Godot Basics:

Godot is a free game engine with a liberal MIT license that ensures it will always remain free to use, modify, and even profit from in an unlimited manner.

But why do we need a game engine to make games? It is true, you can create games without an engine in a variety of languages and development environments. The advantage of a game engine is that many of the things commonly needed for making games are included without the need to build them from scratch. These are presented in a general way that you can then customize for your specific game concept.

This allows you to spend your time on developing your idea, rather than creating all the tools necessary to make your game. The trade-off with this is that your game may call for a feature that is not included in the game engine. With many engines this can be a serious road-block. However as a fully open-source project, Godot can be changed by community effort to include new features and capabilities. Additionally it has a robust extension system that helps ensure additions to the engine do not disrupt or break existing features.

The main areas of functionality a game engine will provide include:

Rendering

Painting graphics to a screen is something that we don’t typically think about, but today there are so many screens with so many hardware profiles that outputting graphics to them could get incredibly complex. In addition there are related technologies such as GPUs that provide additional options for rendering. All of these are based on instruction sets that may be difficult to work with at a low level. A game engine provides a simple means of painting to the screen across a wide variety of hardware profiles, and will usually provide additional means to use features of the GPU.

Godot Specifics:
- 2D and 3D rendering
- XR support
- Support for raster formats (jpg, png)
- Support for 3D formats (gltf 1 and 2, DAE, OJG, FBX)
- Blender file importer

Physics

Even the simplest games make use of some kind of system for physics (think about Pong). Modern games might make use of many aspects of physics for creating game play, including concepts such as gravity, acceleration, friction and much more. Included in the category of physics are the actions of detecting and responding to collision between objects. And all of this becomes a good deal more complex in 3D games. A game engine will provide a system for selectively applying physics effects to game objects without spending long amounts of time in the math of these operations.

Godot Specifics:
- Swappable physics engine
- Bullet Physics (removed in 4), Godot Physics, Jolt Physics

Platform Support

Much like displays, the hardware that is available for running software comes in a staggering array of configurations. From the mobile phone to the multi-GPU gaming desktop, directly coding support for platforms is a difficult task, depending on how many ways you want to distribute your game. A game engine often provides the means to prepare your game for multiple platforms, which once again allows you to concentrate on the game development.

Godot Specifics:
- Publish to Windows, MacOS, Linux
- Publish to Android, IOS
- Publish to Web (single threaded as of 4.3)
- No built-in publish to consoles, requires conversion (proprietary platforms)

Development Environment

The main purpose of an Integrated Development Environment is to provide a single development tool with a predictable workflow, so that you can become familiar one time, then create many different kinds of games.

Godot Specifics:
- Integrated development environment
- Swappable concurrent languages (GDScript, C#, C++)
- Purpose-built Python-style language
- Web hosted IDE

The Godot game engine is a tiny download compared to other game engines, largely owing to a few factors:

- It is purpose built for games. Many other game engines incorporate additional features for things like Visual Effects, or Virtual Production. The core of Godot is smaller because it considers other uses to be extensions to the engine, completely separate.
- Publishing options are likewise not downloaded until needed. For example the .NET runtime is not downloaded until we want to publish our game for Windows (and if we are not targeting windows we will never download that library)

- All code is visible to the community, meaning it comes under review more often and is frequently updated.

Download Godot Here

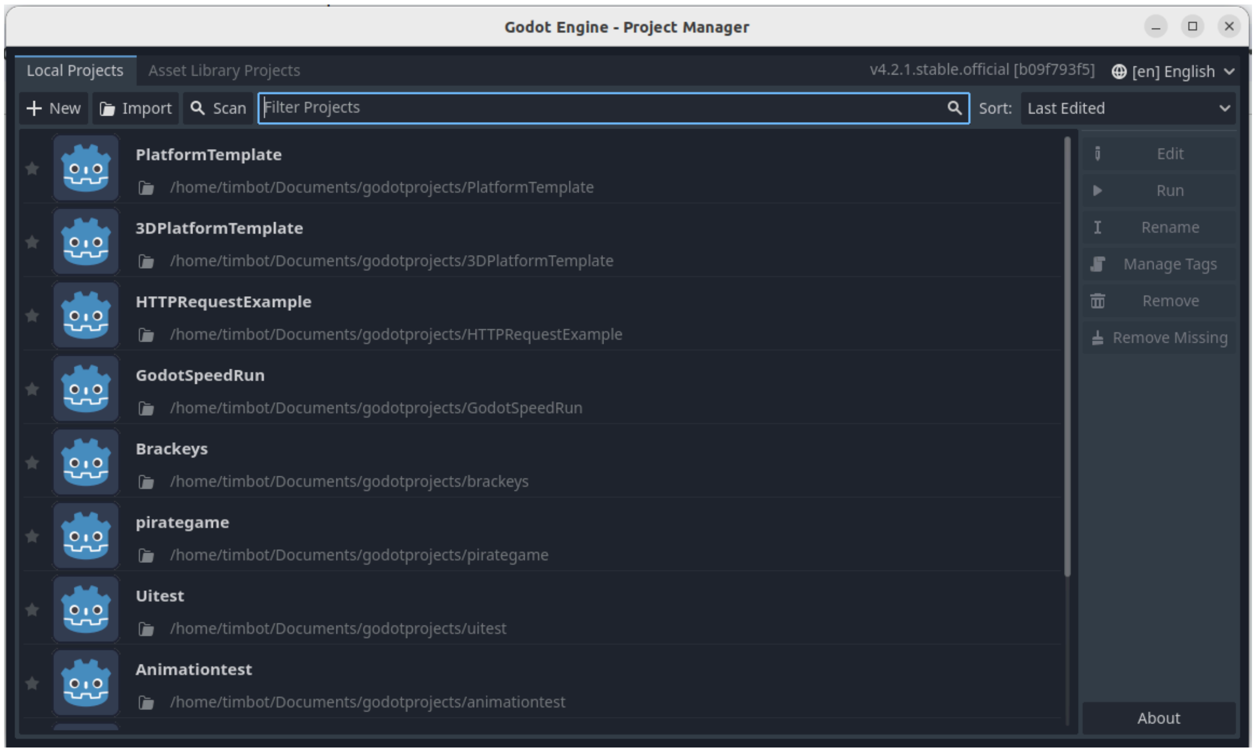
Setting Up

Once downloaded, Godot is easy to launch as it requires no installation - it can run as a completely standalone application. Configuration options you select will be stored in the standard area provided by your operating system, such as the home folder of the currently logged in user.

The first thing you will see is the launcher. Godot will keep track of all of your current projects in the configuration and will present these projects when you start the engine. This will allow you to quickly access and resume your projects.

Godot Specifics:

- Godot requires each project to be kept in a single folder that includes all resources for the project - all sounds, images, models etc. They can not be spread around your folder system, so you may need to copy or move files.
- Godot allows you to run multiple instances at one time. This means you can actually be working across two or more Godot projects at the same time.

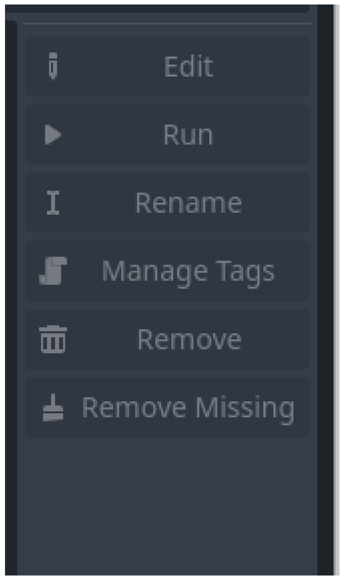


PlatformTemplate
/home/timbot/Documents/godotprojects/PlatformTemplate
3DPlatformTemplate
/home/timbot/Documents/godotprojects/3DPlatformTemplate
HTTPRequestExample
/home/timbot/Documents/godotprojects/HTTPRequestExample
GodotSpeedRun
/home/timbot/Documents/godotprojects/GodotSpeedRun
Brackeys
/home/timbot/Documents/godotprojects/brackeys
pirategame
/home/timbot/Documents/godotprojects/pirategame
Uitest
/home/timbot/Documents/godotprojects/uitest
Animationtest
/home/timbot/Documents/godotprojects/animationtest

To get started you can either import a godot project, or you can create a new one. Creating a project is simple, because you will not create the configuration for the project until the next step. Initially setting up is just a matter of creating/finding a folder to put your project in.

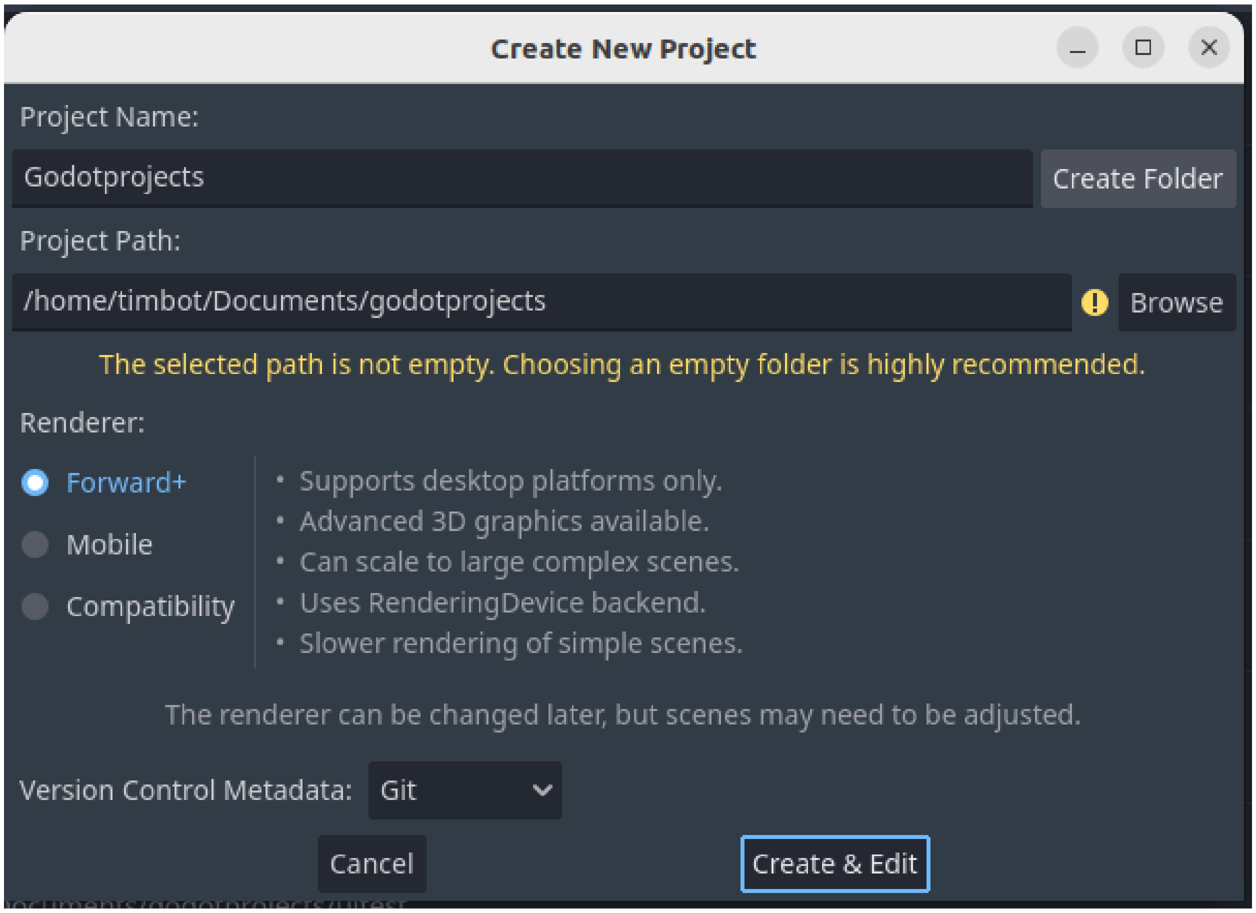
Tip: You should probably choose a single folder in an easily accessible place in your documents to house all of your Godot projects so they don't get spread out. As you progress in your game dev journey this will also be a good place to store assets and plugins you find you are frequently using (although to use them you will always need to copy them into the current project you are working on)

Note that you can either edit existing projects, or you can just run them:



When you run a project from the launcher it will do so using the "main scene" - a concept we will be looking at shortly.

In the image below, we can see that a folder has been created for Godot projects, and inside of that each folder represents a single project:



Create New Project

Project Name:

Godotprojects

Create Folder

Project Path:

/home/timbot/Documents/godotprojects

Browse

The selected path is not empty. Choosing an empty folder is highly recommended.

Renderer:

(●) Forward+
( ) Mobile
( ) Compatibility

- Supports desktop platforms only.
- Advanced 3D graphics available.
- Can scale to large complex scenes.
- Uses RenderingDevice backend.
- Slower rendering of simple scenes.

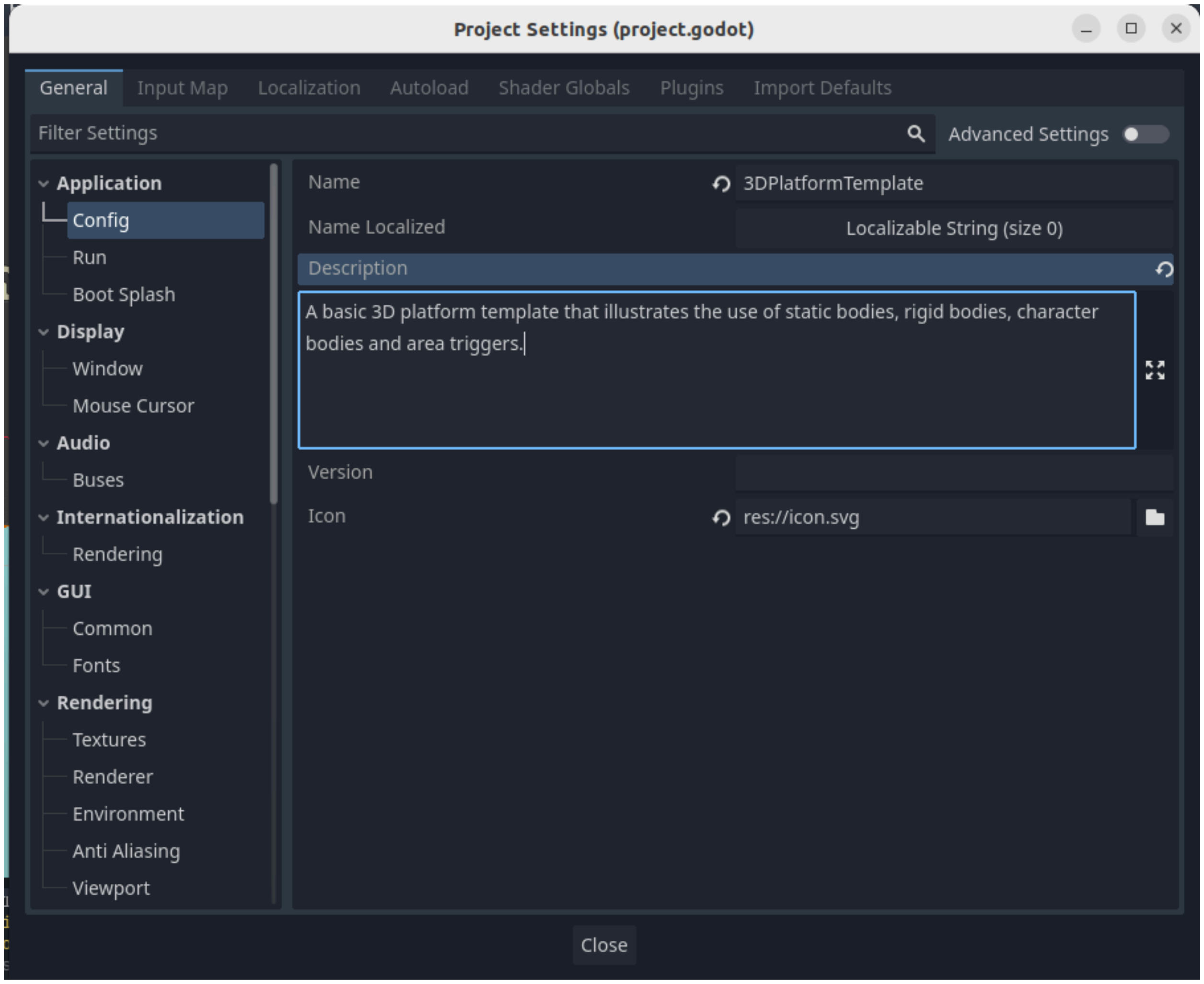
The renderer can be changed later, but scenes may need to be adjusted.

Version Control Metadata: Git

You will also have the option to choose the renderer you want to use for your project. In brief, the options will either turn on or off features of the renderer that are likely to be enabled or disabled on the target devices you are developing for.

It is possible to change rendering options when you are already in development, but it may take a bit of work to adjust a game that is already in development.

With a project created you can visit the configuration options by clicking on Project > Project Settings:



Name: 3DPlatformTemplate

Name Localized: Localizable String (size 0)

Description: A basic 3D platform template that illustrates the use of static bodies, rigid bodies, character bodies and area triggers.

Version:

Icon: res://icon.svg

Configuration is a big aspect of the Godot game engine, but can generally be learned as you go, since all configuration options will be populated with default values. In the image above you can see the very basic configuration where you can name your project, give it a description, and an icon that will show up in the project launcher.

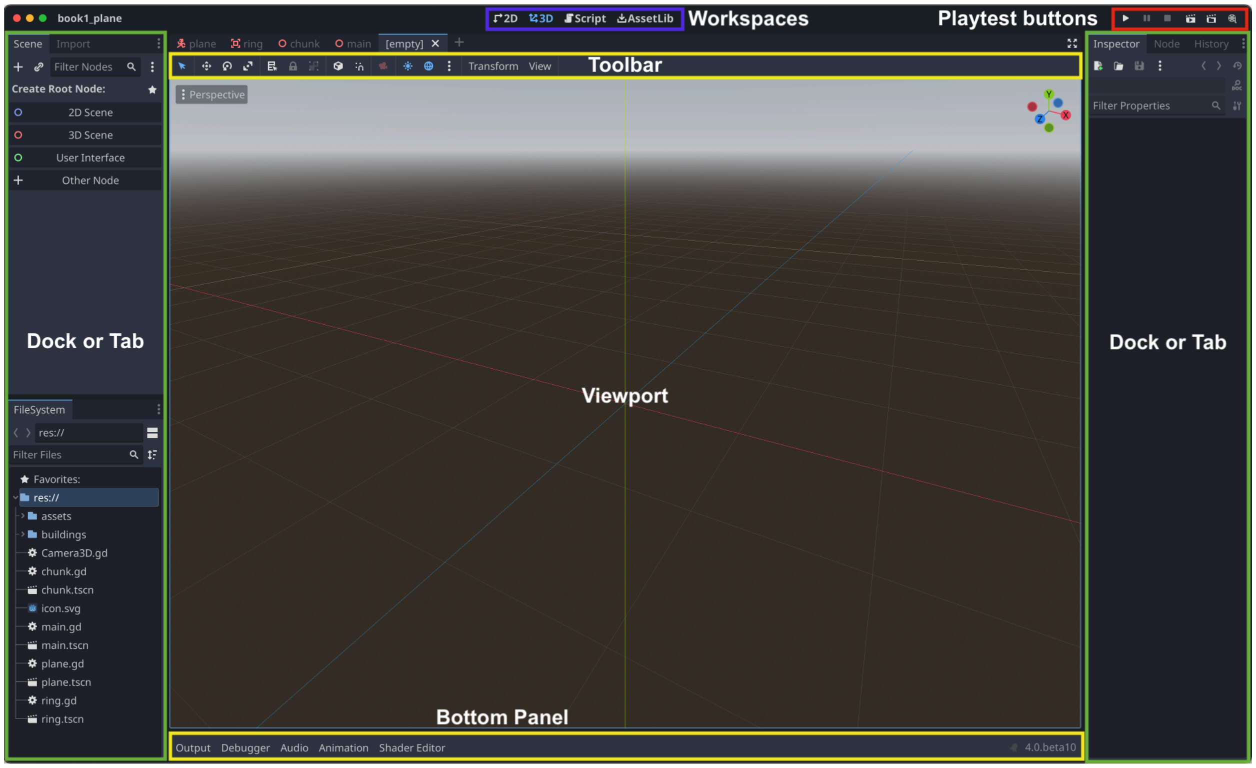
Other common things you will likely end up configuring include:

- Rendering (how to display graphics such as PNG files)
- Display (size of screen, windowing, resizing etc.)
- Input Map (support for game controllers)
- Autoload (global settings and code scripts)
- Plugins (extensions to the core engine)

For now we can use the default settings, and we will push on to learning about the areas of the interface. Following that we can jump into learning about the game objects - features common to all video game development.

Learning The Interface

Once you have a new project you'll be greeted by the interface, which is divided into categories:



<u>Left Dock:</u>
Includes the Scene Tree and Importer, as well as the FileSystem.

The left dock is all about seeing all of the game assets available to your game, whether they are currently in use or awaiting use.

<u>Center Viewport and Toolbar:</u>
Includes the toolbar, scene tabs, viewport, and bottom panel.

The center area is used to view your current game scene. Every project must have at least one scene called the “main scene” or “world scene”, and most games will have many more scenes, each representing some aspect of the game - a player, an NPC, a treasure or trap, etc.

In the bottom panel you'll find tools that let you monitor what is happening in your game, and depending on what kind of assets you are working with you may find extra tools appear that let you do things like control animations.

<u>Right Dock:</u>
Includes the Inspector, Node list, and History

The right dock is all about examining individual assets being used in your game world, and setting their specific settings.

Workspaces (top):
Includes 2D, 3D, Script, AssetLib

The workspaces selector allows you to easily switch your viewport between a 2D view, a 3D view, a code editor, and the Godot Asset Store. This is an often used part of the UI since most games will use a combination of 2D, 3D, and code.

Playtest Buttons (top right):
Includes play, pause, stop, play current scene, play specific scene, and movie-maker mode

The first three controls play the main or default scene so you can watch and interact with your game as a player would. The other play buttons allow you to play a specific scene - this is important because Godot games are made up of multiple scenes. For example the default scene would be the game world you are developing, but nested within that scene can be many other scenes, each representing a part of your game you would like to work on in isolation. For example the character - maybe they need to jump, run, fight, etc. and you would like to work on all of that outside of the crowded main game. In that case you could make the character it's own scene, and test/play just that scene to see how your character looks.

Tip: Besides your main game, and your character scene, set yourself up with a very basic scene that is not your full game, but that you can use for testing your scenes as you develop them. For example to test your character you could create a scene that includes a dead-simple floor and some walls that you can bring your player character into to test jumping/running/fighting etc.

By setting up a 'test room' like this you will not need to launch your full game with all of it's graphics, sounds, lighting etc. when all you need is to test some features in isolation.

Nodes and Scenes (Composition)

From our quick review of the interface, you'll have noticed that a fair amount of the editor is set up for managing "scenes", and you'll quickly learn that scenes are composed of "nodes".

IMPORTANT: Scenes and Nodes are a core concept of Godot

Scenes are reusable units that group together all of the functionality for an important element of your game. How much you choose to include in a scene is really up to you, but other than your main game scene they usually benefit from being simple.

For example in a racing game your car might be considered a scene. Why? Likely you will want to do quite a lot of work on the car and it is helpful to be able to do this in isolation. You will want to encapsulate how the car looks, how it reacts to physics and user input, what sounds the tires make, and possibly much more.

In that scenario, the “car” scene could be developed on its own, then composed into the main game scene, that adds on things like lighting and environment for the car.

In our example, you would develop the car scene by using “nodes”.

In Godot, nodes are discrete reusable bits of functionality intended to be used across many scenes for describing the features of the scene. Because nodes are the essential building blocks of a game, they get grouped into categories based on their intended purpose. Some are for showing graphics, others relate to the physics engine, lighting, UI, networking functions and anything else you could think of. Some very common nodes you will use include:

Sprite nodes - allow you to show an image in the scene
Animated sprite nodes - show sequences of images
Collision Shape nodes - detect intersection with other objects
Control nodes - show user interface, accept input

A very common “recipe” for a scene is to include this mix of nodes

- A 2D or 3D node that includes a “transform” - information about the position, size, and scale of an object
- A sprite or mesh node that shows a game asset like an image or 3D model
- A collider node that interacts with the physics engine to notice collisions
- A script that defines the behaviour of the scene using code for decision-making

In that recipe, each node in the scene has a specific job to do. The 2DNode positions the object in space and keeps track of where it is positioned.

The sprite or mesh node provides a visual representation of the game object - the way that it looks.

The collider node defines a simplified physical form for the game object, and is capable of interacting with the physics engine to notice collisions with other objects.

A code script gets updated by the nodes composed into the scene and can make simple decisions about how the scene should behave. For example if the collider node notes a collision, the script might react by updating the position of the 2DNode - thus achieving a reaction to the collision.

Many game objects (scenes) working together form a game, with all of its many components all working together to define the gameplay.

Big Concept: Local and Global Position

If every visible element on screen in our game world is a “scene”, the position of that scene, along with its size and scale (a multiplier of the size) are held in perhaps the most important property a node can have - the “transform” property.

We have seen that scenes are basically collections of nodes. The way those nodes are organized is with a master or “parent” node that acts as a holder for all the other nodes needed to create the scene.

That “parent” node is typically a 2DNode or 3DNode that has a very important property - the transform property. It also has properties for visibility, and whether the node is enabled or disabled.

On its own, the 2D or 3D node has no ability to show any graphics or detect collisions so these abilities are added by composing other nodes into the parent node. As an example the Sprite2D node could get put inside the 2DNode to create a “scene” that can show an image. The Sprite2D would then be called a “child” node of the 2DNode.

In that scenario, it is important to understand that the parent node has a transform - but so does the child node. Because the parent resides at the root of the game, its transform contains what is called the “global coordinates”. But the sprite nested inside has its own transform as well that you can see in the inspector, and this transform contains coordinates relative to the origin position of the parent. This is called “local coordinates”.

BEWARE: It is a common mistake to move objects in the wrong coordinate system, with very unexpected results.

For example, if we were to try to move our image to another location on the screen, we might very well end up moving the child node (the sprite) rather than the parent node (the 2DNode). But we would be moving the child in the local coordinate system, even though we intended to move it in the global coordinate system.

Visually this may appear correct - until we attempt to rotate the sprite, or it reacts to the physics engine, or any number of other scenarios. At that time we would find that the sprite is now off is axis and it will end up moving very unpredictably in global coordinates.

Tip: Use the “group” option to lock child-nodes relative to their parent.

Keep grouping in mind when composing scenes, it will save you a lot of frustration.

And with that tip in hand, we are free to move on to the most important nodes you will use making a game in Godot, and how you will compose them into scenes.

The Four Nodes of the (fun) Apocalypse

Of all the many nodes available in Godot, there are four that are the most universal in terms of what they do. So common are they that they are probably the first nodes you should learn. They include static bodies, rigid bodies, character bodies, and area bodies.

These are the core building blocks of your game, and are available for both 2D and 3D games.

These are core behavioral nodes, into which you will stuff all of our graphics, colliders, sounds and so forth.

The main purpose of these “big four” nodes is to determine behavior relative to the physics engine.

Why do we need four nodes related to physics? Shouldn’t everything in the game world rely on physics just like in the real world? In many cases - no. Think about how many times that would ruin your favorite games. Sonic the Hedgehog would splatter against the first loop he ran into moving at those speeds. Characters falling from three times their height would suffer crippling damage.

No, we play games to have fun, and in many cases that means bending the rules of physics.

In general terms, physics potential for movement is imparted by either a force, or an impulse.

Force - a constant motivator (think gravity, wind, current)

Impulse - a temporary motivator, delivered as a pulse (think a kick, shove, shot, explosion)

The big four allow us to be the laws of fixes regarding imparting forces and impulses in the following ways.

Static Body

A static body allows physics to register interactions, but does not allow the physics engine to employ a reaction to those interactions. Their transform can not be affected by collision with other bodies.

Uses: these are excellent bodies to use for building out the game world. Floors, walls, ledges, platforms are all good candidates to build using static bodies.

Exercise 1: Create a floor boundary object for our game world
Exercise 2: Create a reusable block for walls, platforms, etc.

Rigid Body

The rigid body IS affected by the physics engine, both in terms of detecting collisions and reacting to them. Rigid bodies transforms can be affected by collisions that impart either a force or an impulse to them.

These bodies take into account a wide variety of factors in interpreting physics, including their mass, and even properties related to their materials such as friction.

Uses: bullets that ricochet, debris, interactable objects

Exercise 1: Create a “crate” that conforms to physics
Exercise 2: Make the crate its own scene, and duplicate it to produce more crates

Character Body

Character bodies, as their name implies, are used for players and often for non-player characters such as enemies. Character bodies, like static bodies, do not react to physics, though they can trigger collisions. Think of them like feather-light objects moving through the game world, unable to push anything in their way.

Since characters do need to react to the game world and interact with things around them, those aspects must be created using code. And this is the point of a character body - to give the game designer ultimate control of how the character affects and is affected by the environment.

Character bodies do include some very helpful functions to assist the developer in crafting these interactions, such as the “move_and_slide” function (helpful for moving on sloped surfaces) and “is_on_floor” (for knowing if a character can jump).

Uses: player characters, enemies, vehicles

Exercise 1: Use a default script template to give movement to a player object
Exercise 2: Use code to impart a physics impulse to crates (rigid bodies) as the player moves around
Optional: Improve upon the default movement template

Areas

Areas are different than other bodies in that they are not usually intended to have any visible graphics associated with them. They need only a collider and they can then detect collisions, and even the state of the collisions i.e. detecting when a body has entered an area, or left it.

Areas are your means for detecting collision with the other body types and are usually handled by the logic of the game. These are used to trigger decisions when another body enters an area. This might be anything from playing music, to springing a trap, to unlocking a secret.

Uses: triggers, traps, objectives
Exercise 1: Create a trigger area that can detect collision with a)the player, and b)the crates
Exercise 2: Use a custom signal to alert the game engine to interactions