a database.

You can download the project files as a ZIP archive by using Project > Tools > Download Project Source. This can be used to export the project using a [native Godot editor](#), since exporting from the web editor isn't supported yet.

In the future, it may be possible to use the [HTML5 FileSystem API](#) to store the project files on the user's filesystem as the native editor would do. However, this isn't implemented yet.

2.13.4 Advanced features

The articles below focus on advanced features useful for experienced developers, such as calling Godot from the command line and using an external text editor like Visual Studio Code or Emacs.

Command line tutorial

Some developers like using the command line extensively. Godot is designed to be friendly to them, so here are the steps for working entirely from the command line. Given the engine relies on almost no external libraries, initialization times are pretty fast, making it suitable for this workflow.

Note: On Windows and Linux, you can run a Godot binary in a terminal by specifying its relative or absolute path.

On macOS, the process is different due to Godot being contained within an .app bundle (which is a folder, not a file). To run a Godot binary from a terminal on macOS, you have to cd to the folder where the Godot application bundle is located, then run Godot.app/Contents/MacOS/Godot followed by any command line arguments. If you've renamed the application bundle from Godot to another name, make sure to edit this command line accordingly.

Command line reference

Legend

- Available in editor builds, debug export templates and release export templates.
- Available in editor builds and debug export templates only.
- Only available in editor builds.

Note that unknown command line arguments have no effect whatsoever. The engine will not warn you when using a command line argument that doesn't exist with a given build type.

General options