Moving from Unity to Godot

An In-Depth Handbook to Godot for Unity Users

Alan Thorn

Apress®

Moving from Unity to Godot

An In-Depth Handbook to Godot for Unity Users

Alan Thorn

Apress®

Moving from Unity to Godot: An In-Depth Handbook to Godot for Unity Users

Alan Thorn
High Wycombe, UK

ISBN-13 (pbk): 978-1-4842-5907-8 ISBN-13 (electronic): 978-1-4842-5908-5
https://doi.org/10.1007/978-1-4842-5908-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5907-8. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Alan Thorn is an expert on leading technical teams for game development. He previously worked at Microsoft, Teesside University, Apress Publishing, and Disney. Alan specializes in helping "tech heads" thrive and flourish in their chosen fields. With 18-year game industry experience, Alan has written 28 books, presented 30 online courses, and created 33 games including the award-winning adventure *Baron Wittard: Nemesis of Ragnarok*. Alan is dedicated to helping creative people make high-impact experiences. He was a Studio Director at Wax Lyrical Games and a Senior Author at LinkedIn Learning, and now he leads the prestigious MA program for Games Design and Development at the BAFTA-winning National Film and Television School, an incubation space for breakthrough gaming talent. Alan is a pioneer of the new "Open Stream" model of Transformative Learning, and he advises in higher education on disruptive curriculum content and instructional design. Alan speaks passionately worldwide about the future of interactive experiences. In this book, he clearly details Godot-specific terminology, how to use its interface effectively, how scenes are structured, coding in C#, and optimal ways of working.

About the Technical Reviewer

Doug Holland is a Software Architect at Microsoft Corporation in the One Commercial Partner team. In his role at Microsoft, Doug provides guidance to Microsoft's partners looking to digitally transform with cloud computing, mixed reality, and other emerging technologies. Doug holds a master's degree from Oxford University in Software Engineering and lives in Northern California with his wife and children.

Introduction

Congratulations on joining me to take an exciting journey, one that moves from Unity to the Godot engine. Godot is a completely free, open source, and cross-platform engine that's making amazing strides and progress. Godot can make 2D and 3D games, and it can be used for other products like visualizations, Movie Previz, historical recreations, and more. In the competitive climate of game development, where new tools and technologies are continually reshaping the landscape, and challenging established norms, it can be difficult knowing which tools to learn or to trust in. Every software, it seems, has its rise and fall. It's difficult knowing which one represents the actual future and who is just a passing fad. But your decisions about which software to use are crucial for your business and your success. Your choice of engine influences *your* future. It influences how quickly and easily you can work, which platforms your end product can target, and which tools you can integrate with.

One reason so many people today are switching from Unity to Godot, or at least considering the switch, is because Godot represents a vision and a promise, a free engine that's open source and community driven, an engine that's easy to use and reliable, and an engine that's open about its development road map.

One of the historic fears surrounding the use of any game engine – as opposed to making your own engine – was its closed and proprietary nature, that it would keep you locked into specific ecosystems and to specific agendas led by companies and parties outside of your own business. And until recently, you never really had much choice if you wanted to make a professional-grade game. You had to lock yourself to a

closed engine driven by people behind closed doors. But Godot changes that, and that's why it's an especially important engine today. It represents an opportunity to make games on very different terms. And with the outstanding success of other open source tools, like Blender, we have every reason to feel excited for the future of Godot. I'm truly glad you're going to join me on this journey.

CHAPTER 1

Introducing Godot: Why Migrate?

This book helps you convert easily from Unity to Godot. It assumes you're already familiar with Unity (at least the basics) but are completely new to Godot. This book translates Unity terminology to Godot terminology. Plus, it features comprehensive tutorials and guides to get you started quickly in Godot. Godot is a completely free and open source game engine that's growing rapidly in popularity, acceptance, and adoption, especially for independent games. More and more developers worldwide are happily joining the Godot community to make great games together, and there's never been a better time to start learning. In this chapter, we explore what Godot is, how to download and install it, and strong reasons why we should use it compared to other engines, like Unity or Unreal. So let's get started.

This book was written for the Godot 3 release cycle. This includes 3.0, 3.1, and later releases in the 3 cycle. You may be using a later release than the one featured in this book; but most of the content presented here should be good for the near future.

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5_1

Getting and Installing Godot

If you're using Unity, you probably already know what Godot is! Godot is a game engine. It's software for making games and interactive experiences. It features a world creator, a graphical interface, a code editor, a complete API, and build tools for deploying your game to different platforms as a stand-alone application. In short, imagine a game engine that's very similar to Unity in its ease of use, but it's completely free and open source. That's Godot.

You can download Godot from its home page https://godotengine.org/. Simply navigate a browser there and click the *Download* button from the top menu. This takes you to the Download page. From there, be sure to download the *Godot Mono Version C#*. See Figure 1-1.

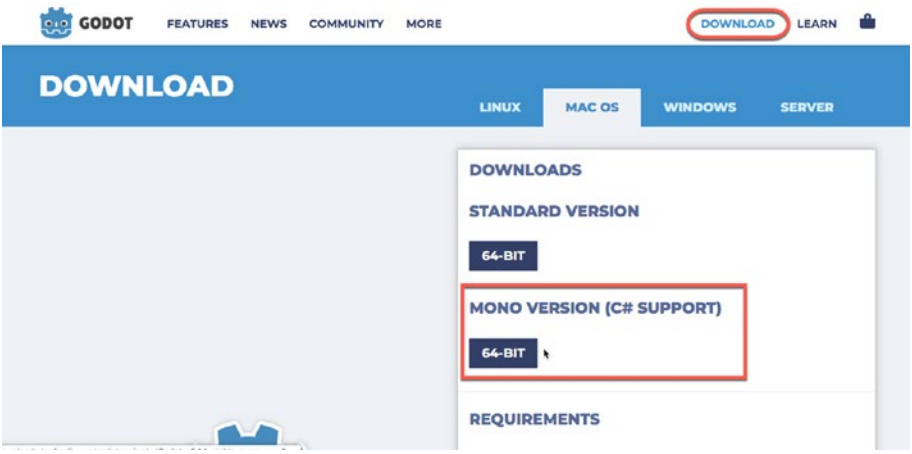

Figure 1-1. *Download and Install the C# Godot Version for Your Operating System*

Multiple versions of Godot are available for download, all for multiple operating systems. Specifically, Godot natively supports Windows, Mac, and Linux; and it's available in a *Standard Version* and a *Mono Version*. The Mono Version lets you make script files in the C# scripting

language – like Unity – while the Standard Version supports Godot's custom language, GDScript, only. This book uses the Mono Version and writes script files in C#. Using C# not only makes the transition from Unity to Godot simpler; it's also a great language offering premium level runtime performance. So I recommend choosing the Mono Version in all cases. Godot features full support for C# 8.

Why Use Godot?

So why use Godot at all? That's a good question! After all, there're tons of apparently free and spectacular game engines available today. For example, you can download *Unity*, *Unreal*, or *Lumberyard* right now and make games with them. These engines are well documented, widely used, and are responsible for many well-loved games on the market. So why use Godot instead? This section lists important reasons for choosing Godot as opposed to its alternatives.

Godot Is Free

Godot is completely free of cost. You pay nothing for using Godot or for selling your Godot games. Really. This is actually very different from Unity and Unreal even though many people think they're fully free. Unity and Unreal are, in fact, not completely free. True, you may download and use them *free of charge*, but you'll need to pay up if you release a game commercially and its revenue surpasses a specific threshold. The amount differs per engine. But let's put this into perspective as of November 2019. For Unity, if your company earns more than *100,000USD* in a single financial year, then you'll need to purchase a Unity Professional License. This costs *125USD* per month. See the Unity FAQ page here: `https://unity3d.com/unity/faq/2491`. In future, this pricing structure could increase! Similarly, for Unreal Games, you'll need to pay 5% of your gross

revenue (before tax) after you earn 3,000USD (see www.unrealengine.com/en-US/faq). This could amass to a huge sum if your game becomes successful! So, neither Unity nor Unreal is fully free. But Godot is. You pay nothing for using Godot, ever. Simple.

Godot Is Open Source

Godot is completely open source. This means you can download, inspect, edit, and compile its source code immediately. Plus, you can use and redistribute any derived versions you may make. Open source should be reassuring to a game developer. This is because their work won't depend on a closed foundation of proprietary code that's maintained by a third-party developer whose agenda may be very different to yours. The engine is licensed under the MIT License, which is summarized by Godot:

> *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.*
>
> ***—Godot website (https://docs.godotengine.org/en/3.1/about/faq.html)***

This book doesn't cover compiling Godot from source. It assumes you'll be using a fully built version that's downloadable from the Godot website. For more information on compiling from source, visit the Development website here: https://docs.godotengine.org/en/latest/development/compiling/getting_source.html.

Godot Is Evolving

Godot is always evolving. It was established in 2014 by software developers Juan Linietsky and Ariel Manzur, and it's continued to grow in popularity and features year after year. Godot has a full range of features that any Unity developer will recognize and expect from an engine, plus really interesting features that may pleasantly surprise you. Godot supports Light Baking, full Global Illumination, Navigation Meshes and Path Finding, Visual Scripting, Constructive Solid Geometry (CSG), and 2D functionality for sprites and User Interfaces. Godot currently supports two scripting languages, namely, GDScript and C#; the former is more established in the engine, while the latter is newer. Godot is actively maintained by a strong development community, and many people financially support the project through *Patreon*, here: `www.patreon.com/godotengine`.

Godot Is Supported

Godot is strongly supported by a growing community of game developers, educators, and creative evangelists. Together they have a ton of experience. They have a shared purpose in promoting Godot, helping each other, and helping newcomers learn the tools. Online documentation, books like this, videos, and video courses are all available to help you learn Godot from scratch and to continue using Godot professionally. Godot has comprehensive online documentation that should be used frequently as a reference, in conjunction with this book, as you progress chapter by chapter. It's available here: `https://docs.godotengine.org`. Furthermore, you'll also want to check out `https://godotsharp.net` for a C#-oriented reference and also my YouTube channel BeIndie.biz (`www.youtube.com/channel/UCF1X3sTIj-pCIcR_C2wg6SQ/`) for regular tutorials on Godot.

Godot Is About Games and Experiences

If you attend game conferences, you'll find a growing community of game developers discontented with the strategic direction of both Unity and Unreal right now. These developers are seeking alternative options, and understandably so. There're several important reasons for this search worth mentioning here. Some consider Unreal too expensive and technically burdensome, especially for small development teams with less specialist needs. Some see Unity as focusing too heavily on architectural and automotive visualization features at the expense of games. And some feel that both engines have just gotten too wrapped up in the arms race toward photorealism, the sense that both are "growing too big to fully understand the needs of independent developers and small teams." There's a sense, either real or perceived, that in both Unreal's and Unity's most recent growth (especially in the latter), there's also a significant detachment from their original user base. In this search, then, many people are now finding their ideal in the Godot engine – driven by and built by the game development community. Godot is first and foremost a *game engine*. Its structure, tools, feature set, and build options have games (and interactive experiences) in mind from the outset. Its source code is open and its development road map are constantly under an open discussion by the developer community. The full development road map can be found on GitHub here: `https://github.com/godotengine/godot-roadmap`.

Godot and C#

As a Unity developer, you'll be familiar with *C#* for scripting – unless you use a Visual Scripting plug-in like *Playmaker*. C# is a versatile, powerful, and easy-to-learn language that delivers comparatively excellent runtime performance. And so it's likely you'll want to continue using it on migrating

to Godot. After all, you probably don't want to learn a completely new language and syntax just because you're moving to a new engine. Thankfully, you can bring your C# knowledge with you! This is because the Godot 3.1 Mono Build (or higher) supports C# as a scripting language. However, there are some important limitations to using C# that you should know. These limitations may be ironed out in later releases, but as of 3.1 they apply:

1. You can't build C# projects for the Web or mobile devices.

   Godot lets you build games for many platforms, including desktop, Web, and mobile – using the GDScript language. However, C# projects build only for desktop systems – both Web and mobile are currently excluded. C# support may be added soon, but this is in development.

You can keep track of the development status for these features, as follows:

Android

https://github.com/godotengine/godot/issues/20267

iOS

https://github.com/godotengine/godot/issues/20268

HTML5

https://github.com/godotengine/godot/issues/20270

---

2. You won't get C# code completion or syntax highlighting for the native script editor.

   Unlike Unity, Godot provides an integrated text editor for creating scripts as part of the main editor interface; see Figure 1-2. This is the script editor window. You can use this window to type C# script files, but it lacks full language support. It's designed for GDScript. So you won't get full syntax highlighting or code completion or any coding assistance for C#. To get this, you'll need to use external code editor! This book will show you how to use **Visual Studio Code** with Godot – complete with syntax highlighting and code completion. Visual Studio Code is supported on Windows, Linux, and macOS.

3. It's easy to break the metadata for your C# project.

   As you add and remove files to and from your project, Godot maintains metadata. This tracks connections between files and resources, and it maintains lists of script files needed for compiling. In C# projects, it's currently easy for anybody to accidentally break the metadata. This causes compilation to fail. Don't worry though. These issues can usually be fixed easily and manually, and we'll see how soon.

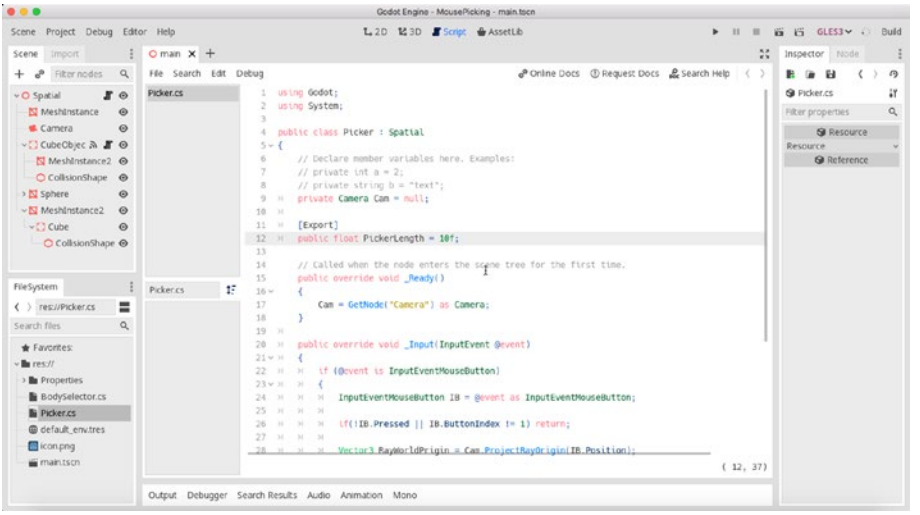

Figure 1-2. *The Godot Editor Features a Scripting Window for GDScript and C#. However, Third-Party Editors Are Better for C#, As We'll See*

Getting Started with C# in Godot

So you've finally downloaded and installed the Godot Mono Version for your OS, as mentioned earlier in this chapter. Great! Let's now take the next steps to configure Godot for C# scripting. Once set up, we'll be ready to start making games. Open up a web browser and download Visual Studio Code to your computer. See Figure 1-3. The URL is `https://code.visualstudio.com/`.

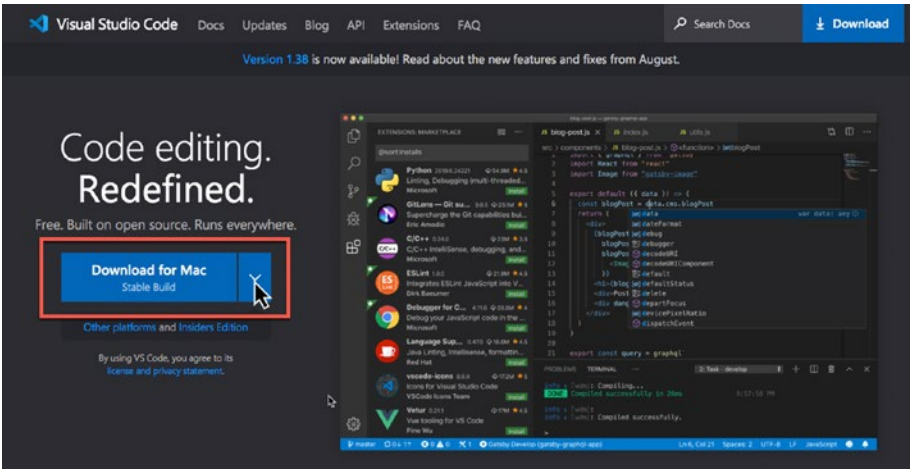

Figure 1-3. *Downloading Visual Studio Code for Your Operating System. This Book Supports All Platforms for Godot, Although I'm Using a Mac for Screenshots and Examples*

After running Visual Studio Code for the first time, select the Extensions tab and install the *OmniSharp C# Extension* for syntax highlighting and code completion support. See Figure 1-4.

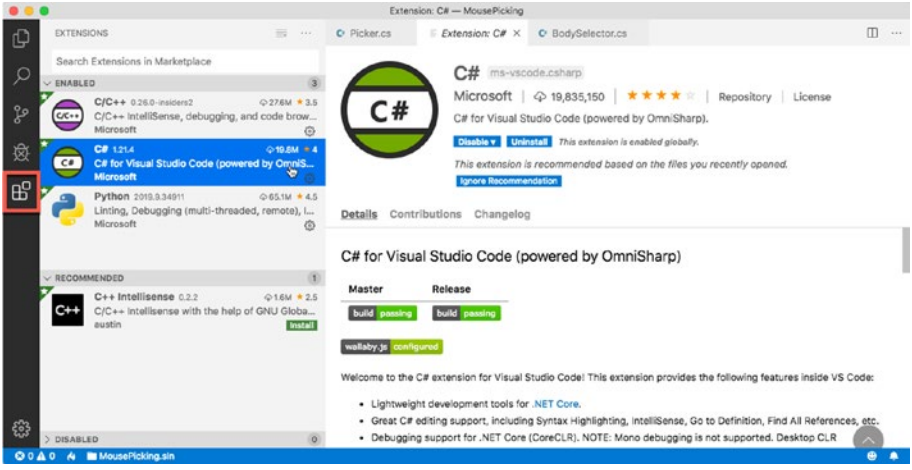

Figure 1-4. *Installing the C# OmniSharp Extension for Visual Studio Code*

Next, Open Godot and create an empty project from the Project Creation Window. Click the **New Project** button and enter your project name and location. You can name your project anything and store it anywhere. We only need to create a new project to access the Editor Settings window from the main interface. See Figure 1-5.

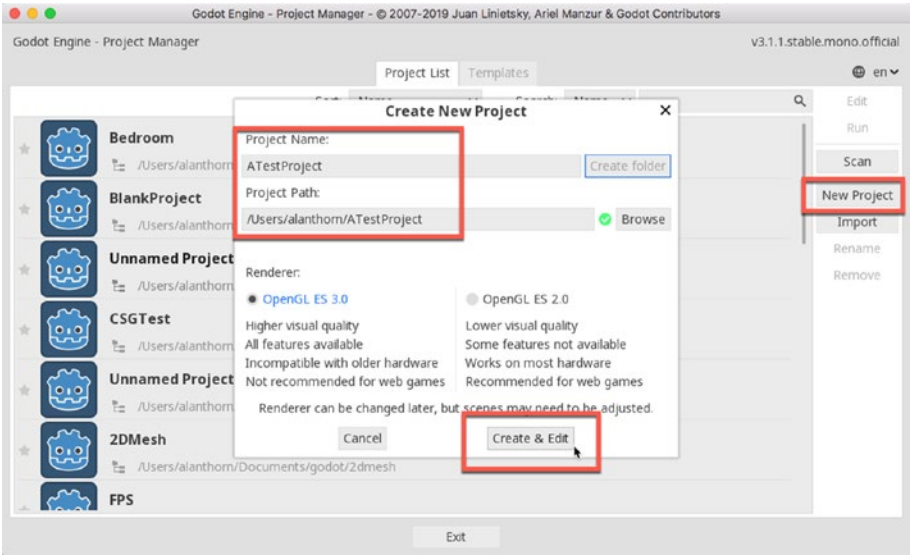

Figure 1-5. *Creating a New Godot Project*

From the main Interface, select the menu item *Editor ➤ Editor Settings*. This displays the Editor dialog where you can customize application settings. See Figure 1-6.

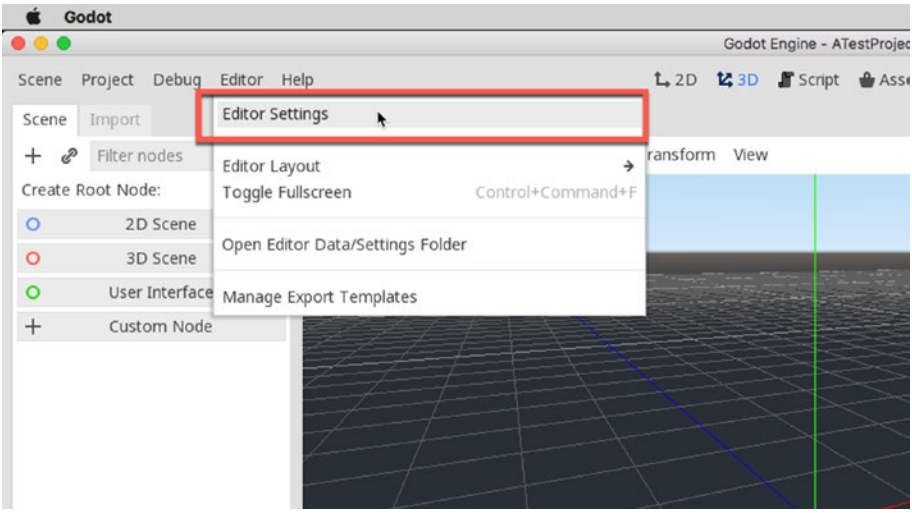

Figure 1-6. *Accessing the Editor Settings Window*

From the **Editor Settings** Window, choose *Mono* ➤ *Editor* from the list view and then select Visual Studio Code from the **External Editor** drop-down menu. This sets Visual Studio Code as the default code editor for Godot. This means that Visual Studio Code will open automatically whenever you open script files from the Godot **File System** (Project Panel). See Figure 1-7.

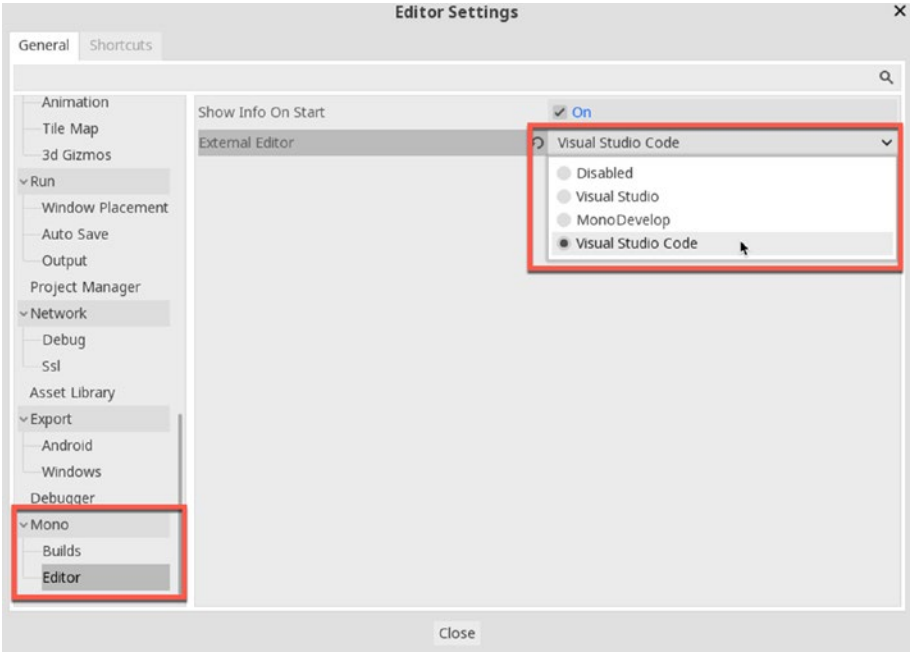

Figure 1-7. *Setting the Default Code Editor*

Excellent! You've now configured Visual Studio Code as the default code editor for Godot. This editor will now open to display script files in your project. You're now all set to get started!

You can view a free YouTube video tutorial on configuring Visual Studio Code on my BeIndie.Biz channel here: `www.youtube.com/watch?v=ra-BJ-fJ6Qo&t`.

Summary

On reaching this point, you've now downloaded and installed Godot Mono, and you're ready to start coding games using C# for any desktop platform – Windows, Linux, or Mac. Godot is an excellent, free, and open source engine; and by using this tool, you can create great products. Now let's jump in and learn Godot quickly. I'll be converting your Unity knowledge to Godot knowledge. Let's go!

CHAPTER 2

Godot Fundamentals

So you've now installed Godot and Visual Studio Code. That's great! Chapter 1 explained how to configure Godot on your computer – Windows, Mac, or Linux – and how to create Godot projects successfully. This chapter picks up from the previous to get you started quickly. Specifically, it's a high-powered conversion course. It helps you move from the already familiar world of Unity into the new world Godot. In this chapter, you'll learn how to create Godot projects, how to use the editor interface, how to convert from Unity's language and terminology into Godot's, and how to build scenes with basic game functionality. If you've never used Godot before, then you absolutely need to read this chapter. So let's go.

Note Even if you've used Godot before, it's strongly recommended that you read this chapter. It covers Godot's most core features, comparing them to Unity's and also making important distinctions that you need to know about.

Unity comes with a whole bunch of names and words you'll already know, such as *GameObject*, *Component*, *Scene*, *Script*, *Asset*, *Hierarchy*, *Inspector*, and more. Godot unsurprisingly has exactly the same concepts and ideas! But it sometimes uses different names for them. Let's start then by exploring the names we'll be encountering extensively throughout this chapter. Table 2-1 lists how Unity's names and terms compare to Godot's. Later, we'll see more in-depth explanations for them and also make important distinctions.

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5_2

Table 2-1. *Terminology: Unity vs. Godot*

Unity	Godot
Project	Project
Scene	Scene
GameObject	Node
Prefab	Scene
Hierarchy	Scene Tree
Project Panel	File System
Inspector	Inspector
Empty Object	Spatial Node
Asset	Resource
Tag	Group

Godot Projects

The first step in working with Godot is to create a new **Project**, just like working with Unity. In Godot, "One Project" refers to "One Game" or "One Simulation," or – more generally – *one complete experience*. By creating a new Project, Godot effectively creates a folder in your computer's file system, and it will contain all necessary files for your game, including metadata, textures, meshes, sounds, animations, and more. If you ever want to share your project with other team members or friends – allowing them to open it inside Godot for editing – then you'll need to send them your complete project folder. To create a new Project, simply launch the Godot application, and you'll be presented with the Project Creation

Introduction Window. This is basically Godot's version of the **Unity Hub**. From this Window, select the **Projects** tab and click the **New Project** button from the right-hand margin. See Figure 2-1.

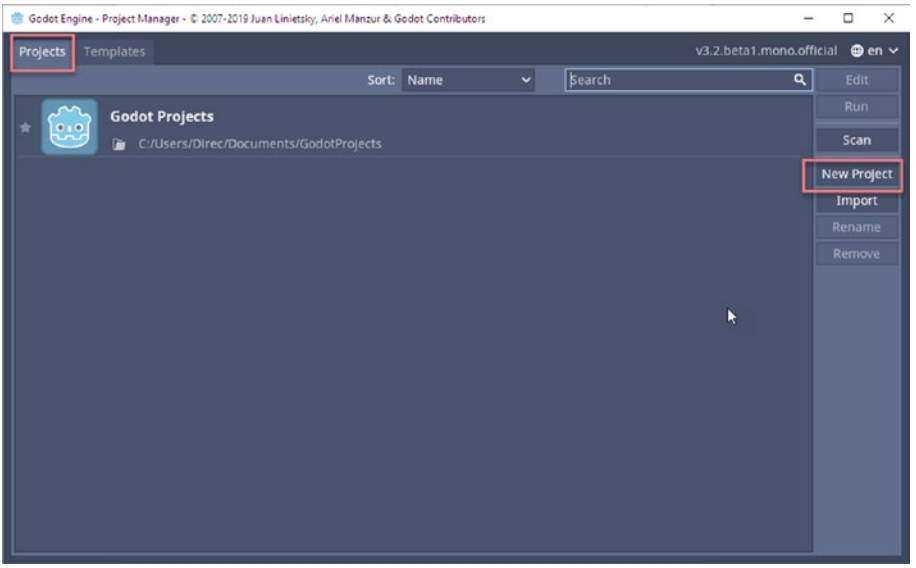

Figure 2-1. *Creating a New Godot Project*

The project creation dialog is small and contains very few options. But they're important. By default, the confirmatory **Create & Edit** button in the bottom-left corner will probably be disabled, requiring you to make some changes. See Figure 2-2.

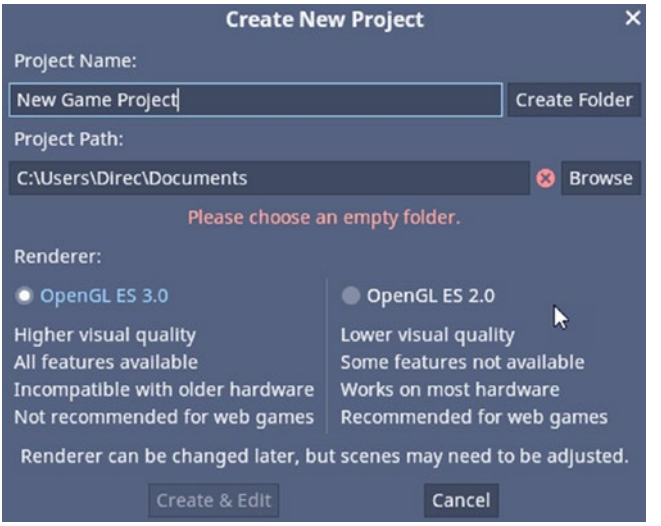

Figure 2-2. *The Godot Project Creation Window*

Specify a project name using the **Project Name** text field and then click the **Create Folder** button next to the field if the **Project Path** location is suitable for you. The Project Path specifies the folder on your computer's file system where a new sub-folder will be created, matching the Project Name. It will contain all project files and become the Project Folder. Be sure to leave the option **OpenGL ES 3.0** enabled, unless you're making a game for very old hardware, older mobile hardware, or web games. In this book, we won't be doing that! Godot supports Windows, Mac, Linux, Android, Web, and (soon) iOS. See Figure 2-3.

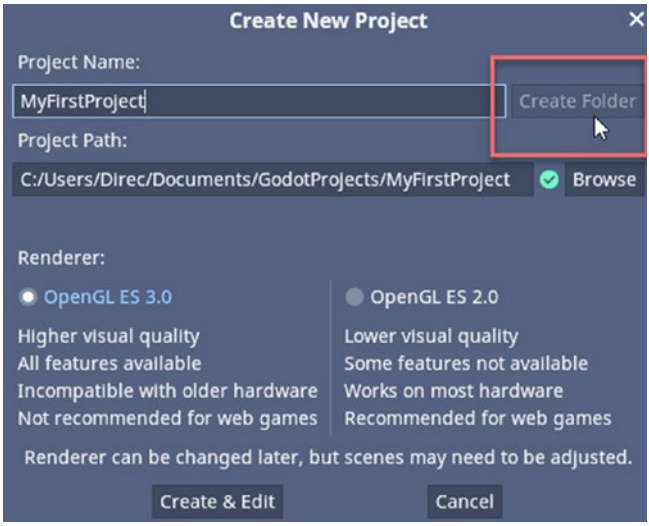

Figure 2-3. *Choosing a Project Folder*

When you're done, click the **Create & Edit** button to generate your Godot Project. When completed, Godot will open the Project inside the Editor Interface, ready for you to edit and begin development.

Note When project creation is completed, you may see a warning window titled *Important - C# Support is not feature complete*. This Window outlines some limitations to the current C# language support in Godot and emphasizes that its support is still in development. You'll probably be pleased to know that, as of Godot 3.2, C# support – while still in development – is extensive and powerful.

Editor Interface

You'll spend most of your development time using the Godot Editor Interface, building games and other projects. The Editor Interface lets you import Assets, create Scenes, and build Projects. Its default layout looks and works similarly to the Unity engine. Let's see a comparison. See Figures 2-4 and 2-5.

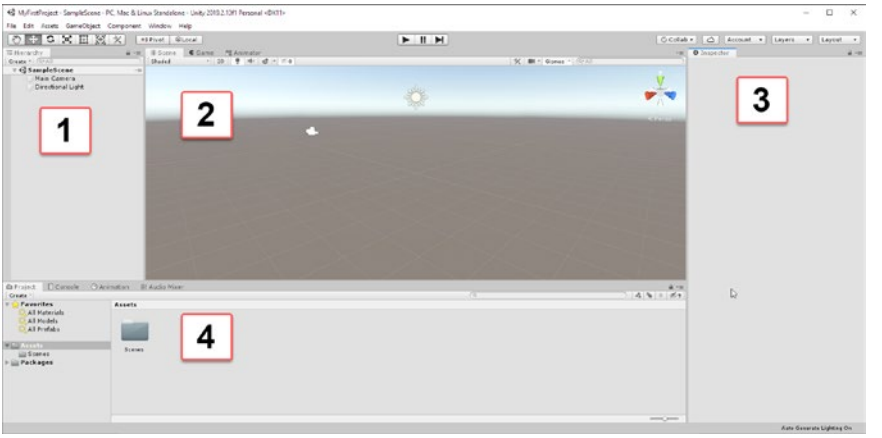

Figure 2-4. *Unity Editor Interface: (1) Hierarchy Panel, (2) Scene Tab, (3) Inspector, and (4) Project Panel*

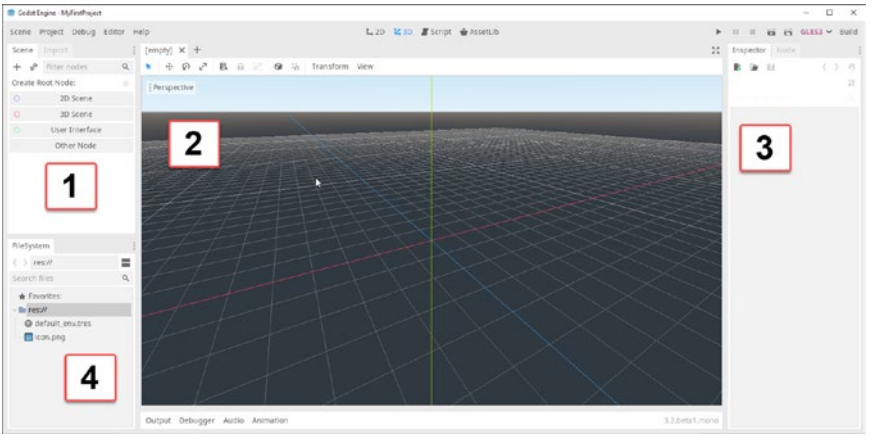

Figure 2-5. *Godot Editor Interface: (1) Scene Tree, (2) Scene Tab, (3) Inspector, and (4) File System*

Note Many screenshots included in this book feature the Godot interface in its light color theme, as opposed to the darker, default theme. You can choose the Interface theme by selecting *Editor* ➤ *Editor Settings* from the application menu to show the Settings dialog. From there, you should choose Interface ➤ Theme from the options list and set the *Theme Preset* drop-down menu to *Light*.

By default, *Unity* creates an empty scene for you in your new project. You can edit this scene through the scene tab – as you'll already know. In contrast, Godot **doesn't** create a scene automatically, even though it initially appears like a scene exists by looking at the 3D viewport. You'll see a skyline and a 3D Grid in the viewport (somewhat counter-intuitively) – making it look like a world exists. See Figure 2-5. Despite appearances, however, Godot is different from Unity. It's actually in a "standby" mode, waiting for you to open any existing scene (if you have one), to import a scene from a file or to create a new scene. We'll do the latter in the next section.

Creating and Editing Scenes

Scenes are worlds or levels. A scene is a single, unified coordinate space in which objects exist in a hierarchy, like characters, buildings, and weapons. Unity calls such objects "**GameObjects**." Godot calls them **Nodes**. Together these Nodes form a world, such as a town, a house, a space station, or any other type of location. Typically, a game consists of one or multiple Scenes. These may be presented during gameplay either successively, like when the player progress to the *next* level, or intermixed together at different times, such as when a town environment changes seamlessly to a forest during an *RPG*. In Unity, a Scene is conceived as a monotype; that is, there's literally only *one type of scene*. Consequently, your development

workflow in Unity will be to create an empty scene and then put things in it: 3D objects, UI elements, and also 2D elements as needed. Simple. You can – of course – make fully 2D games in Unity, as opposed to 3D. Naturally, 2D and 3D seem to be different coordinate spaces; but really – when you make a 2D game – your 2D objects will just live in a 3D world and they'll only be *presented* in 2D to the camera. Godot is different, however. Unlike Unity, Godot *does* divide Scenes into different *types*, specifically three types: **3D Scenes**, **2D scenes**, and **UI** (User Interface) **Scenes**. A scene's type is determined by its **Root Node type**, that is, by the type of Object at the top of the scene hierarchy. Let's try a practical example to show you what I mean.

Note I strongly believe it's a mistake to divide scenes into different types as Godot has done. I think the Unity understanding of a Scene is simpler, more elegant, and more powerful as it separates the world space from the types of objects that inhabit it, allowing you to easily and intuitively mix different kinds of objects, like 3D objects and UI elements. I hope that Godot will revise its scene conception in future releases.

To create a 3D scene, you select *Scene* ➤ *New Scene* from the File menu. This creates a new Scene tab in the interface, and the **Scene Tree** tab prompts you to generate your first, top-level **Node**. For a 3D scene, you must select **3D Scene**. See Figure 2-6.

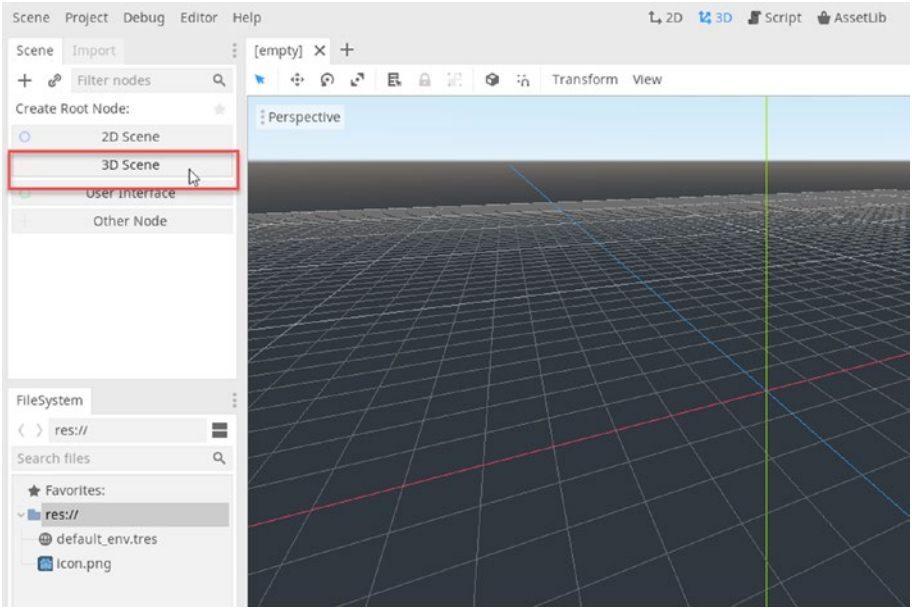

Figure 2-6. *Creating a 3D Scene Root Node*

After selecting *3D Scene* from the menu, Godot auto-generates a new Node as the topmost node in the scene hierarchy (**Tree**). This node is called a *Spatial*. A **Spatial** Node is the Godot equivalent of a 3D **Empty Game Object** from Unity. It represents a location in 3D space. It can be moved, rotated, and scaled; but it has no substance or appearance, like a mesh. The Spatial node exists in 3D space. This is why any scene with a Spatial node as the *Root* of its *Scene Tree* will always be a 3D scene. See Figure 2-7.

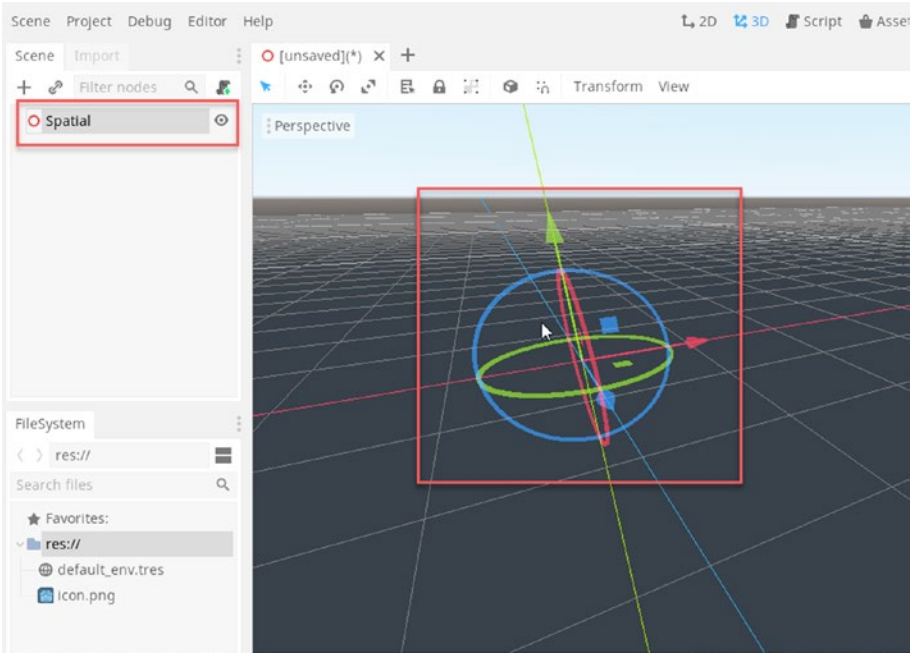

Figure 2-7. *Spatial Nodes Are at the Root of 3D Scenes*

After creating any scene, you'll need to *save* it. A scene with unsaved changes appears with an asterisk symbol (*) beside its name in the Editor. You can save the open scene by pressing *Ctrl+S* (Windows and Linux) or *Cmd+S* (Mac) or by choosing *Scene* ➤ *Save Scene* from the application menu. See Figure 2-8.

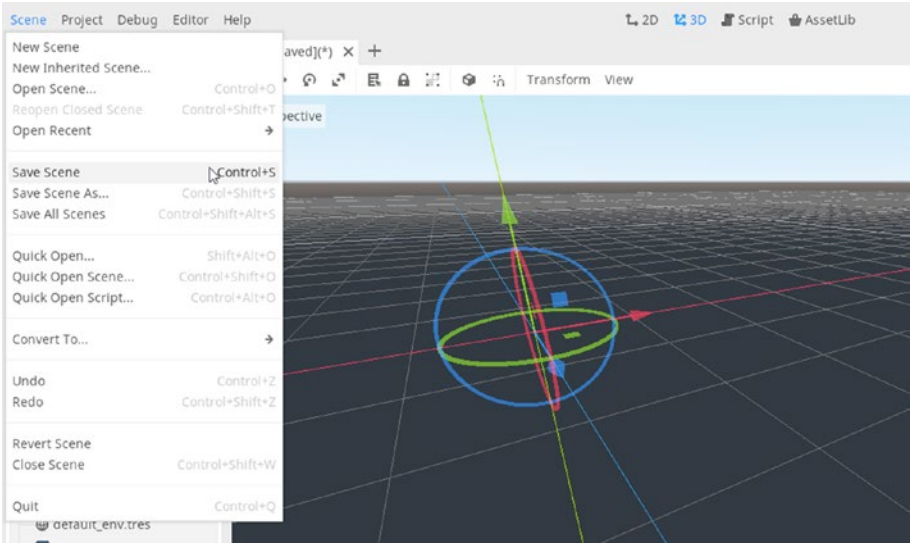

Figure 2-8. *Saving a 3D Scene*

Scenes are saved as an Asset (**Resource**) of the project and will appear in the File System panel. Each scene is saved in the **.tscn** format.

Note The tscn format defines the contents of a scene in a text file form. Godot uses this text file to construct the contents of a saved scene when shown in the viewport or during gameplay. Being a text file, scenes can be edited and modified using a text editor. This makes it useful to batch process a scene or make far-reaching edits by using Search and Replace tools. More information can be found on the tscn format at https://docs.godotengine.org/en/3.1/development/file_formats/tscn.html.

Creating a 2D scene or a UI scene consists only on generating a different root node. For 2D scenes, choose **2D Scene**; and for UI scenes, choose **User Interface**. Both 2D and UI scenes look substantially different in the viewport, as they use a 2D coordinate system. See Figure 2-9.

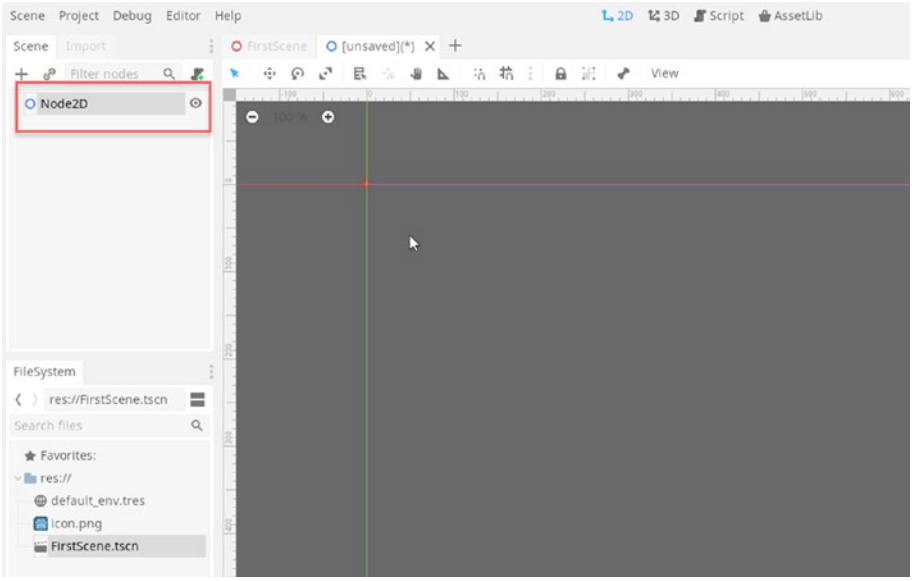

Figure 2-9. *Creating a 2D Scene*

Scenes and Nodes – Adding a Cube

So you can now create 3D, 2D, and UI scenes directly from the Godot interface. It's easy. As mentioned, Godot distinguishes between scene types, between 2D and 3D coordinate systems. In this section, we'll focus on the 3D scene for demonstration, because it's the closest type to a Unity scene. Even so, Godot Nodes (GameObjects) work the same across scenes. So everything you'll learn here for 3D scenes will be good for 2D and UI ones too. Let's explore a really basic task, specifically how to generate a basic cube mesh primitive in the scene. In Unity, that's easy to do. For example, in Unity 2019 and 2020, you just choose *GameObject* ➤ *3D Object* ➤ *Cube* from the application menu. And voila, a cube is made! See Figure 2-10. In Godot, however, the situation is different. There is no Primitive Creation menu option!

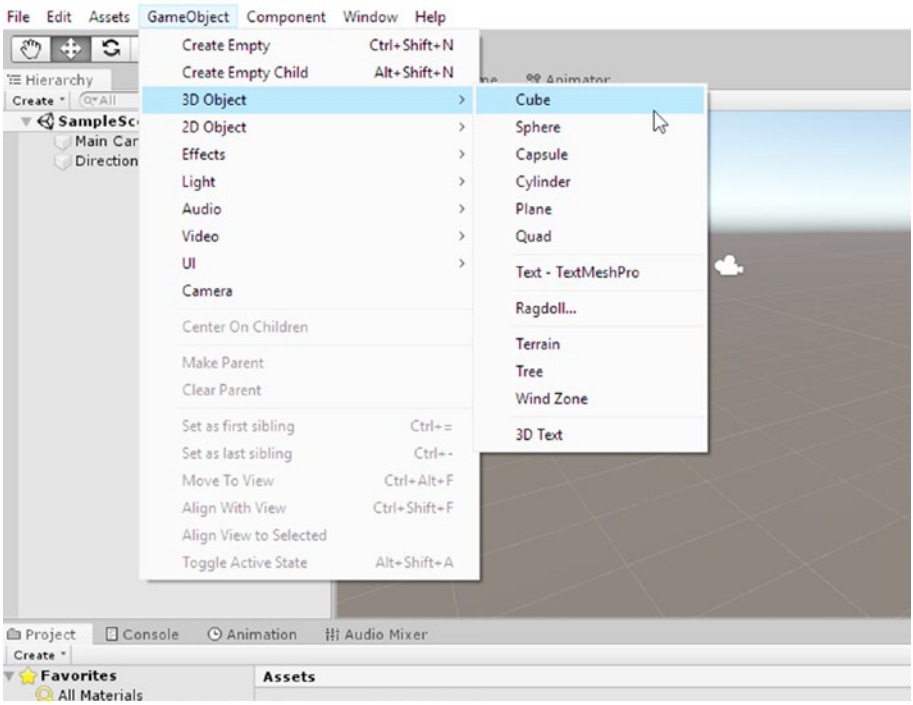

Figure 2-10. *Creating a Cube in Unity. Easy!*

In Godot, you must make mesh primitives manually. To do this, select the scene root node by left-clicking it from the **Scene Tree** panel. For a 3D scene, the root node will be a **Spatial Node**. To create a Child Node from here, right-click the Spatial Node from the Scene Tree, and select **Add Child Node** from the context menu. See Figure 2-11. This displays the **Node Creation Window**.

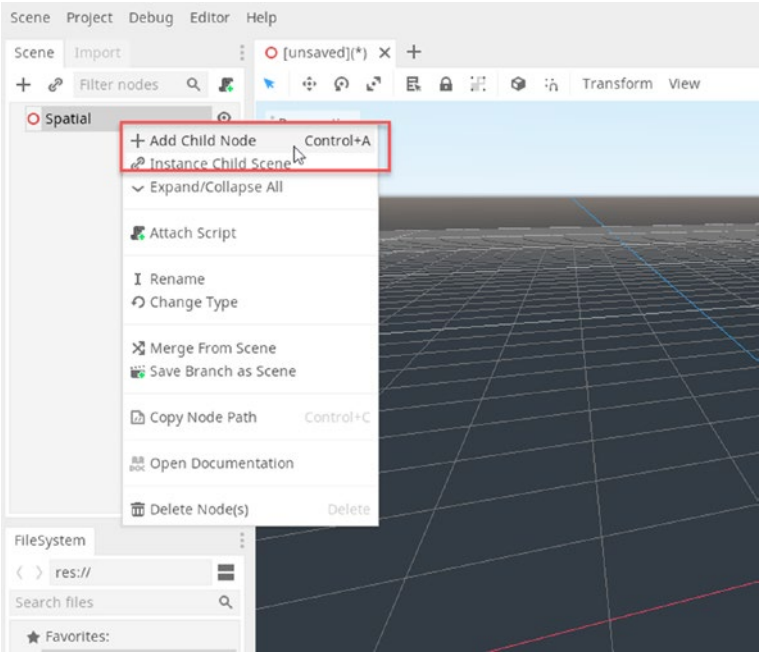

Figure 2-11. *Creating a New Node*

From the Node Creation Window, enter *MeshInstance* into the **Search** field. This applies a filter to the **Matches** list, showing you all available, matching nodes that can be added to the scene. See Figure 2-12. The *MeshInstance* node is the Godot equivalent of a Unity GameObject featuring both a *MeshRenderer* and *MeshFilter* component. It allows you to display a mesh in a 3D scene. Select the *MeshInstance* node and click the **Create** button.

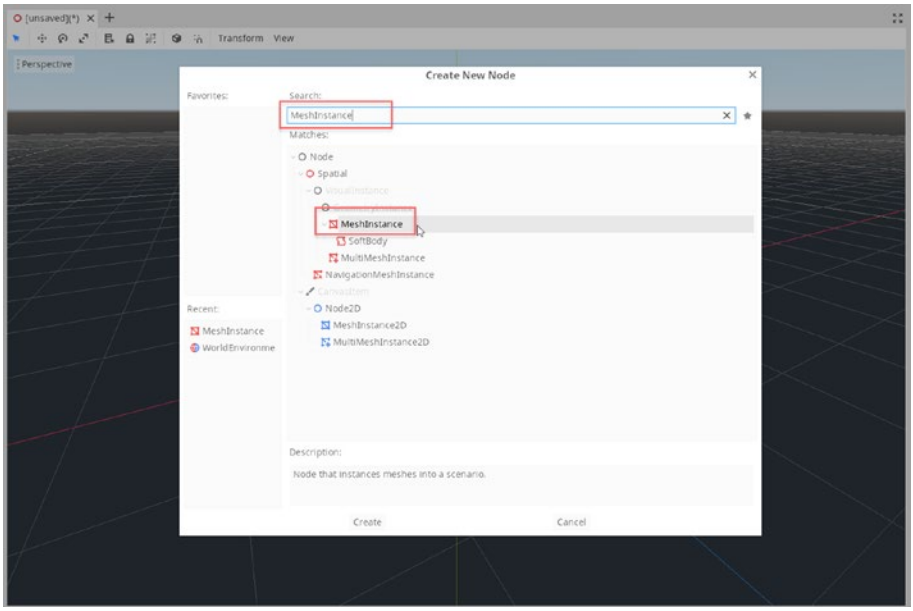

Figure 2-12. *Creating a New MeshInstance Node*

Note Pretty much anything *functional* in a scene – lighting, navigation, post-processing, and so on – must happen through Nodes. Godot is far more node driven than Unity. Unity features many *Editor Windows* as *tools* that are used to *work on* GameObjects. There is a clear distinction between a GameObject *in the world* and a tool for *editing* properties of that object. Godot, by contrast, embeds much Editor functionality into a scene through nodes, as we'll see. You create nodes by using the Node Creation Window, making this tool highly important. More information on Scenes and Nodes can be found online here: `https://docs.godotengine.org/en/3.1/getting_started/step_by_step/scenes_and_nodes.html`.

After adding a *MeshInstance* node to the scene, it'll appear as a Child of the previously selected Node in the Scene Tree. If the MeshInstance gets created as a child of an unintended node, then you can fix this by simply dragging and dropping the MeshInstance onto another node inside the *Scene Tree Panel*. *This* creates a parent-child connection between the nodes. Dragging and dropping Node X to Node Y will always make Node X a child of Node Y. See Figure 2-13 where a *MeshInstance* node is a child of the Scene Root object, a Spatial Node.

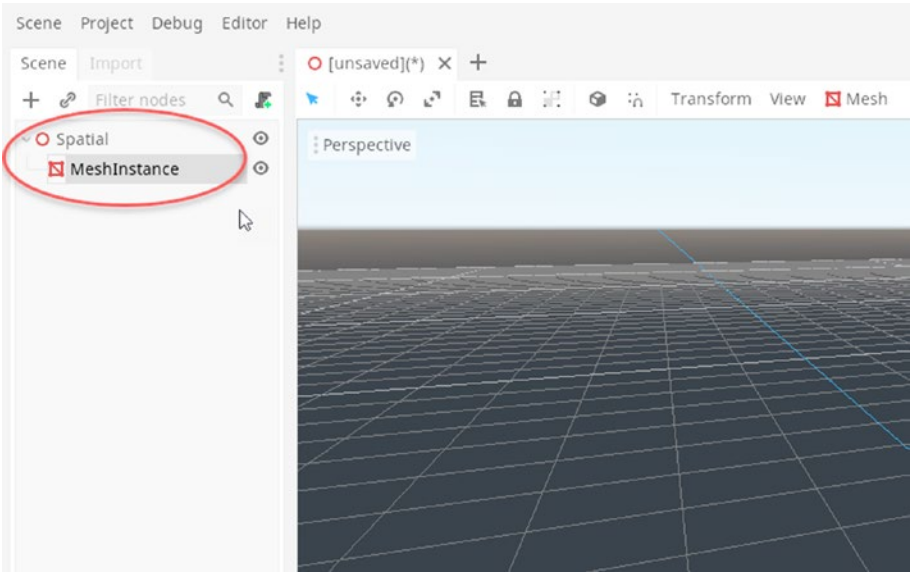

Figure 2-13. *Adding a MeshInstance As a Child Node of a Spatial*

MeshInstance nodes are used for displaying meshes in 3D scenes. Each *MeshInstance* node is a single, unique *instance* of a mesh inside the scene. They can display complex meshes imported from third-party software like *Blender (`www.blender.org`)* or auto-generated Primitive meshes like Cubes and Spheres. To generate a Cube, select the *MeshInstance* by left-clicking it in the scene viewport, or from the *Scene Tree Panel*, and then (from the Inspector) click the **Mesh** drop-down field and choose **New CubeMesh** to create a new Cube Mesh Asset in the Project. See Figure 2-14.

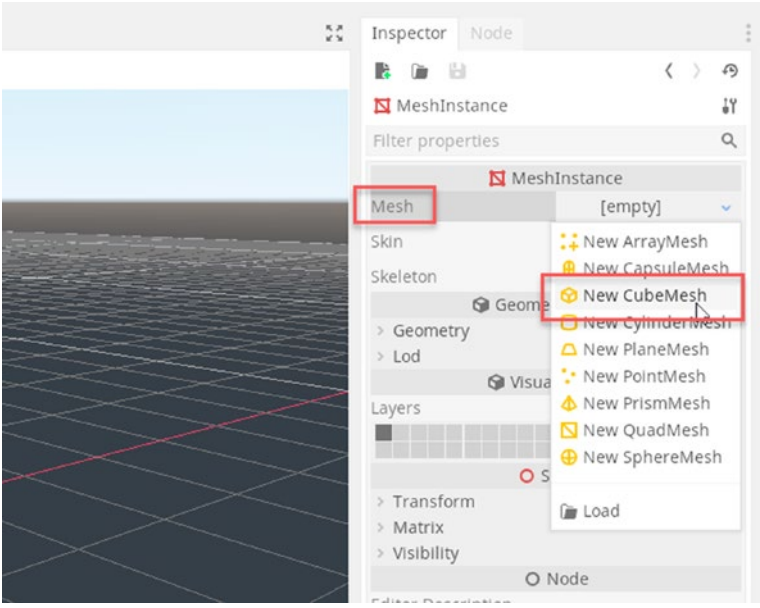

Figure 2-14. *Generating a New Cube Mesh Asset in the Godot Project...*

After selecting *New CubeMesh*, a cube will be shown in the scene. By default, all meshes are visible in Godot. Clicking this option generated a new procedural mesh that Godot has saved internally as a *Resource* (Asset) of the Project, even though nothing extra is shown in the *File System Panel*. The newly generated mesh has been assigned as the **Mesh** property for the *MeshInstance* object, which positions the mesh by the **Transform Component**. See Figure 2-15.

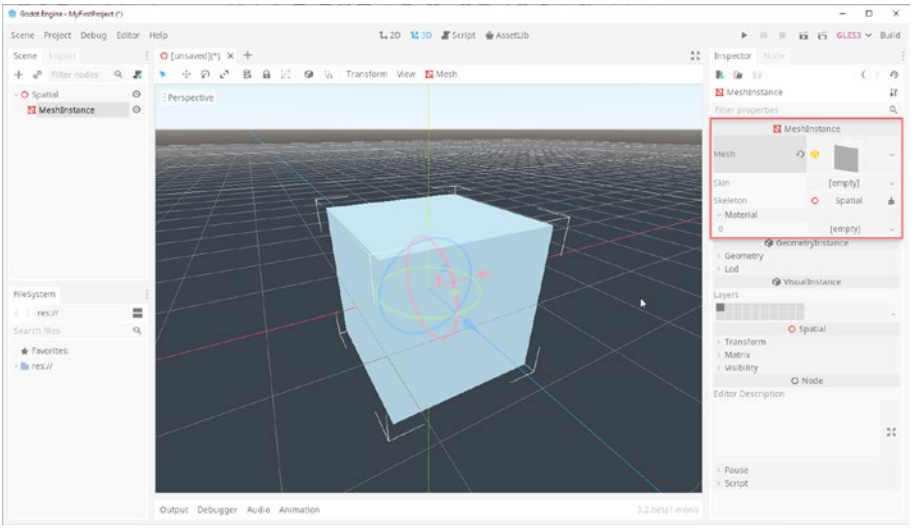

Figure 2-15. *Generating a Cube Mesh and Displaying It in the Scene*

If you want to change the **Location**, **Rotation**, or **Scale** of a Node in a 3D scene, then you can use its **Transform Component**. To view this, select the Node by left-clicking the viewport, or select from the *Scene Tree*, and then expand the **Transform** group inside the Inspector. You can type numerical values directly into the *X*, *Y*, and *Z* fields to set them immediately; or you can left-click and drag your mouse over the *X*, *Y*, and *Z* labels beside the fields to smoothly change the values. Dragging leftward decreases the field and rightward increases. See Figure 2-16.

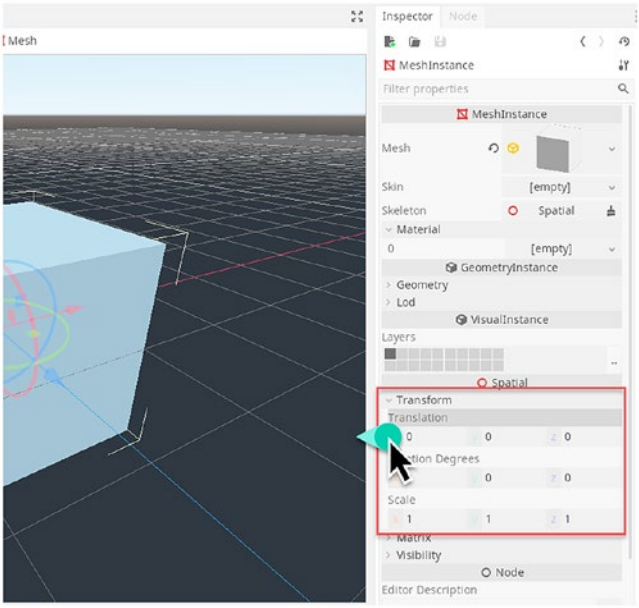

Figure 2-16. *Adjusting Transformation Properties for a Node*

Congratulations! You just made your first Godot 3D object. It's easy and rational but significantly distinct from the Unity method. There is however an important limitation. Specifically, Godot **will not** automatically generate any colliders for a primitive object made in this way. Consequently, physical objects – like first-person controllers and rigid bodies – will smoothly pass through the cube mesh as though it were an apparition or hologram, an object without substance. This is not a problem here specifically; but it would be a problem if you wanted objects to bump into the cube or to be obstructed by it. We'll see shortly how to add collision properties to the cube.

Navigation and Transformation

We saw previously how to create Godot projects. Plus, we saw how to add primitive objects – like cubes and sphere – inside 3D scenes. Although we saw important differences in the workflow between Godot and Unity, the

underlying principles are really the same. You should first create a Scene and then simply put objects in it. This section looks at the main Godot control system, both for moving objects around a scene and for navigating the scene camera. You'll be pleased to know that there's hardly any difference between Unity and Godot here – if you set things up right.

Let's start by configuring Godot to use the Maya Control scheme. This makes Godot's controls behave like Unity's controls. To do this, choose *Editor* ➤ *Editor Settings* from the application menu. This displays the Editor Settings Window. From here, select Editors ➤ 3D from the options list, and then set the *Navigation Scheme drop-down* to *Maya*. See Figure 2-17.

Note If you're using Godot on a laptop or mobile computer with a trackpad for a mouse, then you'll want to enable *Emulate 3 Button Mouse* to get access to all viewport navigation features.

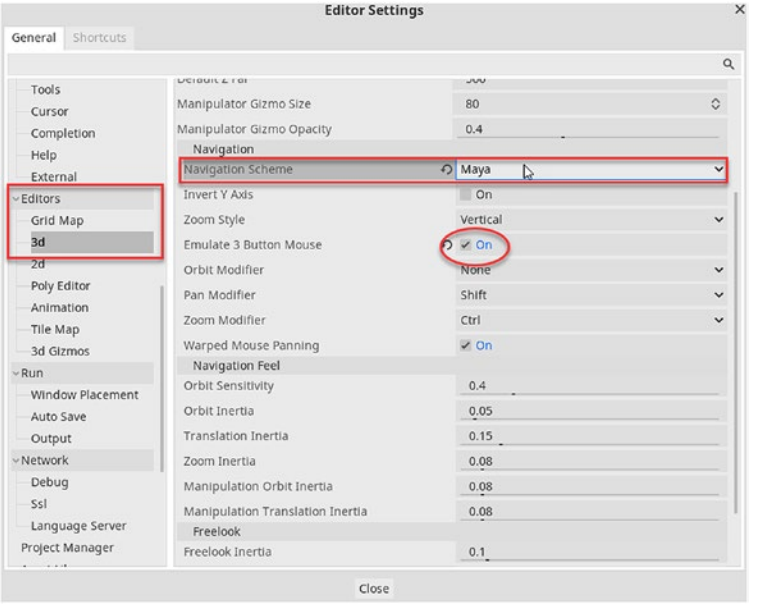

Figure 2-17. *Configuring Godot Controls...*

Zoom

You can zoom in and out of the viewport by rolling the middle mouse wheel forward or back, respectively, or by sliding two fingers along a trackpad. See Figure 2-18.

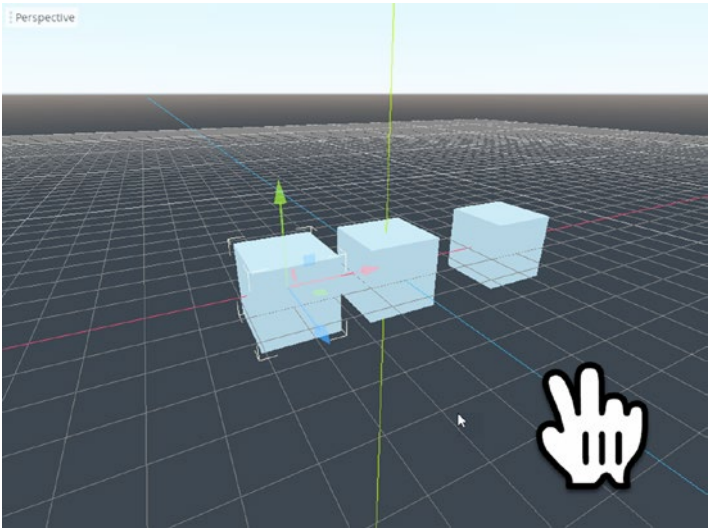

Figure 2-18. *Viewport Zooming*

Note Technically, with Zoom, we're actually *dollying* the camera and not *zooming*. That is, we're *moving* the camera forward in 3D space, as opposed to *keeping the camera static* but changing the focal length of the lens.

Pan

Pan allows you to slide the viewport camera left or right or up or down. Just drag the mouse around while holding down the **Alt** key and middle mouse button. If you're using a trackpad, then hold down the shift key and drag around the pad. Moving left, right, up, or down will pan the view in that direction. See Figure 2-19.

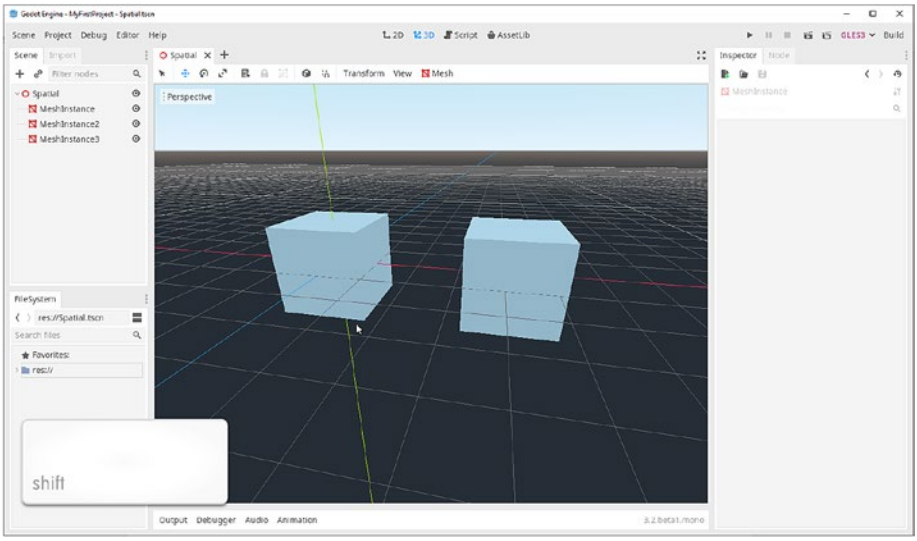

Figure 2-19. *Panning Around the Viewport*

Frame and Orbit

Frequently, you'll want to rotate your viewport camera around a centered node – like a mesh – to get view of it from any angle. You can do this using the Frame and Orbit technique, which is really simple. Start by "Framing" the object you want to rotate around. To do this, select the object that will be your target – either by left-clicking it on the viewport or by clicking it from the Scene Tree Panel. To frame the selected Node, you can either press the **F** key on the keyboard or double-click the Node icon from the Scene

Tree Panel – either way works. When you do this, the viewport will center its aim on the selected node. Then you can freely rotate (orbit) around it by holding the **Alt** key (**Option** key on a Mac) and then *left-clicking and dragging* the mouse in the direction of rotation. See Figure 2-20.

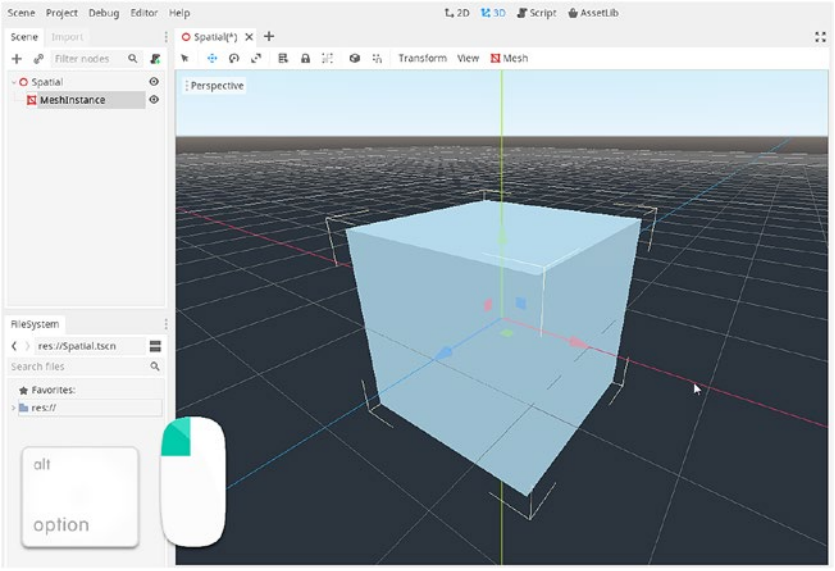

Figure 2-20. *Orbiting the Framed Object*

First-Person Controls

Now then, who can truly resist exploring their 3D worlds in first-person mode, as though you were actually there? Certainly not me! So thankfully, Godot offers first-person controls, allowing you to view the world directly from the perspective of a *character in the world*. To access first-person controls, hold down the right mouse button, and then use the **W**, **A**, **S**, and **D** keys on the keyboard to move the camera forward, left, down, and right, respectively. See Figure 2-21.

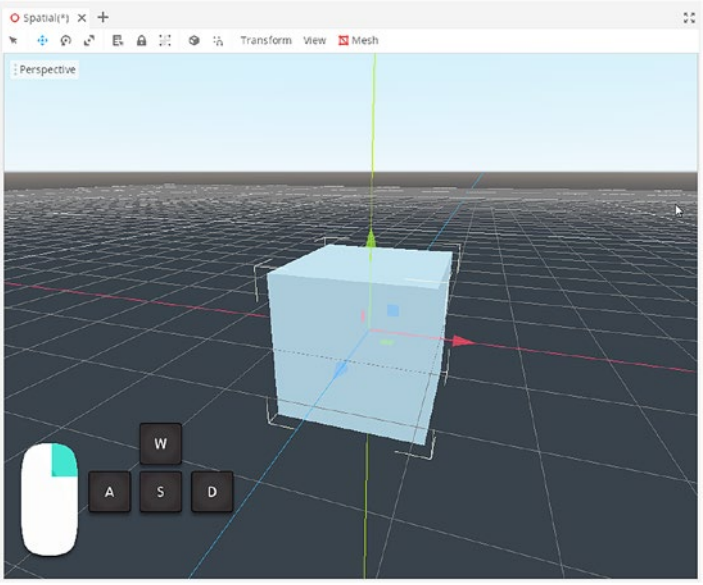

Figure 2-21. *First-Person Controls…*

Selecting, Moving, Rotating, and Scaling Nodes

You've seen how to move the viewport camera to get a better view of a scene. But you can also move, rotate, and scale nodes in the scene – just like Unity. The controls are the same too. To select an object without transforming it, just hit the **Q** key and then left-click the node in the viewport, or select it from the *Scene Tree Panel*. You can also activate the **Select Tool** from the Godot toolbar, after which you can left-click objects in the viewport to select them. See Figure 2-22.

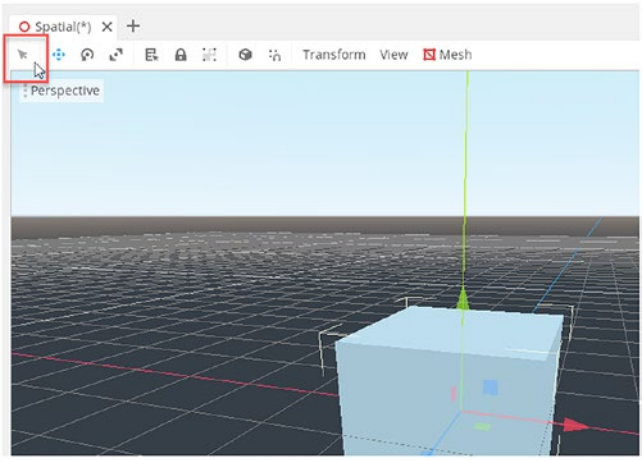

Figure 2-22. *Select Tool*

You can use the **W** key to activate the Move tool, allowing you to move selected Nodes around using the viewport Gizmo along the X, Y, or Z axis. Similarly, you can rotate objects using the **E** key and scale them with the **R** key. After activating the *Move*, *Rotate*, or *Scale* tools, you should simply click and drag the selected Node in the viewport to apply the tool, just like Unity. See Figure 2-23.

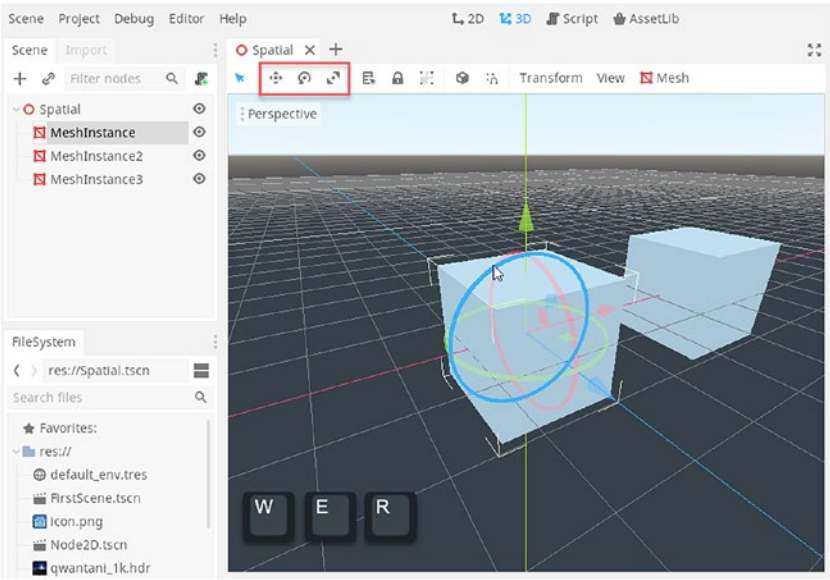

Figure 2-23. *Translate, Rotate, and Scale Tools for Transforming Nodes*

Local and World Space Transforms

Moving objects around in 3D is easy. But, as soon as you start rotating objects and then moving them, you might run into confusing coordinate troubles. For example, if a character that's standing still while facing an oncoming target ahead is rotated to a new orientation, where should the character go when you move them forward? They could move *forward* in world space – along the Z axis – which might be different from the direction in which the character is now facing after being rotated. To solve this problem, Godot understands the scene using two different coordinate systems. The first is **World Space**. This is an absolute and fixed coordinate system divided into **X**, **Y**, and **Z**. Every spatial node in the scene has a position in world space, measured from the world origin at (0,0,0). The second is **Local Space**. Each spatial node has its own Local Space,

and each node is the origin of its own space. When a Node is rotated, the coordinate system rotates with it, so that the Z axis in local space always points *forward*, in the direction that the object is facing. Godot makes it easy to switch between World and Local spaces. You can do this by pressing the **T** key or by toggling the Local/World space icon, to switch back and forth between Local and World space, making transformations – like move, rotate, and scale – easier to apply. See Figure 2-24.

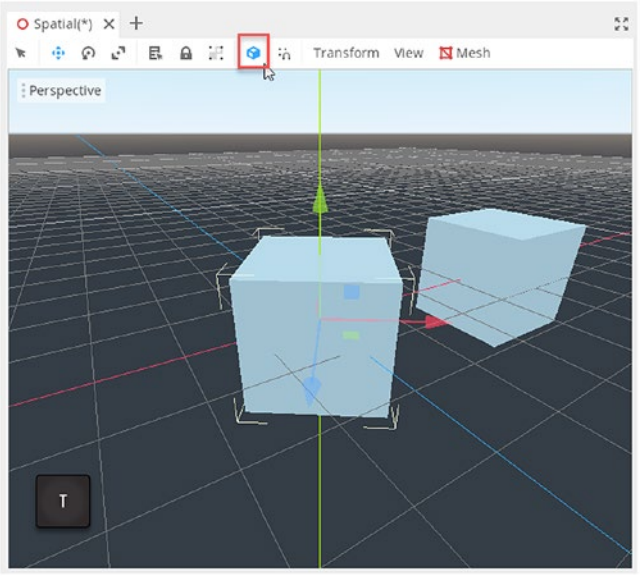

Figure 2-24. *Local vs. World Coordinate Space*

Scene vs. Game Mode

The Unity interface features a *Scene* and *Game* tab. These represent two distinct modes for the Unity Editor. In the *Scene* tab, you build a game world; add objects and configure the code. In the *Game* tab, by contrast, you play-test your game from the perspective of a gamer or a game tester. See Figure 2-25.

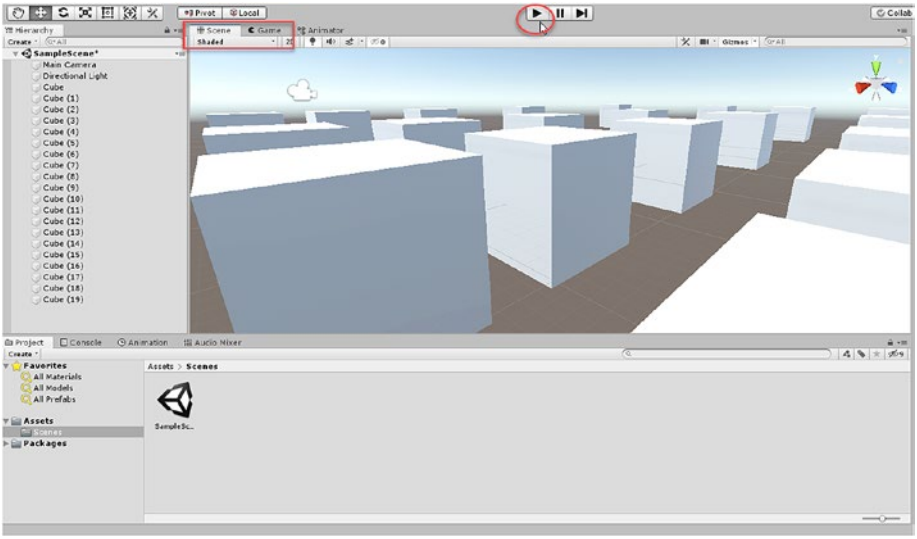

Figure 2-25. *Scene and Game Mode in Unity*

The great thing about *Unity's* **Game Mode** is that, even while your game is running in the Game tab, you can still switch over to Scene mode and inspect your objects and the scene live. This makes debugging much simpler and even fun! Godot, however, doesn't feature a Game tab exactly like Unity's; but, you can still do most of the same things. Let's see how. To play-test your game, simply click the **Play** icon on the application toolbar or press **F5** on the keyboard. See Figure 2-26.

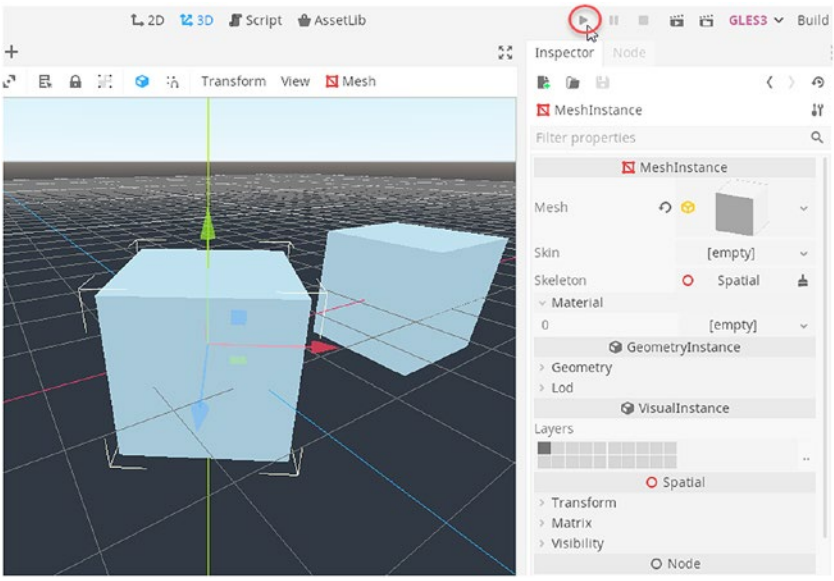

Figure 2-26. *Launching Godot's Play Mode*

If this is the first time you're pressing **Play**, Godot will probably display a confirmation notice, asking you to specify the **Main Scene**. The Main Scene is the default scene, the scene that'll always load first when your game is executed. From the Confirmation Window, you can make the active scene the Main Scene by simply choosing **Select**. The active scene then becomes the Main Scene and will execute. See Figure 2-27.

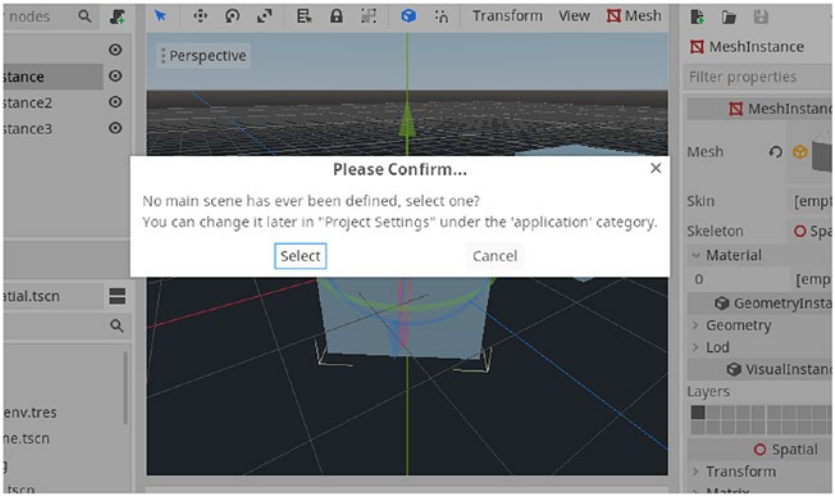

Figure 2-27. *Setting the Main Scene*

You can always change the Main Scene by choosing *Project* ➤ *Project Settings* from the application menu to display the active Project settings. From here, select *Application* ➤ *Run* from the *Settings* list, and then either remove or choose a different scene for the **Main Scene** field. See Figure 2-28.

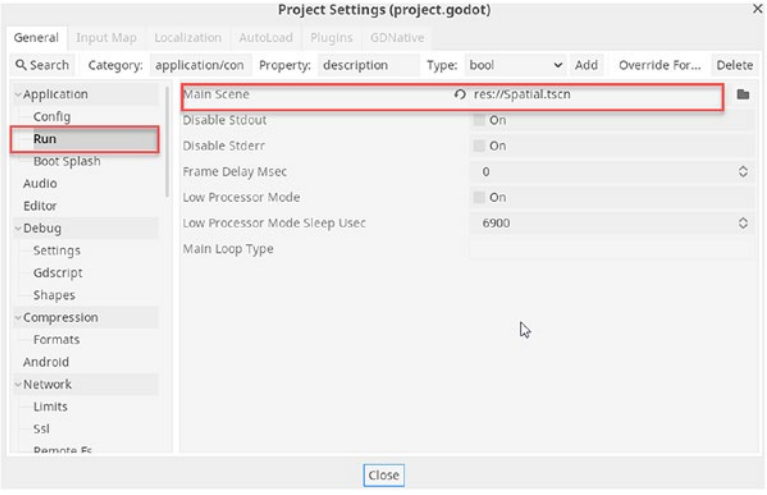

Figure 2-28. *Changing the Main Scene*

If you just want to play the active scene immediately, as if it were the Main Scene, then just click **Play Scene** or press **F6** on the keyboard. See Figure 2-29.

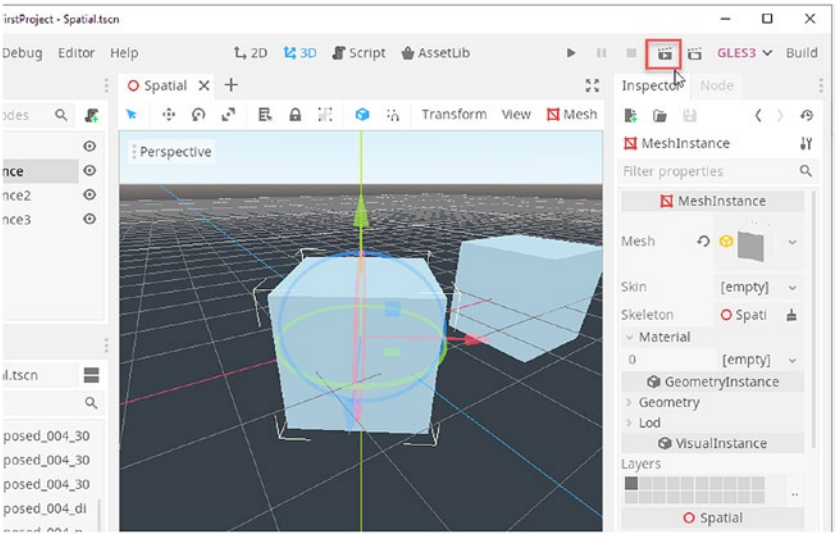

Figure 2-29. *Starting to Play the Active Scene*

After entering **Play** mode, your scene will always run in a separate, free-floating Window, which is a completely separate process. See Figure 2-30.

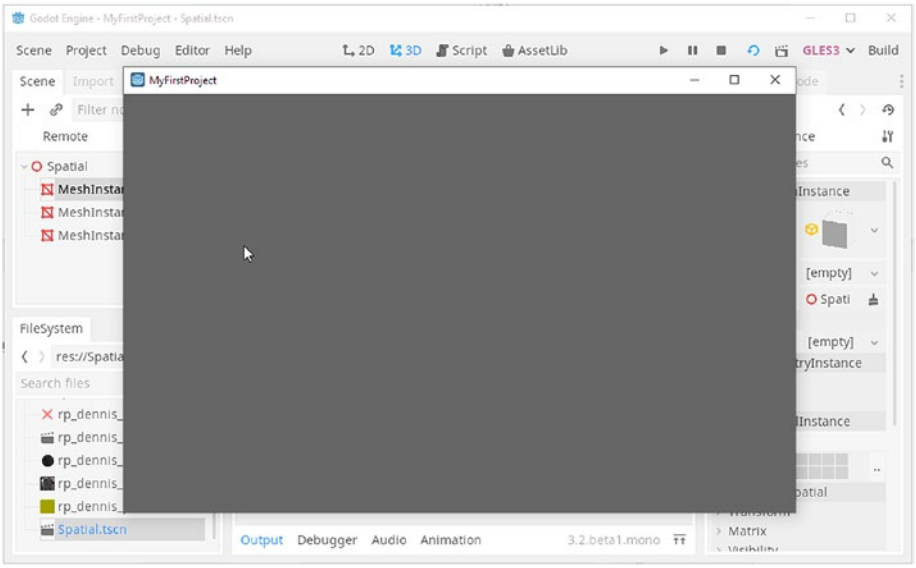

Figure 2-30. *Playing the Active Scene*

Remember, your scene will only be visible in the Application Window during Play Mode if it has a camera node in the tree. Scenes are not created with cameras by default, so you'll probably need to add one manually. You can do that in the same way as adding any other Node. Right-click the root node in the *Scene Tree* panel, and then choose *Add Node* from the context menu. From the Node Creation Window, select **Camera** to add a scene camera. See Figure 2-31.

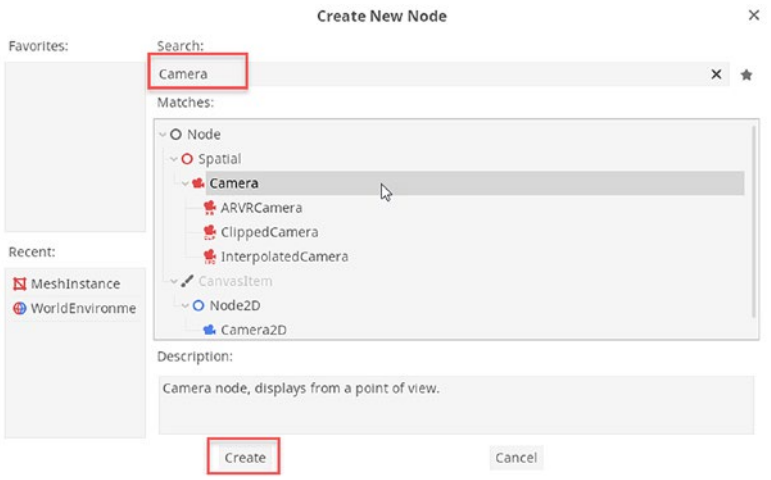

Figure 2-31. *Creating a Camera Object*

When your scene has a camera Node, then **Play** mode will display the scene from the perspective of the camera. See Figure 2-32.

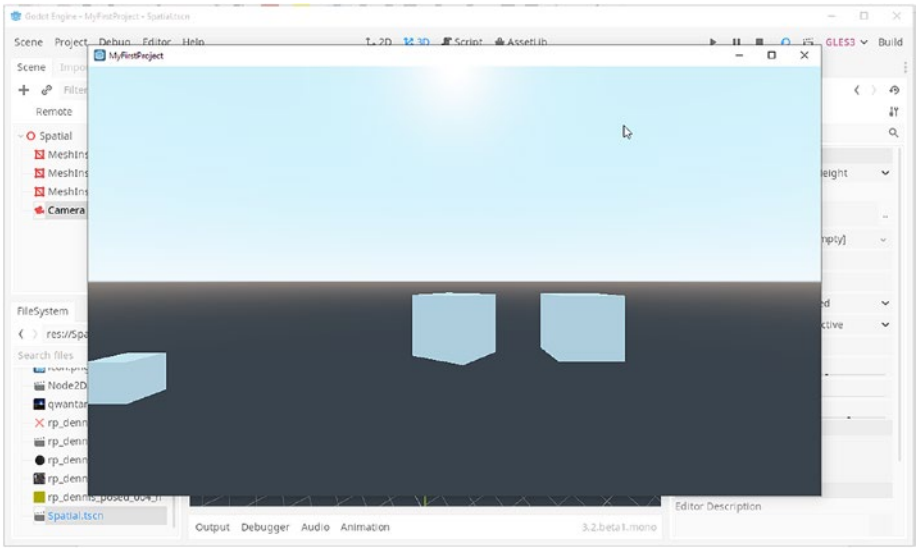

Figure 2-32. *Playing a Scene with a Camera Node*

Since Game Mode runs in a separate Window, as a separate process, it's completely detached from the main Godot Editor and the scene. Consequently, any changes you make in the Godot Editor during Game Mode – such as moving a mesh or moving the player character – will not be synchronized to the Game Window by default. However, you *can* synchronize changes between the two modes. To do this, select *Debug* ➤ *Sync Scene Changes* from the Application Menu. Now, when you hit *Play* with this option activated, any changes you make in the scene during Game mode will be updated in the Game Window automatically and immediately. See Figure 2-33.

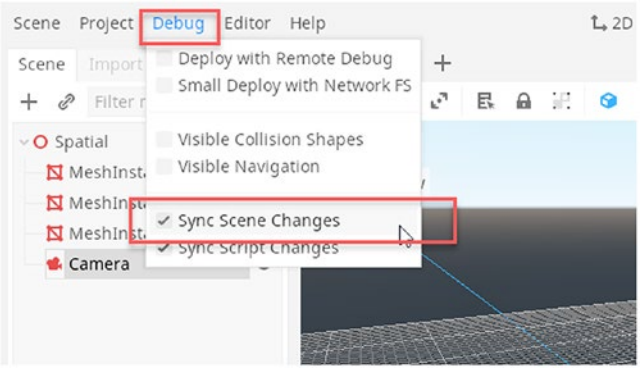

Figure 2-33. *Activating Sync Scene Changes*

Note Unlike Unity, where any changes made to a scene during Game mode will be reversed as Game mode ends, all Godot changes will persist and remain.

Sync Scene Changes works great whenever you need edits made in the Scene view to transfer directly to the Game Window. But sometimes you need a more extensive connection than the one-way one from Editor to Game. To create a two-way connection between editor to game and

game to editor, you should enable the Remote View from the Scene Tree Panel during Game Mode. This updates the Editor to reflect the Game View and updates the Game View to reflect the editor. This allows you to select objects in the Editor and to inspect their properties while the game is running, just like Visual Debugging in Unity. To do this, click the *Play* button to enter Game mode, and then click the **Remote** button from the Scene Panel. See Figure 2-34.

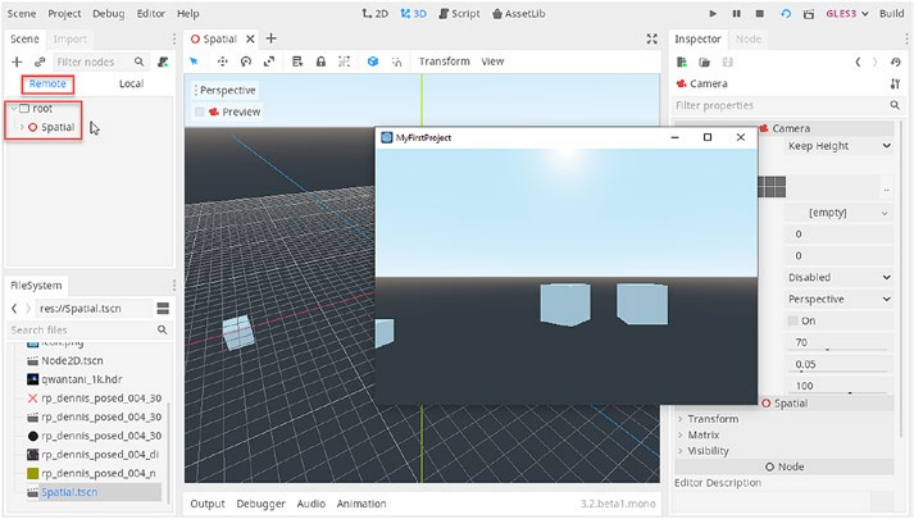

Figure 2-34. *Activating the Remote View from the Scene Tree*

Excellent! You can now use Godot's Play Mode and Visual Debugging features. At first sight, the Play mode features seem limited; but it turns out they're extensive and offer a lot of parity with Unity's Game Mode. Next, and finally, we'll turn to working with Resources in Godot.

Resources

Basically, a **Resource** in Godot compares to an **Asset** in Unity. It includes all the files on which your project depends: meshes, textures, audio, animations, scenes, and more. Some resources are *fundamental* insofar as their structure means they must necessarily contain no external dependencies, for example, image files, audio files, and text files. Other resources are of a higher order. These may contain dependencies, such as a mesh file that *depends on* a set of external images for its materials or a scene file that depends on a set of meshes because it contains instances of them. Godot doesn't actually make any formal distinction between fundamental and higher-order assets as discussed here. But, the distinction nevertheless exists in practice because higher-order assets simply won't function as intended whenever their dependencies are missing from the project. To import an asset into Godot, you just need to drag and drop the files or folders into the *File System Panel*. When you do this, Godot imports any valid and supported asset files. See Figure 2-35.

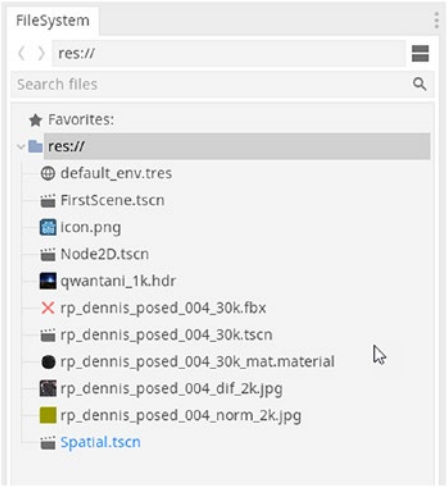

Figure 2-35. *Importing Different Asset Types into a Project*

Godot supports many different file formats. For image files, both JPG and PNG are supported, including transparency; and for Audio, WAV and OGG are supported. From Godot 3.2 onward, the AssImp library (http://cms.assimp.org/) has been integrated into the core code, meaning that Godot now supports an extensive range of 3D formats for meshes. This includes FBX, DAE, glTF, and more.

Note For a complete list of supported mesh types, please see the AssImp documentation here: http://assimp.sourceforge.net/main_features_formats.html.

After importing an asset, Godot copies the imported file to a hidden folder within the project. This ensures that any changes you may make during development will not damage the original if it's ever needed again. You can change the properties for an imported asset by selecting the asset in the *File System Panel* and then switching to the **Import** tab in the *Scene Tree Panel*. This displays a special Inspector that applies to **Assets**. See Figure 2-36.

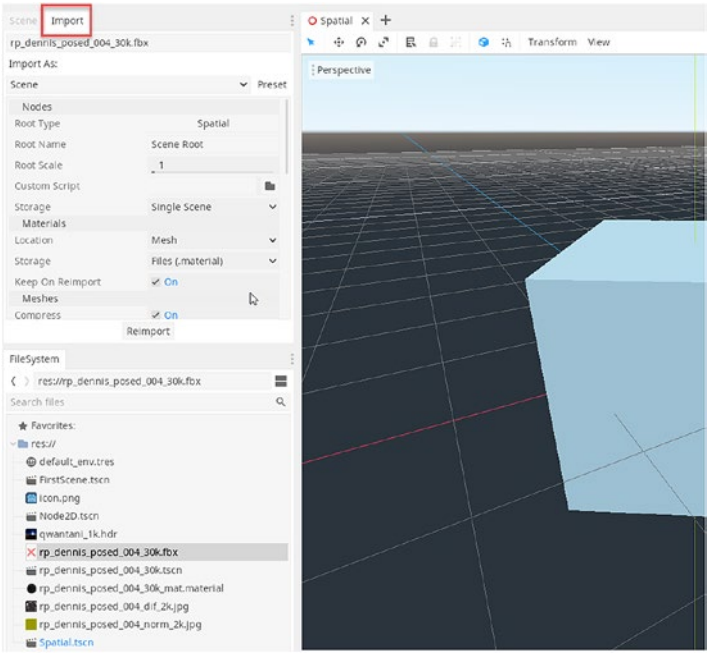

Figure 2-36. *Configuring File Import Settings*

Summary

This chapter explored the fundamentals of Godot that every Unity user simply needs to know. There's more to learn, of course. But this chapter introduced the basic building blocks for creating a project and getting started quickly for 3D scenes. 2D scenes – though distinct in their coordinate space – follow the same workflow we've seen here. Most 3D object types in Godot have 2D counterparts. From this point forward, each chapter will build on the knowledge contained here. The next turns to C# and to creating script files in Godot.

CHAPTER 3

Scripting with C# in Godot: Common Tasks

Godot supports many scripting languages unofficially through add-ons and patches. You'll find people who've used *GDScript*, *Python*, *Visual Scripting*, and *JavaScript*. This book and chapter, however, focus on C#. C# is natively supported by Godot and is the natural choice for Unity developers where C# is used almost exclusively. Here, we'll write our first Hello World script file with C# and see how to code common gameplay tasks with C# through the Godot API (Application Programming Interface). If you've coded with C# in Unity, you'll be used to a whole string of commands and classes, like *Transform*, *GetComponent*, *Time.deltaTime*, *Update*, and lots more. Now, while Godot often uses very different names for its classes and functions, you'll be pleased to know that Godot has Unity equivalents and is remarkably similar sometimes. Table 3-1 shows some common mappings between Unity and Godot classes and function calls. Figure 3-1 demonstrates how to download the Godot engine with C# support natively included.

Note To use C# in Godot, you'll need to download the Godot Mono Build from the Godot home page: `https://godotengine.org/download/`.

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5_3

Figure 3-1. *Downloading Godot with Mono Support*

Table 3-1. *Terminology: Unity C# vs. Godot C#*

Unity	Godot
Print	GD.Print
Update()	_process()
Start()	_ready()
Prefab	Scene
Hierarchy	Scene Tree
Project Panel	File System
Inspector	Inspector
Empty Object	Spatial Node
Asset	Resource
Tag	Group

.NET and Build Problems with C#

With Unity, you can normally just install the engine and start using C# right away. This isn't always the case with Godot, however. Many people report compilation issues or other problems preventing them from effectively writing and compiling C# code out of the box. Let's start by seeing what issues arise, in case they happen for you, and then how to address them. When creating a Godot project for the first time after a fresh install, it's a good idea to check that your C# code will compile successfully for you. To do this, simply create a new Script file. Right-click your mouse over the *res://* item in the *FileSystem* panel, and choose *New Script* from the context menu. See Figure 3-2.

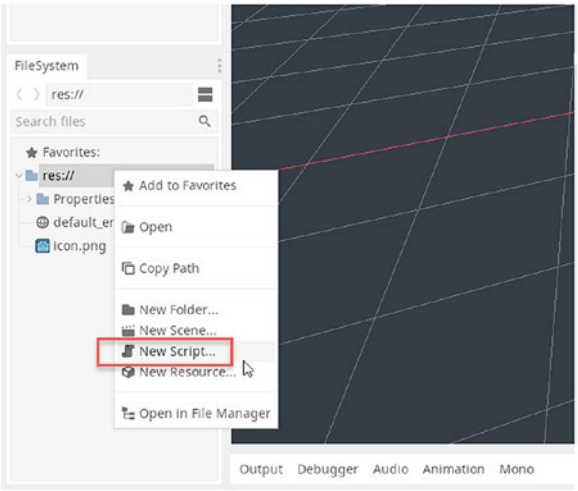

Figure 3-2. *Creating a New Script Resource*

Next, select C# for the scripting language, and inherit from the *Spatial* class, since we'll be adding this script to a *Spatial* object. I've named the script *HelloWorld* to print a simple message to the console for a Hello World program, coded later in this chapter. See Figure 3-3. When you've chosen these settings, simply click *Create*. The script file will be generated and added to the current Godot project.

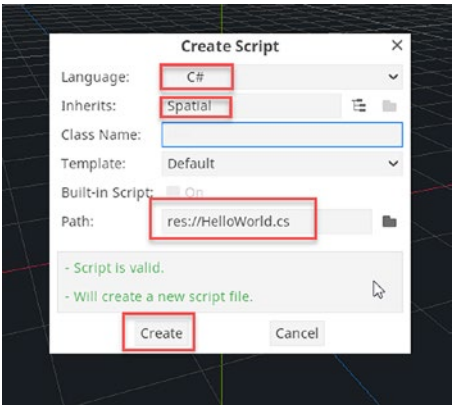

Figure 3-3. *Creating a New Script File… If Class Name Is Left Blank, Godot Will Name the Class After the Filename*

After the C# file is generated, it'll appear inside the *FileSystem* panel, as shown in Figure 3-4. And just as a Unity script file is populated with default code, derived from *MonoBehaviour*, Godot also generates a code file, as shown in Listing 3-1.

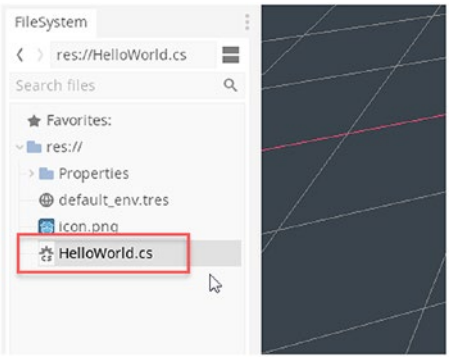

Figure 3-4. *Script Files Are Added to a Godot Project and Appear in the File System Panel*

Listing 3-1. Default Generated Godot Script File

```
using Godot;
using System;

public class HelloWorld : Spatial
{
    // Declare member variables here. Examples:
    // private int a = 2;
    // private string b = "text";

    // Called when the node enters the scene tree for the first time.
    public override void _Ready()
    {

    }

//  // Called every frame. 'delta' is the elapsed time since the
previous frame.
//  public override void _Process(float delta)
//  {
//
//  }
}
```

Now just click *Build* at the top-right corner of the Godot Editor interface. See Figure 3-5. Clicking this button begins the C# compilation process. The result will indicate whether you'll successfully compile C# code at all using the current install. You may see a compilation progress bar briefly, as shown in Figure 3-5.

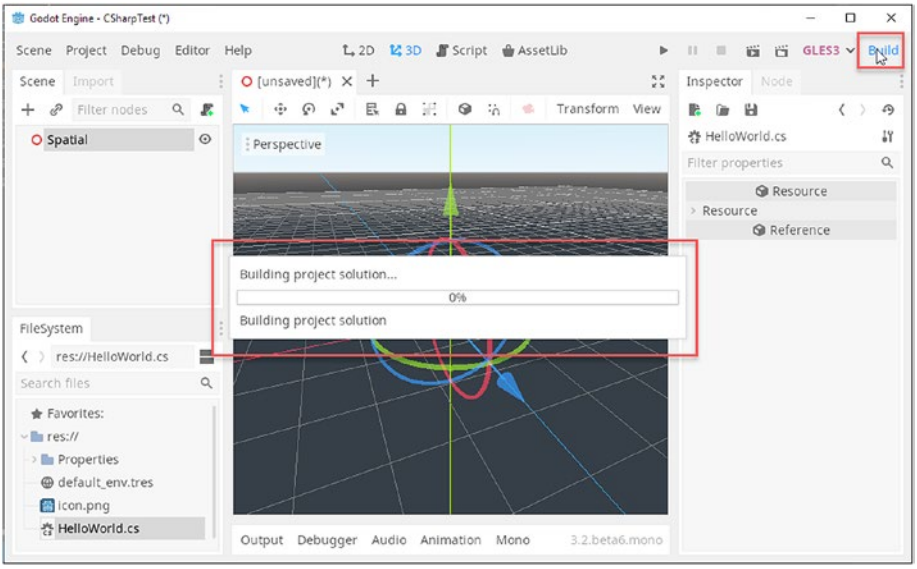

Figure 3-5. *Compiling a C# Script File*

After compiling, if the Godot engine returns silently and the output window (at the bottom of the interface) doesn't display any errors, then everything probably compiled successfully. To confirm, press the *Play* button from the toolbar to run the application. If it runs successfully, the project compiled successfully and you're good to continue to the next section. However, if the *Output* window displays and prints an Error message, then there's a problem with your C# setup. See Figure 3-6.

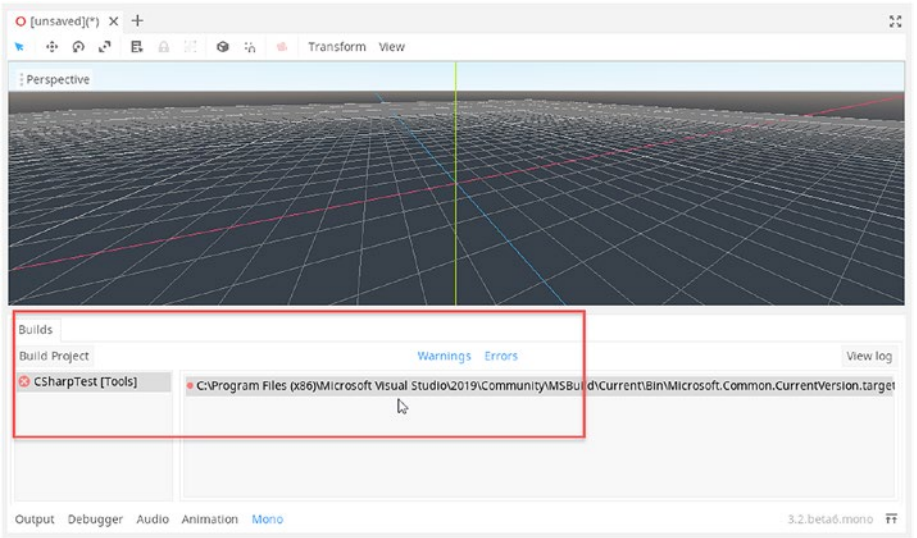

Figure 3-6. *Examining a Compilation Error*

The printed Error Message will usually be related to a mismatch in the .Net Framework version. Godot is unable to compile your C# for the targeted .Net framework. To solve this, first ensure your Godot project is targeting the MS .Net framework, as opposed to the Mono Framework. You can set this by choosing *Editor* ➤ *Editor Settings* from the application menu to display the *Editor Settings Window*. Then, choose *Mono* ➤ *Builds* from the Editor menu. Select *VS Build Tools* for the *Build Tools* drop-down. See Figure 3-7.

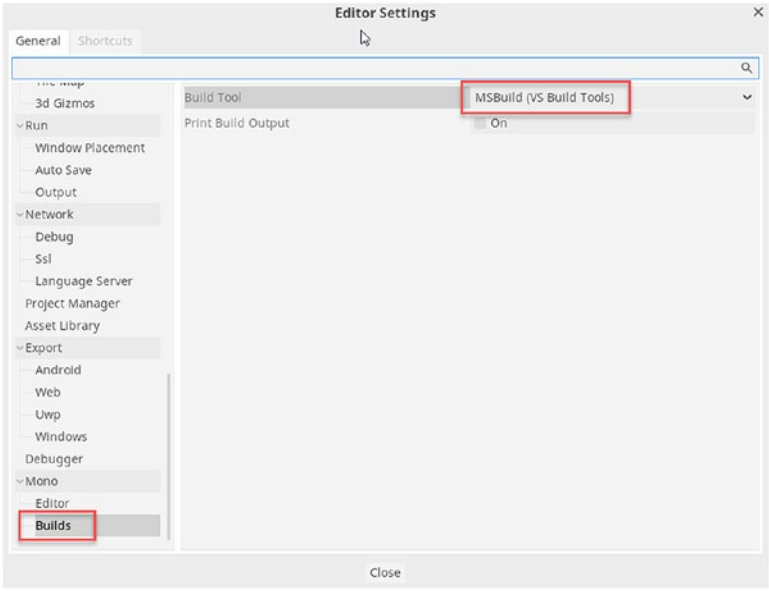

Figure 3-7. *Setting the VS Build Tools*

Next, you'll need to **either** (1) change the targeted .NET framework for your Godot project to the version matching the latest system one or (2) install the .NET Framework 4.5 targeting pack for Visual Studio. This chapter explores the second option. To get started, you'll need to install Visual Studio Community. You can download it from here: `https://visualstudio.microsoft.com/downloads/`.

Note To view steps for the first option, check out my free BeIndie.Biz tutorial on YouTube here: `www.youtube.com/watch?v=KyqBKq_wQQw`.

While running the Visual Studio Installer, select the *Individual Components* tab, and from the *.NET* group, enable the *.NET Framework 4.5 retargeting pack*. See Figure 3-8. Then continue or complete the installation. After installation, restart your computer, and your Godot project should now compile successfully. Great!

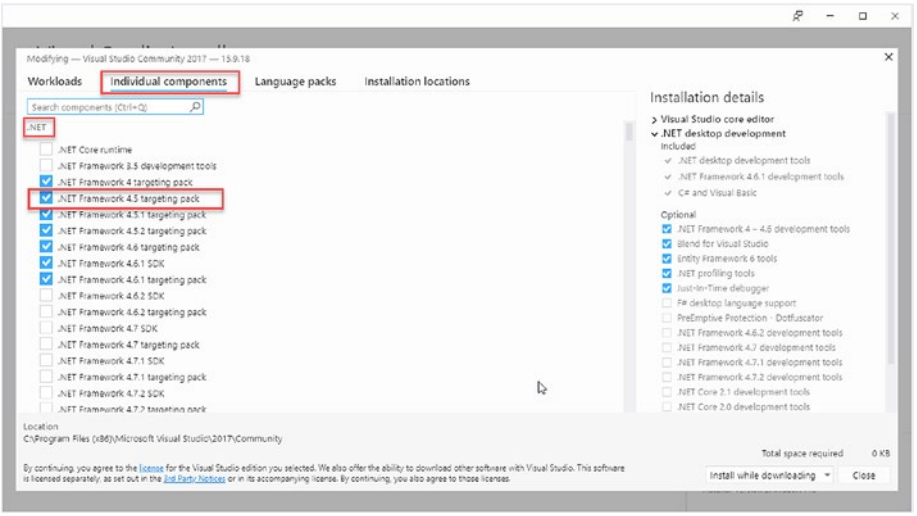

Figure 3-8. *Installing the .NET Retargeting Pack*

Building a Hello World Program

Hello World programs simply print the message "Hello World" to an output, like a screen, or a printer, or (in our case) the *Output Window*. Printing the message "Hello World" isn't especially useful itself – certainly not. But the process of being able to send a human-readable string message to the output is incredibly helpful. It's useful notably as a Debugging tool. You can print messages to show the order of code execution, the value of variables, and other data. In this section, we'll write a Hello World program for Godot. To start, create a Hello World C# script file, as demonstrated in the previous section. Then double-click the script

file from the *FileSystem* Panel to open it inside your code Editor, such as *Visual Studio Code*. See Chapter 1 for details. See Figure 3-9 for a default code file open in an Editor, ready to code.

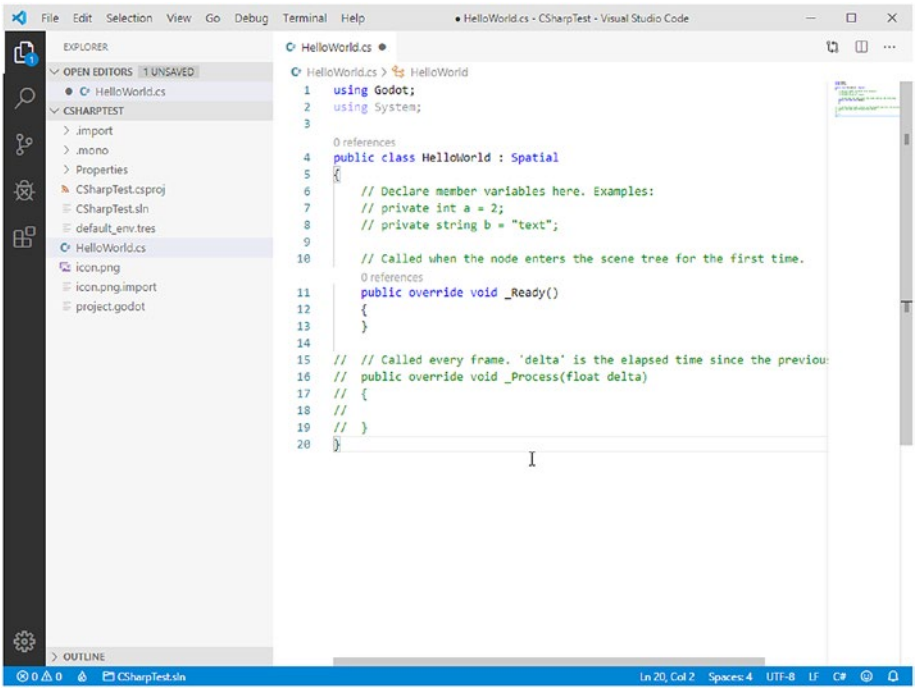

Figure 3-9. *Preparing to Code a Hello World Program*

Note You can change the Visual Studio Code Color Scheme by choosing *File* ➤ *Preferences* ➤ *Settings* from the application menu. Then select Workbench ➤ Appearance. And set the Color Theme to your preference. *Visual Studio Light* is used in Figure 3-9.

In Godot, the C# function *GD.Print* prints a string to the output console for reading in the editor interface while the game is running. Our Hello World application should print the message “Hello World” to the console

at application startup. For this reason, we'll need to use the *_ready* event. *_ready* is comparable to Unity's *Start* event. *_ready* is called after a Node is created and when it is added to the scene tree, that is, when the object enters the scene hierarchy. Look at Listing 3-2 to see an example of a Hello World program in C#.

Listing 3-2. A Hello World Program

```
using Godot;
using System;

public class HelloWorld : Spatial
{
    // Declare member variables here. Examples:
    // private int a = 2;
    // private string b = "text";

    // Called when the node enters the scene tree for the first time.
    public override void _Ready()
    {
        GD.Print("Hello World");
    }

//  // Called every frame. 'delta' is the elapsed time since the
previous frame.
//  public override void _Process(float delta)
//  {
//
//  }
}
```

Note Unity also has the *Awake* event, which is called before *Start*. Godot has a similar event called *_init*. The *_init* function is invoked when a node is created in memory *but before* it is added to the scene tree, becoming a "thing in the world." The *_init* event always happens before *_ready* and is useful for initializing private member variables, as well as for running private member functions. The *_ready* event should be preferred, however, for any initialization code that references external nodes or objects or which accesses the scene tree. The sequence in which *_ready* events are invoked across all nodes in a scene should never be assumed. For this reason, every *_ready* event should assume that all other nodes are already initialized as a result of their *_init* event called earlier.

After writing your *Hello World* code, as shown in Listing 3-2, you should attach to it a Spatial node in a 3D scene. To do this, select a spatial node in the scene, and from the Object Inspector, expand the *Script* group. From the *Script* drop-down field, select the *Load* option. See Figure 3-10.

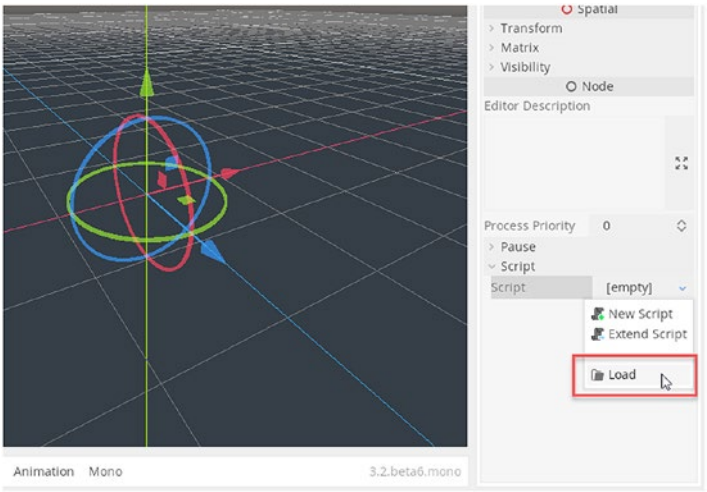

Figure 3-10. *Adding a Script to a Node*

From the *Load* menu, you may choose any .cs script file to attach to the selected node where it'll be instantiated and run in Play Mode, just as it does in Unity. In our case, simply select the newly created *HelloWorld.cs* file from the file list and then click the *Open* button at the bottom-left corner. See Figure 3-11.

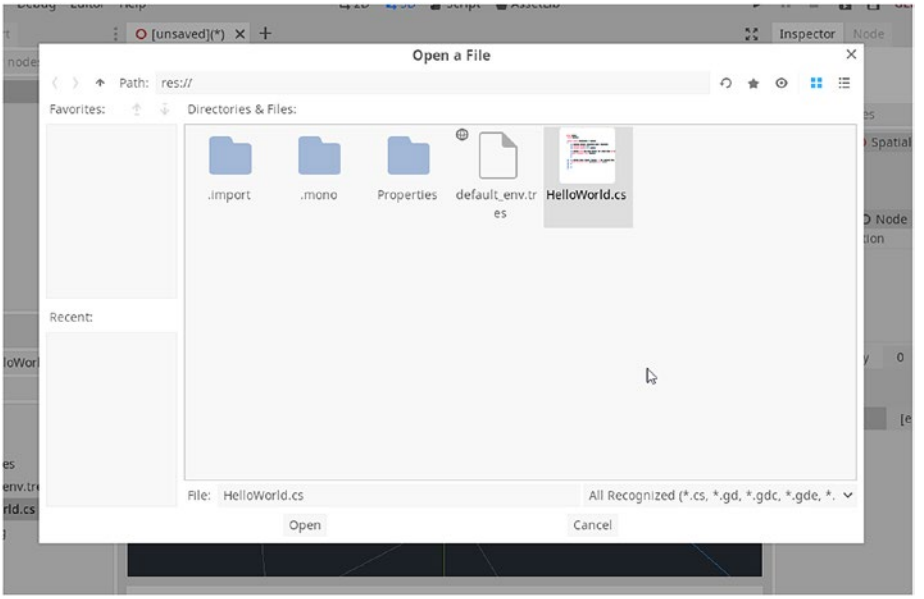

Figure 3-11. *Attaching a Hello World C# Script to the Selected Node*

After clicking *Open*, simply confirm the Script has been added to the Node by checking the Script field from the Inspector, which should now display the associated filename. Further, always click the *Build* button, from the top-right corner of the interface, before playing your project. This ensures you're always testing with the latest compiled code. See Figure 3-12.

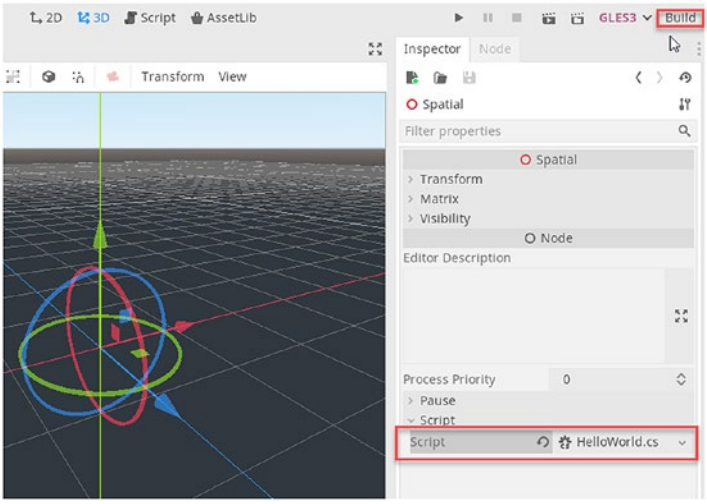

Figure 3-12. *Always Build the Latest Code Before Playing Your Project*

After compiling, click the *Play Scene* button (F6) from the toolbar to run your game in an application Window. See Figure 3-13.

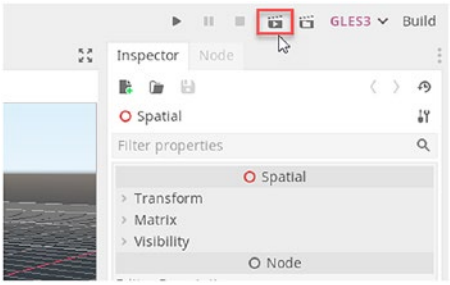

Figure 3-13. *Playing the Open Scene*

On running your game, the *_ready* function executes immediately for every Node in the scene. By viewing the *Output Window* at the bottom center of the Editor interface, you'll now see the complete message "Hello

World" printed straight from the *_Ready* function. Printing messages like this is great – it's an effective strategy for debugging your code, seeing its flow as the application executes. See Figure 3-14.

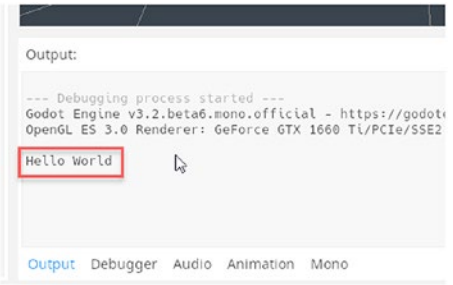

Figure 3-14. *Printing Hello World to the Output Window*

Working with Nodes

A Godot scene is, fundamentally, a hierarchical tree of Nodes (the **SceneTree**). The SceneTree has one Root node – the topmost node – from which all others are children. They are children either directly by being a child of the root node – or indirectly by being a grandchild of the root node – or by having an even more distant connection. Each Node in the scene may contain a maximum of *only one attached* Script file. In Unity, by contrast, a *GameObject* may have multiple Scripts attached, each becoming a *Component*. Godot however expects you to attach only one script to a Node wherever you need custom behavior. More complex behaviors – requiring multiple systems – are created not by adding more scripts to a node, but by building a hierarchy of Nodes with different Scripts attached to each, or, in some cases, by using C# inheritance to customize a single Script on a Node. This may seem like a limitation compared to Unity; but really, it only requires a different way of thinking and a different approach, as we'll see. Each Node, therefore, is an instantiation of a *single class* only, one that derives from the *Node* class directly or indirectly by one of Node's descendant classes. In the previous

section, for example, a Hello World class was derived from the *Spatial* class, which in turn derives from the *Node* class. The *Godot API Reference Documentation* provides a solid description of each native class – such as Node, Spatial, Camera, Light, and more – including its member variables and functions. Each class gets its own dedicated documentation page. You can see any class' inheritance connection to its ultimate ancestor Node class by viewing the ***Inherits*** section of the class' documentation page. See Figure 3-15. Knowing this connection is important for understanding the C# variables and methods supported by any given node in the SceneTree.

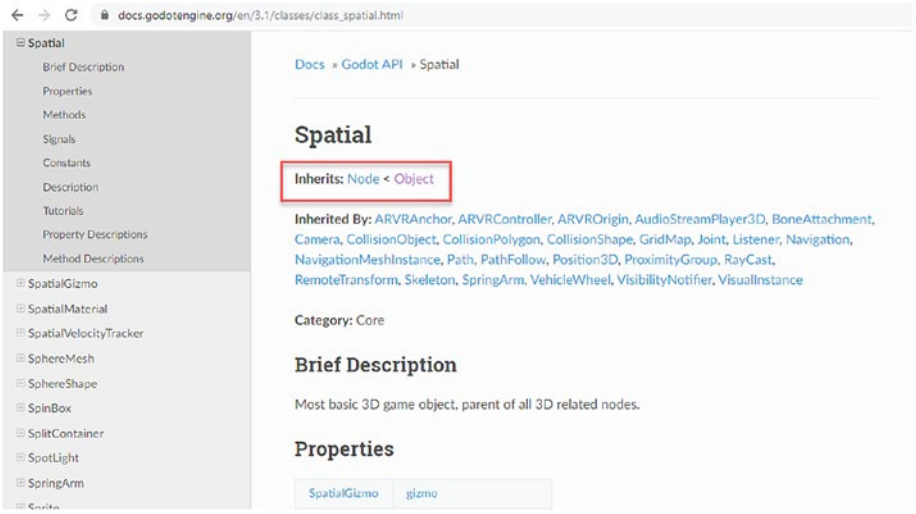

Figure 3-15. *Class Hierarchy Can Be Viewed from the Godot API*

Iterating Through Child Nodes

Being able to navigate the *SceneTree* effectively by code is critically important for creating gameplay, for understanding how a scene is working, and for accessing different objects in a scene. Let's start by looking at how a script on any Node can cycle through all direct children, one by one. There are two common methods given in Listings 3-3 and 3-4, respectively.

Cycling through child nodes is a valuable and powerful technique if you structure your scene tree cleverly. For example, by cycling through children, you can make inventory systems that keep track of all items the player has collected. You can count how many enemies are remaining in the scene. And you can determine which rooms are on the same floor. And lots more! Listings 3-3 and 3-4 achieve the same result. When you attach the scripts to nodes in the scene, it'll print the names of all children to the Output Window.

Listing 3-3. Printing Child Object Names – Method 1

```
using Godot;
using System;

public class NodeTraverse : Node
{
    public override void _Ready()
    {
        //Print all child node names, Method 1
        for(int i=0; i<GetChildCount(); i++)
        {
            Node N = GetChildOrNull<Node>(i);

            if(N != null)
                GD.Print(N.Name);
        }
    }
}
```

Listing 3-4. Printing Child Object Names – Method 2

```
using Godot;
using System;

public class NodeTraverse : Node
{    public override void _Ready()
    {
        //Print all child node names, Method 2
        Godot.Collections.Array ChildArray = GetChildren();

        foreach(Node N in ChildArray)
            GD.Print(N.Name);
    }
}
```

Finding Nodes by Name

Unity features the *GameObject.Find* function. This searches the scene hierarchy for the first object matching the specified name. Godot has an equivalent function. It's the *FindNode* function. This function searches all child nodes recursively for a matching node. Unlike the Unity equivalent of *GameObject.Find*, however, Godot's *FindNode* function only searches the entire scene tree *if* executed from the scene's root node. Otherwise, it only searches from the parent downward to all children. Finding nodes by name is useful when you need access to specific singular objects in the scene, like the player character, or the GameManager, or a Respawn Location. Listing 3-5 searches all children recursively for an object named "Player".

Listing 3-5. Finding Child Nodes by Name

```
Node PlayerNode = FindNode("Player", true);

if(PlayerNode != null)
    GD.Print("Player Node found!");
else
    GD.Print("Player Node not found!");
```

You can always search the entire scene for an object by name by simply searching from the root node. If your script isn't attached to the root node, you can always get access to it by using the *Node.GetTree* function. This function returns the complete scene hierarchy, allowing you access the root node. See Listing 3-6 to search the scene hierarchy for an object by name via the Root Node.

Listing 3-6. Finding Child Nodes by Name from the Root Node

```
SceneTree ST = GetTree();
Node PlayerNode = ST.CurrentScene.FindNode("Player");

if(PlayerNode != null)
    GD.Print("Player node found");
```

Finding Nodes by Path

An excellent feature of Godot is that each Node can be referenced by both an absolute and a relative path. You can select a Node from a C# script by using the *GetNode* function or by its *GetNodeOrNull* version. The former throws a game-breaking exception if the specified Node isn't found and the latter simply returns NULL in the same event. Listing 3-7 retrieves a root node called *root_level* inside the *_ready* event and prints its name to the Output. The scene hierarchy is shown in Figure 3-16.

Listing 3-7. Getting Access to the Root Node, Called root_level

```
public override void _Ready()
{
    Node N = GetNodeOrNull<Node>("root/root_level");

    if(N!=null)
        GD.Print(N.Name);
}
```

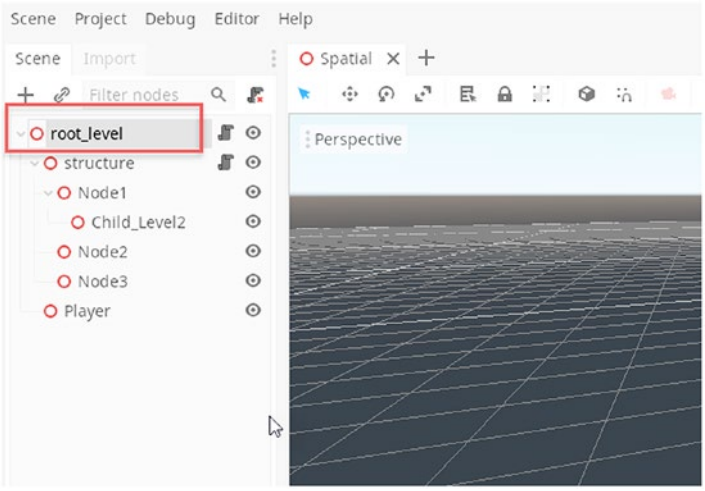

Figure 3-16. *Root Node in the Scene Tree*

Note Remember, if your root node is named differently from *root_level*, you'll need to specify it by name to access it via the GetNode or GetNodeOrNull functions.

You can also access nodes by using relative paths with the *GetNode* or the *GetNodeOrNull* functions. For example, Listing 3-8 uses the path "*../Player*" to get access to a *sibling* node named *Player*. The prefix (..) refers to the Parent node.

Listing 3-8. Accessing a Sibling Node by Name and Path

```
public override void _Ready()
{
    Node N = GetNodeOrNull<Node>("../Player");

    if(N!=null)
        GD.Print(N.Name);
}
```

Godot Groups vs. Unity Tags

If you're familiar with Unity, you'll remember that it has a **Tag** feature. This lets you label related objects like power-ups, enemies, or weapons. Tagging objects helps you find them easily in C# code because you can quickly search for objects with a matching tag. Most objects in Unity are *Untagged*, but some – like the Main Camera and the Player – are tagged by default. See Figure 3-17.

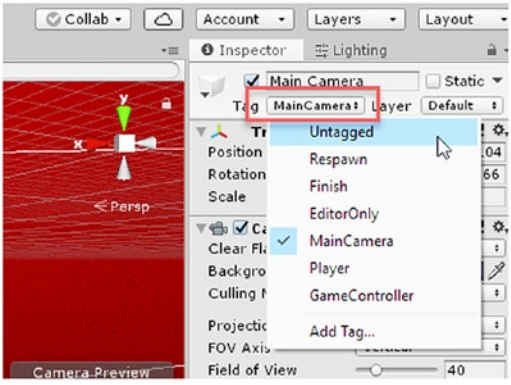

Figure 3-17. *Unity Tags Group Together Related Objects*

Godot supports equivalent tagging functionality through **Groups**. However, the Godot concept of Groups is more versatile and powerful than Tags, as we'll see. In Godot, a Group *is* a collection of objects, *but* an object may also belong to multiple groups. To create a new group, select any node in the scene and switch to the *Node Inspector* – it's next to the Object Inspector and it's easy to miss! See Figure 3-18.

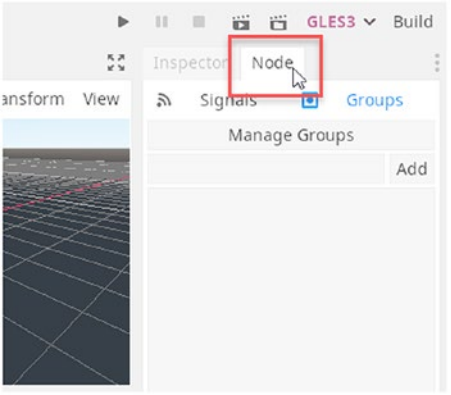

Figure 3-18. *Unity Tags Group Together Related Objects*

Click the *Groups* tag to show the Godot *Group* options. From there, click the *Manage Groups* button to show the *Group Editor Window*. From here, you can create and remove groups. See Figure 3-19.

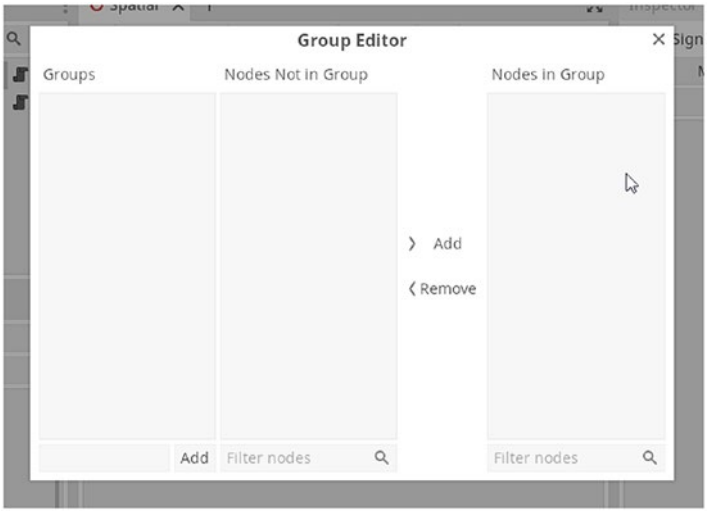

Figure 3-19. *Accessing the Group Editor*

From the *Group Editor*, type in a meaningful Group Name and click the *Add* button to create a new empty group. See Figure 3-20. Meaningful names may include power-ups, enemies, treasure, weapons, waypoints, doors, elevators, and more. It depends on your project. Your scene should have as many groups as needed. They're excellent for organizing nodes, and they make coding easier too.

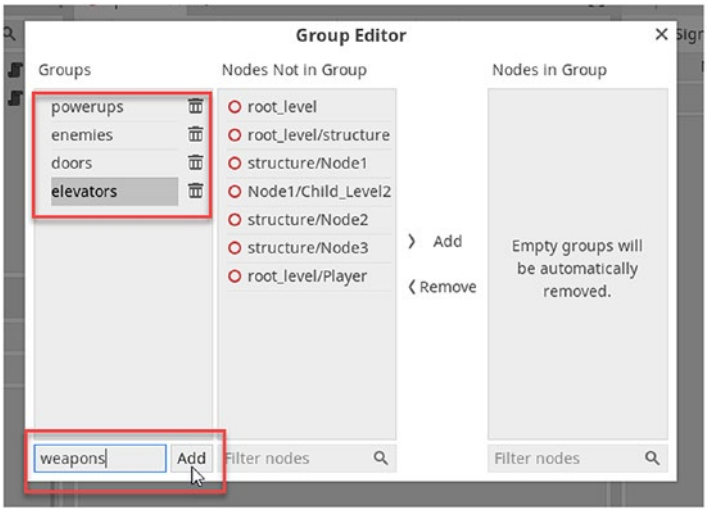

Figure 3-20. *Adding Groups Using the Group Editor*

After creating a Group in the Group Editor, you can add nodes to the group from the *Nodes Not in Group* column. Select the nodes to add and click the *Add* button (you can remove nodes using the *Remove* button). Remember, you can add Nodes to *multiple* groups if needed, for example, *Enemies* and *NPCs* groups or *Weapons* and *Equipment* groups. See Figure 3-21.

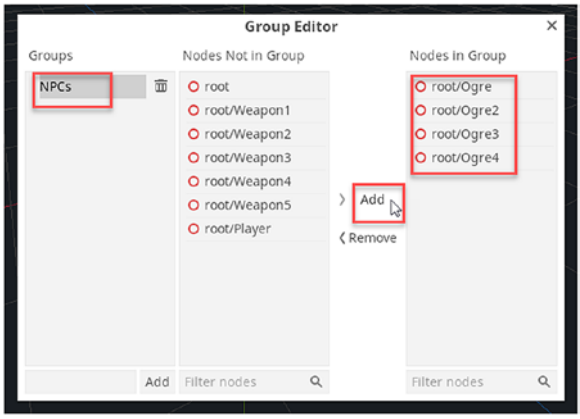

Figure 3-21. *Adding Ogre NPCs to the NPC Group*

Now that you can add nodes to groups in Godot, you can efficiently process those nodes in C# code. Specifically, the *SceneTree.GetNodesInGroup* function returns a list of all nodes in the specified named group. Listing 3-9 retrieves all nodes in the NPC group. The node list is returned as a *Godot.Collections.Array*. More information on the *Array* class can be found at the Godot API documentation here: https://docs.godotengine.org/en/3.1/classes/class_array.html.

Listing 3-9. Finding All Nodes in a Specified Group

```
//Getting nodes in group named 'NPC'
Godot.Collections.Array GroupedNodes = GetTree().
GetNodesInGroup("NPC");

//Cycle through all nodes in group, if any
foreach(Node N in GroupedNodes)
{
    GD.Print(N.Name);
}
```

You can also find all the groups to which a single Node belongs by calling the *Node.GetGroups* function. See Listing 3-10. This listing prints the names for all groups associated with the first found Ogre node.

Listing 3-10. Get All Groups Associated with a Selected Node

```
//Find first child node featuring 'Ogre' in the name
Node N = FindNode("*Ogre*");

//Get all groups to which this node belongs
Godot.Collections.Array Groups = N.GetGroups();

//Print all group names
foreach(string S in Groups)
    GD.Print(S);
```

Note You can also check if a node belongs to a specific named Group using the *Node.IsInGroup* function. This returns a Boolean, indicating whether the node is in the named group.

A really great feature is calling a named Function for all Nodes in a Group. This is achieved using the *SceneTree.CallGroup* function. This lets you run a function on all nodes within a named group. This is great for initiating group behaviors – like charging or fleeing NPCs who must act together at the same time – or for saving and loading scene data, for example, saving the state of a game for all objects in the scene. The *SceneTree.CallGroup* function uses reflection internally, and so it must be used as sparingly as possible for best performance. Avoid calling this function inside the *_process* event. See Listing 3-11.

Listing 3-11. Invoking the Method SaveData on All Objects in the NPC Group

```
public override void _Ready()
{
    //Save data for all NPC characters
    GetTree().CallGroup("NPC", "SaveData");
}
```

Accessing Variables in the Inspector

In Unity, the *public* keyword can be prefixed to a variable in C# – like an *Int* or a *Float* or a *String* – to make it fully readable and writeable in the Object Inspector. This means the public variable is shown in the inspector and can be edited from there as well as from code. This is convenient for many reasons; notably, you can edit a variable's starting value without having to recompile your code, and you can visually observe a variable changing

during gameplay to debug any potential problems. Godot supports similar functionality but *doesn't* use the scope specifier – *public*, *protected*, or *private* – to determine variable visibility. Instead, you must use the *[Export]* attribute for each variable that should appear in the Inspector. See Listing 3-12 for variable declarations and then the corresponding result in Figure 3-22.

Listing 3-12. Making Variables Accessible from the Inspector

```
public class Ogre : Spatial
{
    [Export]
    private string NPCName;

    [Export]
    public float NPCSpeed;

    [Export]
    public float NPCHealth;
}
```

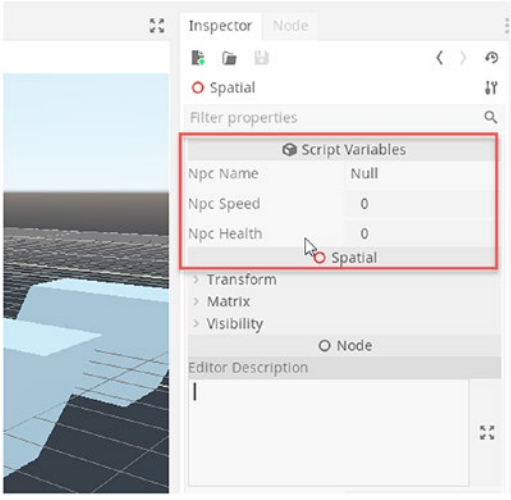

Figure 3-22. *Exposing Variables in the Inspector*

Variables As Properties – GetComponent?

OK, so you've created an *[export]* variable, as shown in the previous section. An Export variable becomes a **Property** of the Node. You can both see and edit properties from the Object Inspector. Each property gets its own unique identifier within the Godot tree – just like Nodes. For example, the following path */root/Spatial/Ogre:Health* refers to the *Health* variable on an object named *Ogre*. You can find the Property name of any variable by simply hovering your cursor over its name in the Inspector. The Property name will display in a pop-up context menu. See Figure 3-23.

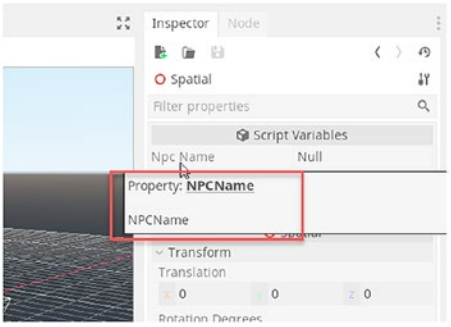

Figure 3-23. *Viewing Property Names*

Excellent! But wait, how can one script access the variables of another? That is, how can *two different script instances* (attached to *different* nodes) communicate and interact effectively with each other at runtime? In Unity, you'd often call *GetComponent* or *GetComponents* to retrieve direct references to other script instances. Godot is different. To access a script on any node – and its variables – you can use the *Typecast* method or the *Property* Access method. Let's see these in turn starting with *Typecasting*. See Listing 3-13. Here, we simply typecast the node to our intended data type (in this case, *NPC*) using the *as* keyword.

Listing 3-13. Typecasting Nodes

```
public override void _Ready()
{
    //Get NPC Object Named Ogre, with an NPC script
    Node N = GetTree().Root.GetNodeOrNull("/root/Spatial/Ogre");
    //Typecast object as type NPC to get script access
    NPC OgreNPC = N as NPC;
    //Set health of Ogre NPC
    OgreNPC.Health = 100f;
}
```

The other method for accessing variables is simple, doesn't involve any typecasting, and feels intuitive. But it's far from optimal and can lead to poor performance when used frequently and often. See Listing 3-14. This involves using the *Object.Set* function for setting a named variable.

Listing 3-14. Setting Properties

```
public override void _Ready()
{
    //Get NPC Object Named Ogre
    Node N = GetTree().Root.GetNodeOrNull("/root/Spatial/Ogre");

    //Set variable value
    N.Set("Health", 50f);
}
```

NodePaths and Node References

We've seen already how functions like *GetNode*, *FindNode*, and *GetGroups* can programmatically search for, and retrieve, references to specific nodes in a scene. This works well when you're retrieving nodes *in code*.

However, you'll sometimes want to specify nodes from the Inspector through a variable property, such as referencing the player character, or the game manager, or the UI system. In Unity, you'd do this by declaring a public *GameObject* or *Transform* variable in your script file, and then you'd drag and drop your GameObject from the Scene View into the associated Inspector slot. However, in Godot, you must use the *NodePath* object type instead, declaring this as an *Export* type. The NodePath doesn't reference a Node directly. Rather, it represents a fully qualified *path to a node*, which gets resolved dynamically when needed. See Listing 3-15.

Listing 3-15. Getting a Node from a Reference

```
using Godot;
using System;

public class ExportVar : Node
{
    [Export]
    public NodePath LinkToPlayer = null;

    //Actual reference to player
    private Node PlayerNode = null;

    // Called when the node enters the scene tree for the first
       time.
    public override void _Ready()
    {
        //Resolve the link and find the object
        PlayerNode = GetNode(LinkToPlayer);
    }
}
```

Listing 3-15 features the *Export* variable *LinkToPlayer*. This is of type NodePath. A NodePath variable doesn't line to a node directly. It must be resolved in code. For this reason, the *_Ready* event uses the *GetNode* function to convert the path to a node reference. See Figure 3-24, which shows a NodePath in the inspector where a node reference can be specified.

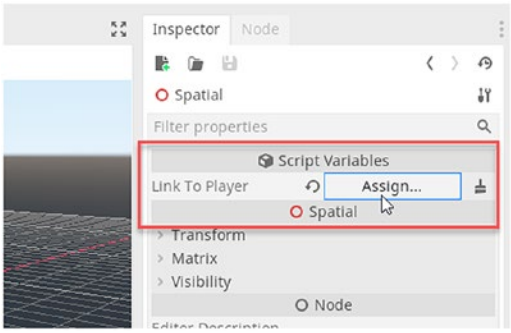

Figure 3-24. *Referencing a Node via a NodePath*

Set an Object's Position

The *Spatial* class is the base for objects that exist spatially in a 3D scene. This includes meshes, particle systems, cameras, lights, and others. When working with Spatial objects like these, you'll commonly want to set their position in the scene in terms of *X*, *Y*, and *Z*. In Unity, you achieve this using the *Transform.Position* variable. In Godot, you achieve this similarly with the *Transform.Origin* variable. The Transform class in Godot is accessed by value rather than by reference. See Listing 3-16 for a script that sets an object's position based on a Vector3 *PosRestrict* variable. Figure 3-25 illustrates what this class will look like in the Inspector when attached to a node.

Listing 3-16. Restricting an Object's Position

```
using Godot;
using System;

public class PostionRestrict : Spatial
{
    [Export]
    public Vector3 PosRestrict = Vector3.Zero;

  // Called every frame. 'delta' is the elapsed time since the
  previous frame.
  public override void _Process(float delta)
  {
      //Get access to the object's transform
      Transform T = Transform;
      //Set the position
      T.origin = PosRestrict;
      //Apply the changes
      Transform = T;
  }
}
```

Note Remember, the Transform member variable will always refer to the transform of the Node to which the script is attached.

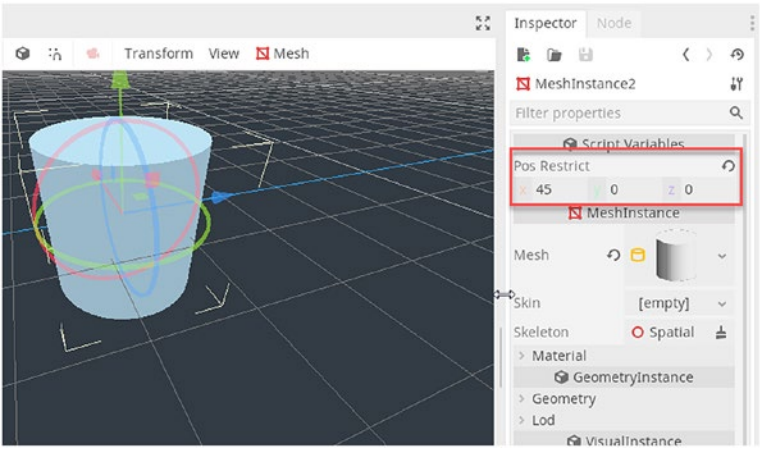

Figure 3-25. *Restricting an Object to a Position*

Make an Object Move Smoothly

The previous section demonstrated how to set an object's position absolutely. Now let's make an object move smoothly. That is, let's *change its position over time*. To be specific, every object faces in a specific direction, namely, its *forward direction*. When an object moves, it normally moves in the direction it's facing. This ensures an object always moves forward, no matter how it's orientated. To achieve this, we'll use the *Transform.Origin* variable along with some clever Vector math. Specifically, we'll offset the object's position along its forward vector, which is expressed by the variable *Transform.basis.z*. This is equivalent to Unity's *transform.forward* variable. Consider Listing 3-17, which moves an object forward continuously.

Listing 3-17. Moving an Object in C#

```
using Godot;
using System;

public class Mover : Spatial
{
    [Export]
    public float Speed = 5f;

    // Called every frame. 'delta' is the elapsed time since
    the previous frame.
    public override void _Process(float delta)
    {
        Transform T = Transform;
        T.origin += T.basis.z * Speed * delta;
        Transform = T;
    }
}
```

Note The Godot *delta* parameter of function *_Process* is equivalent to *Time.deltaTime* in Unity. It represents the amount of time, measured in seconds, since the previous frame completed.

Make an Object Rotate Smoothly

Now let's make an object rotate continuously and smoothly around its central axis, like a spinning power-up coin or a rotating sign. This approach is like making an object move, as demonstrated in the previous section. To make an object rotate, you'll need to rotate an object per frame by a rotational speed. Many rotation functions in Godot expect angles to

be specified in *Radians* and not *Degrees* – or *Euler Angles*. Consequently, if you're specifying angles in degrees, you'll need to convert them using the *Mathf.Deg2Rad* function. See Listing 3-18.

Listing 3-18. Rotating an Object in C#

```
using Godot;
using System;

public class Rotator : Spatial
{
    [Export]
    public float RotateSpeed = 90f;

    // Called every frame. 'delta' is the elapsed time since
    the previous frame.
    public override void _Process(float delta)
    {
        Transform T = Transform;

        T = T.Rotated(Vector3.Up, Mathf.Deg2Rad(delta *
        RotateSpeed));

        Transform = T;
    }
}
```

Detecting When an Object Enters a Trigger

Detecting when an object – like the player or an NPC – enters a 3D volume is useful. It can let us know when an object enters a room or an area in the scene or when the player falls into a lava pit and other scenarios. In Unity, the event *OnTriggerEnter* will fire for every *MonoBehaviour* script when a physical object enters a 3D volume. In Godot, the process for detecting

collisions or intersections is different from Unity. Let's start by configuring a 3D volume marking out an area in the scene. We'll be detecting to see if an object enters that area. To start, right-click the root Node in a new scene from the *Scene Tree* Panel. Then choose *Add Node* to add a node. See Figure 3-26.

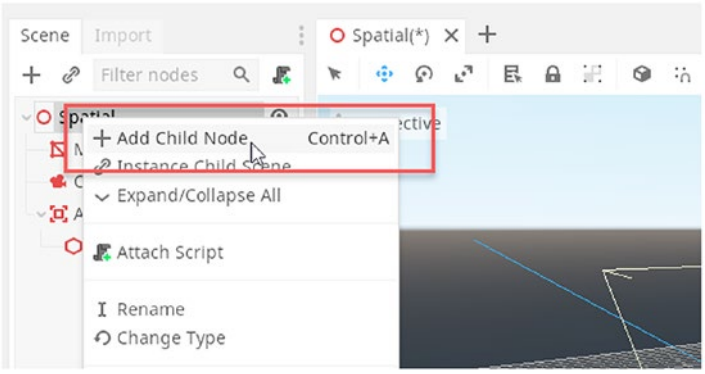

Figure 3-26. *Creating a New Node*

Next, create a new *Area* node by typing "Area" into the *Search* field of the Add Node Window. The Area node is useful for attaching physical forces, interactions, and collision detections to specific areas within a scene. Areas are invisible to the game camera; they simply mark out regions of 3D space. See Figure 3-27.

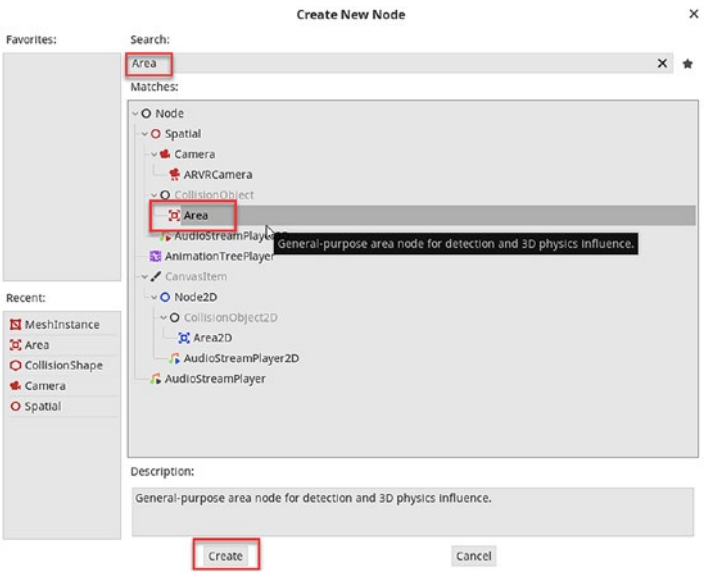

Figure 3-27. *Creating an Area Node*

By default, *Areas* lack width, height, and depth. They represent a physics anchor point in the scene and nothing more. To associate an *Area* with a *Volume*, you must create a *CollisionShape* child node. A Godot *CollisionShape* is equivalent to a Unity *Collider*. To create a *CollisionShape* as a child of an *Area*, right-click the newly created *Area* node from the Scene Tree Panel and then choose *Add Node*. From the Node creation menu, search for *CollisionShape*, and add a Collision Shape object. See Figure 3-28.

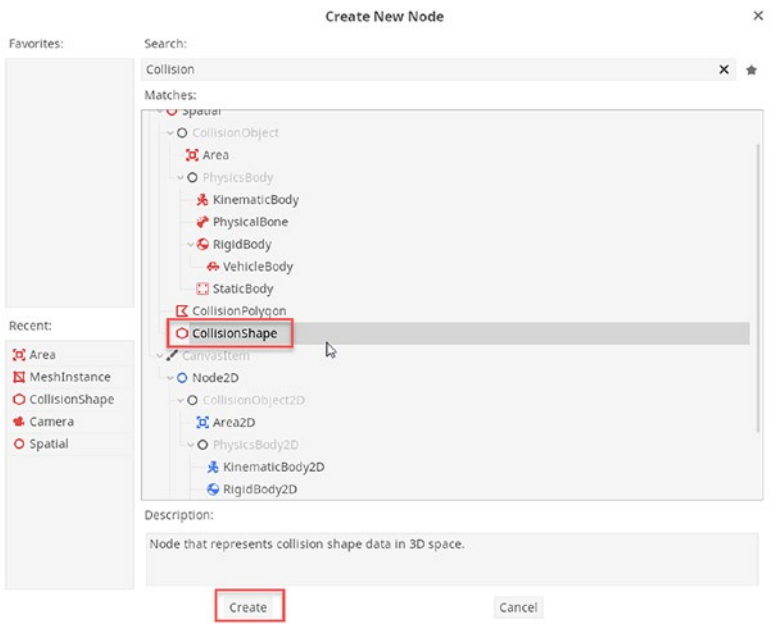

Figure 3-28. *Adding a Collision Shape Node*

Next, you'll need to choose the *Volume* to use for the collision area. This specifies the 3D volume inside which collision can be detected. To do this, select the *CollisionShape* object from the Scene Tree and click the *Shape* drop-down in the Inspector. From there, select a 3D volume. For our example here, let's select a Cube by choosing *New BoxShape* from the menu. See Figure 3-29.

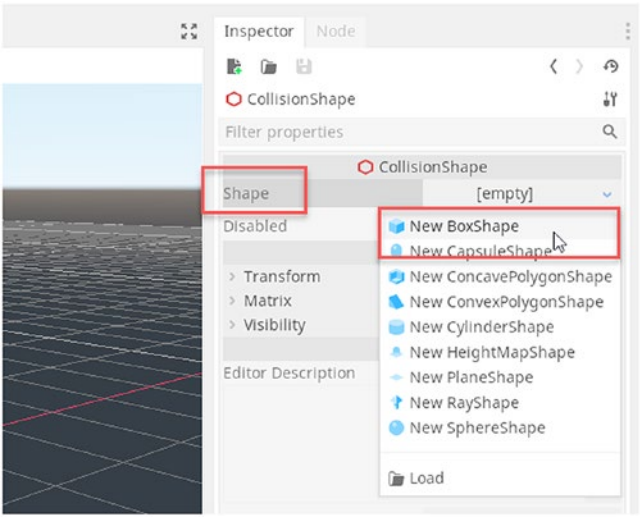

Figure 3-29. *Assigning a New Box Shape to the Shape Field*

When a fully configured *CollisionShape* features as a child of an *Area* Node, you'll end up with a complete Collision Detection area. See Figure 3-30.

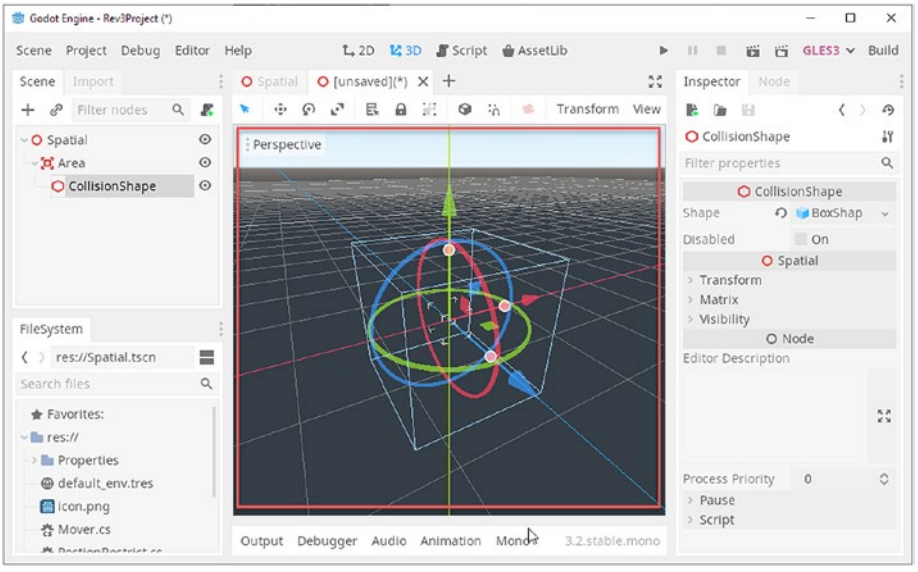

Figure 3-30. *Completing a Cube Collision Area*

Now, simply configure a test physics object, such as a *RigidBody* box, which can fall or enter the Collision Area. To create a physics box, simply add a new *RigidBody* node and then add a *Box Collision* shape as a child node. Then finally, add a Box *MeshInstance* as a child of the Box Collision object. This creates a *RigidBody* node network that can interact with other objects. See Figure 3-31.

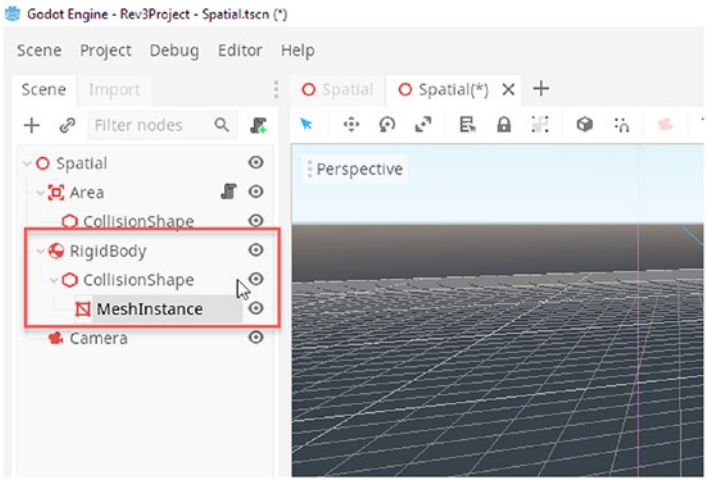

Figure 3-31. *Creating a Rigid Body Object*

Great. You've now configured the scene, ready to detect a collision. You have both a physics *Rigid Body* (a Cube) and a *Collision Area*. To detect when the rigid body enters the area, start by adding a new script onto the *Area* node. This script will handle the collision event. Next, select the *Area* object from the *Scene Tree*. Once selected, click the *Node* tab from the Object Inspector to view the Signals tab. See Figure 3-32.

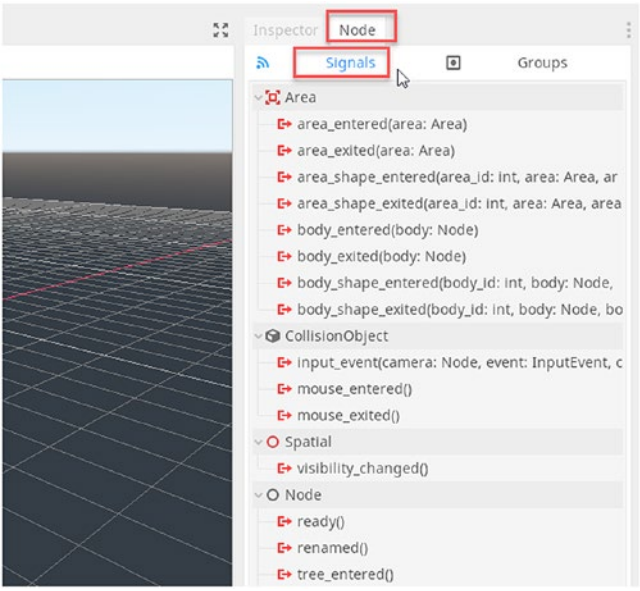

Figure 3-32. *Accessing the Signals Tab*

Signals in Godot equate to *Events* from Unity – or Delegates in C# – that you can connect to different functions in code, which execute when events happen. To detect when rigid bodies enter an area, first double-click the *body_entered* event from the Signals list. This event will execute once whenever a Rigid Body first enters a collider. It corresponds to Unity's *OnTriggerEnter* event. When you double-click the *body_entered* event, the *Signal Connector Window* appears. See Figure 3-33.

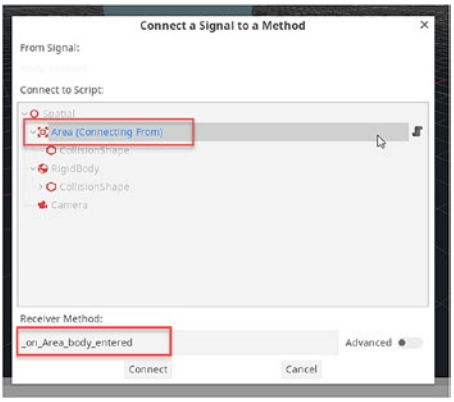

Figure 3-33. *The Signal Connector Window*

From the Signals Connector Window, select the Area object from the *Connect to Script* list. This node should be selectable. If it's not, check that it has a script attached. After selecting the *Area* node, enter a suitable name for the function that is to become the *Event*, using the *Receiver Method* field. Your associated script file should feature a function of a matching name, which will be called by Godot automatically whenever a Rigid Body first enters the area. You may need to add the function manually. See Listing 3-19 to see how a script file should be configured to support a *body_entered* event. Figure 3-34 demonstrates how the node connection should look when configured.

Listing 3-19. Detecting Collisions in C# Script

```
using Godot;
using System;

public class CollisionDetect : Spatial
{
    public void _on_Area_body_entered(PhysicsBody Node)
    {
        GD.Print("entered");
    }
}
```

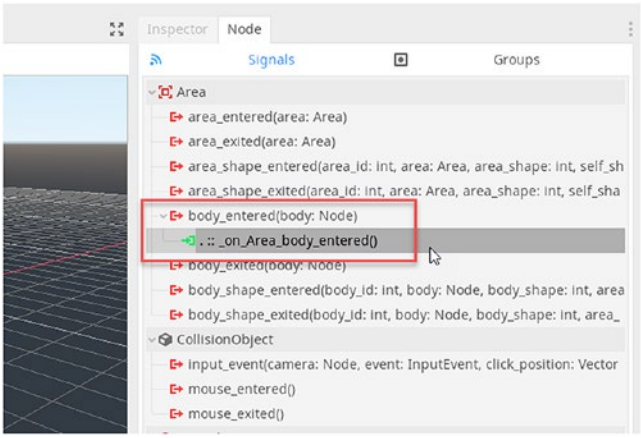

Figure 3-34. *Handling the Area Entered Event*

Great! Your Area is now configured to detect collisions with physical objects. Let's quickly create a *RigidBody* to react with, for testing purposes. To do this, add new *Rigidbody* node and then a cube collision shape as a child, and then finally add a child *MeshInstance* node, created as a Cube. Basically, that's three nested nodes: a *RigidBody*, a *CollisionShape*, and then a Cube *MeshInstance*. See Figure 3-35.

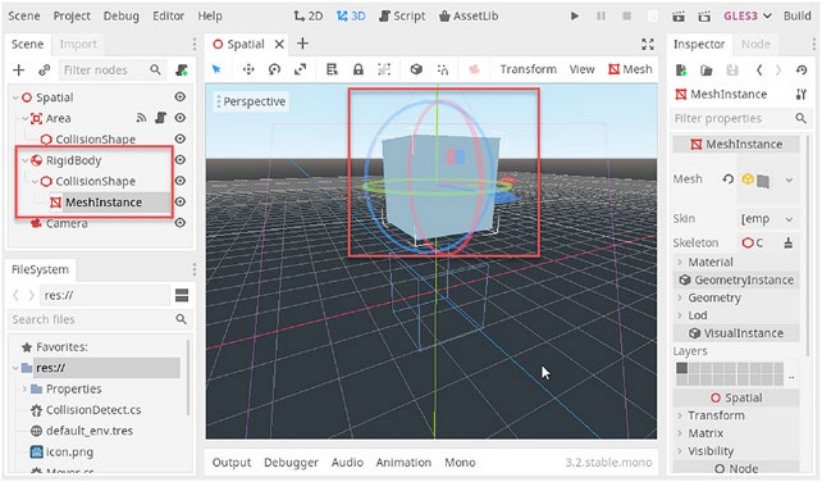

Figure 3-35. *Configuring a Rigid Body Object*

Now just hit *Play* on the toolbar and watch your rigidbody cube fall into the area below during gameplay. You may need to set up a scene camera first to get an ideal view of the interaction. You'll also see a message printed to the console when the collision happens. See Figure 3-36.

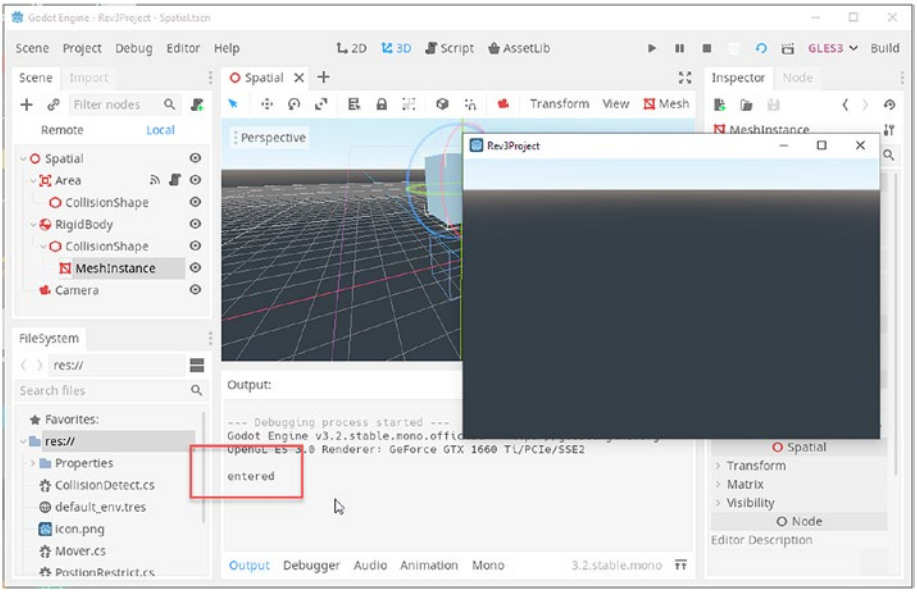

Figure 3-36. *Printing a Message on Object Entry*

Viewing Spatial Nodes

Godot's Spatial Node is, in many ways, equivalent to Unity's Empty Object. It marks a location in 3D space, has no visibility, and can be used – like a folder – to contain child nodes for organization. But, by default, Spatial nodes have no editor visibility either. This makes it difficult for developers to see them in the viewport and to select them by clicking. To select a Spatial node, you must always resort to clicking the object by name from the Scene Tree panel. This can be inconvenient. Instead, if you need to

view and select a Spatial Node from the view, you should use a *Position3D* node. To create one, simply right-click the root node from the *Scene Tree* Panel, choose *Add Node* to display the *Node Creation* Window, and then search for *Position3D* to add a new *Position3D* node. See Figure 3-37.

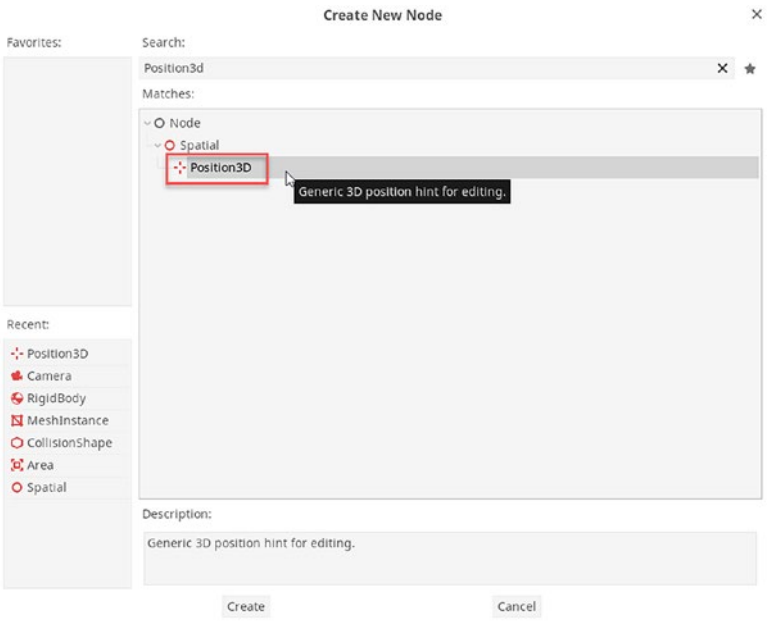

Figure 3-37. *Creating a Position 3D Node…*

After creating a *Position3D* Node, you'll always be able to see it inside the viewport and click it to select it. See Figure 3-38.

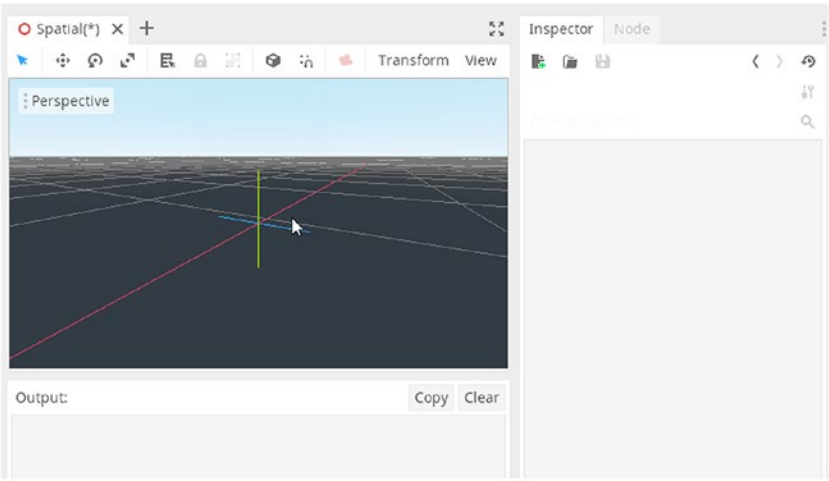

Figure 3-38. *Selecting a Position3D Node in the Viewport*

Reading Player Input

There are many ways to read input from multiple devices in Godot, including the keyboard, mouse, and gamepad. The cleanest method, and the closest to Unity's, is through the Input Mapping System. Let's see how this works. To start, access the Input Map settings by choosing *Project ➤ Project Settings* from the application menu. This displays the general project settings. See Figure 3-39.

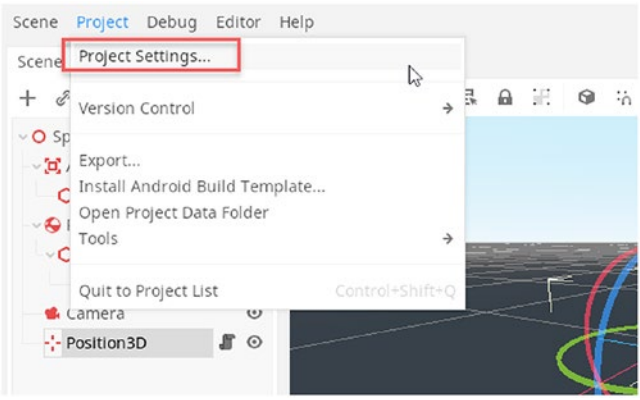

Figure 3-39. *Accessing Project Settings from the Application Menu*

From the *Project Settings* menu, choose the *Input Map* tab. See Figure 3-40. This displays a list of custom input axes, each associated with different input devices and buttons. Its purpose is to assign different buttons from different device types to a single unified input map that works effectively.

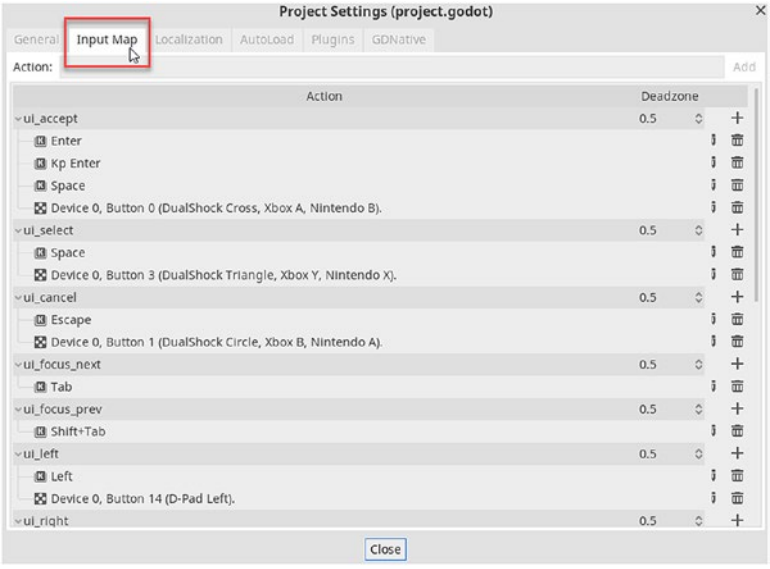

Figure 3-40. *The Input Map Associates Buttons with input axes*

From the *Input Map* tab, let's create a new axis to read input from the keyboard, detecting when the space bar key is pressed. This is useful for handling character-jump or character-shoot events. To start, type a new Axis name into the Action type field at the top of the Input Map tab. Let's call this Axis *fire*. Once entered, click the *Add* button to a new axis with the matching name. See Figure 3-41.

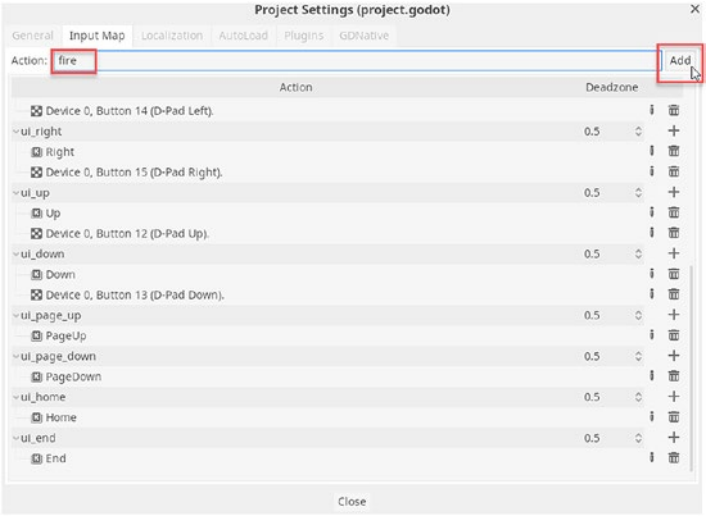

Figure 3-41. *Adding a New Fire Axis*

After adding the *fire* axis, click the + icon to associate a new key to it. From the context menu, select *Key*. See Figure 3-42. After selecting *Key*, simply press the relevant key on the keyboard to assign (here, we'll press the space bar).

Figure 3-42. *Adding a New Key to the Fire Axis…*

Great! You've now created a new Axis associated with a space bar press. You can then use C# to detect important axis events. Specifically, the *Input.IsActionPressed* function will return true *for as long as* the space bar is being held down. The *Input.IsActionJustPressed* function will return true once only on the first occasion of the space bar being pressed and the Input. *IsActionJustReleased* function will return true once only on the first occasion of the space bar being released. Listing 3-20 prints a message when the space bar is first pressed.

Listing 3-20. Detecting When the Space Bar Is First Pressed

```
public override void _Process(float delta)
{
    if(Input.IsActionJustPressed("fire"))
      GD.Print("Fired!");
}
```

This is excellent. You can now read input from specific button presses, such as jumps, shoots, interacts, and other forms of Boolean interactions. However, you'll also want to read analog data across a range of values in an axis. A classic example is left-right and up-down motion of a character. In Unity, you can use the *Input.GetAxis* function to read *Horizontal* and

Vertical data with smoothed values to drive character movement left and right and up and down. You can do the same in Godot too. To achieve this, add additional actions in the Input Map for *Left*, *Right*, *Up*, and *Down*. You can associate them with keyboard presses for WASD and arrows. See Figure 3-43.

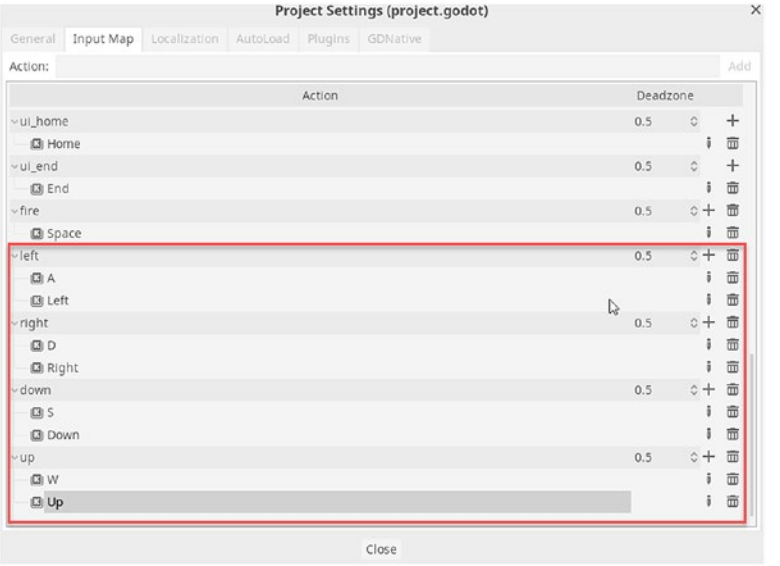

Figure 3-43. *Configuring Left, Right, Up, and Down Actions*

Next, you can read Horizontal and Vertical input values using Listing 3-21. The Horizontal Axis can return –1 (meaning left is being pressed), or 1 (meaning right is being pressed), or 0 (meaning neither direction is pressed). Similarly, for Vertical Input, 1 means Up is pressed, –1 means Down is pressed, and 0 means neither is pressed.

Listing 3-21. Reading Horizontal and Vertical Input

```
public override void _Process(float delta)
{
    float Horizontal = -Input.GetActionStrength("left") +
    Input.GetActionStrength("right");
    float Vertical = -Input.GetActionStrength("down") + Input.
    GetActionStrength("up");

    GD.Print("Vertical: " + Vertical + " Horizontal: " +
    Horizontal);
}
```

Summary

This chapter demonstrates the fundamentals of using C# in Godot to achieve critically important gameplay tasks. This includes searching for named nodes, traversing the node hierarchy, converting node paths to node references, exposing variables in the inspector, detecting collisions and reading input from the player, and others. By exploring the range of coding tasks illustrated here, you can now put together complex scenes with important gameplay behaviors.

CHAPTER 4

Making a 2D Game

Godot has a truly amazing 2D feature set for quickly and easily building 2D games. These games will normally export smoothly across platforms and run impressively. In this chapter, we'll explore Godot's 2D feature set by building a simple yet functional 2D game. Furthermore, we'll explore ways of working that Godot expects, and which differ from Unity. One of the biggest differences is philosophical, as we'll see. For example, in Unity, there's only one type of *Scene*, namely, a *3D Scene*. You can, of course, build 2D games and 2D worlds in Unity. But such worlds are, in truth, just flat objects aligned to the camera in a 3D scene, *only making it appear 2D*. Godot by contrast does distinguish between scenes and world spaces. Godot supports both 3D scenes and 2D scenes, and these do differ significantly. A 2D scene in Godot is *not* simply a flat object aligned to the camera in a 3D world. It's truly a hierarchy of 2D objects, positioned and measured in 2D space. This offers us both advantages and disadvantages as a developer. It makes 2D worlds intuitive and simpler to navigate during development but poses new challenges whenever we want to create mixed worlds featuring both 2D and 3D elements.

The 2D game we'll create in this chapter will be a top-down coin collection game. The player will move a 2D toon-sprite character around a tile-set world to collect coin pickup objects before the timer expires. To achieve this, we'll use a broad range of 2D features. Let's get started. See Figure 4-1 to see the completed game in action. The completed game is included in the book companion files.

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5_4

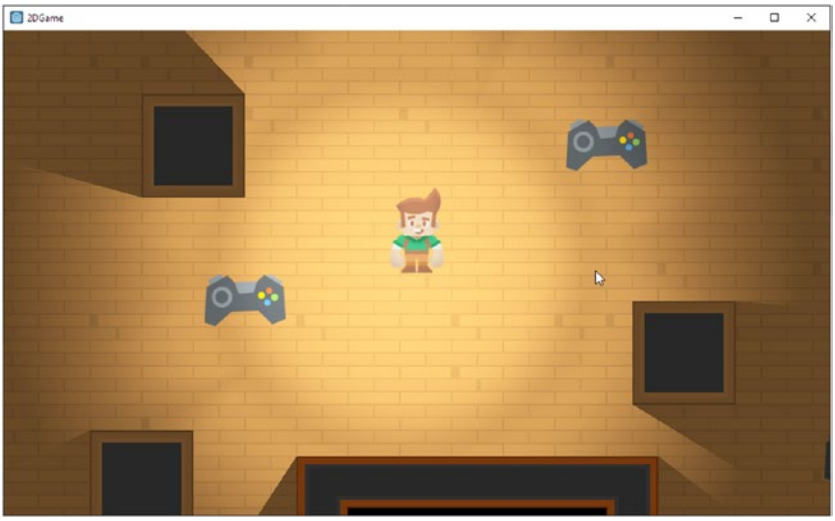

Figure 4-1. *Building a Top-Down 2D Collection Game…*

Configuring a 2D Project

To get started, we'll need to create a project and configure it appropriately. Simply create a new project with OpenGL ES 3.0 activated, choose a folder location for saving, and assign the project a suitable name, such as "*top-down collector*." Once created, you'll begin inside the Godot Editor. From here, access the Project Options by choosing *Project* ➤ *Project Setting*s from the application menu. Let's start by setting the Application Window size, that is, the complete dimensions of the game window in pixels. To do that, select *Display* ➤ *Window*, and then enter values of *1024 x 600* for the pixel width and height, respectively. See Figure 4-2.

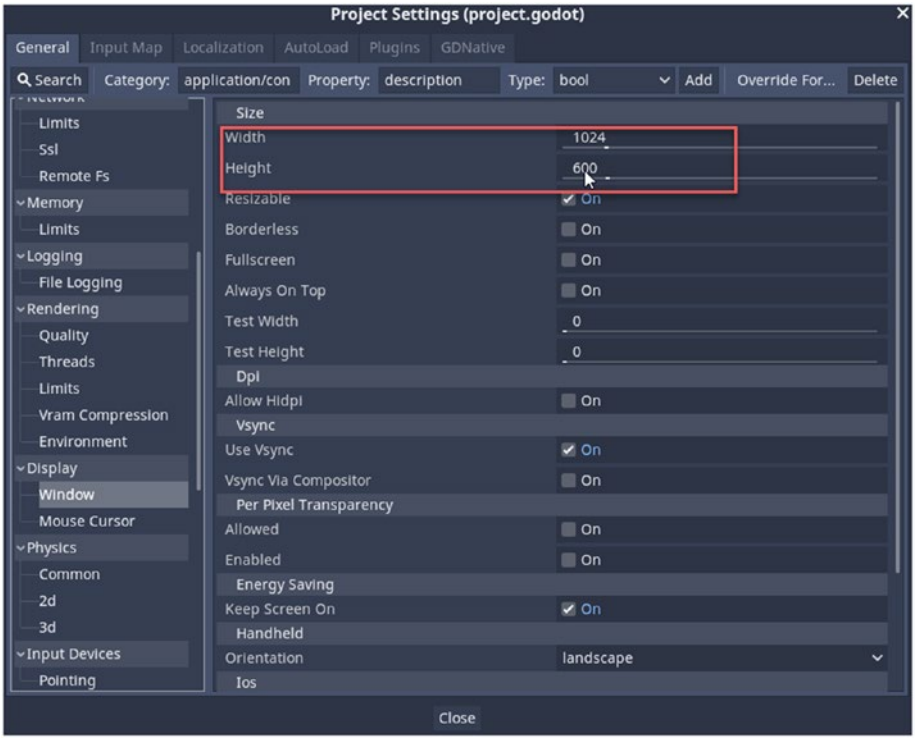

Figure 4-2. *Setting Window Width and Height*

Next, scroll down to the *Stretch* Section of the *Project Settings Window*. This determines how Godot should automatically render your scene when the window size changes from the intended resolution. This may be because the user resized the window or because the game is now running on a device with a different resolution. By default, Godot will maintain pixel sizes. This will be problematic if you want your 2D game to auto-resize or scale to a different resolution. To fix this, change the Mode drop-down to *2D* and the aspect to *keep_width*. See Figure 4-3.

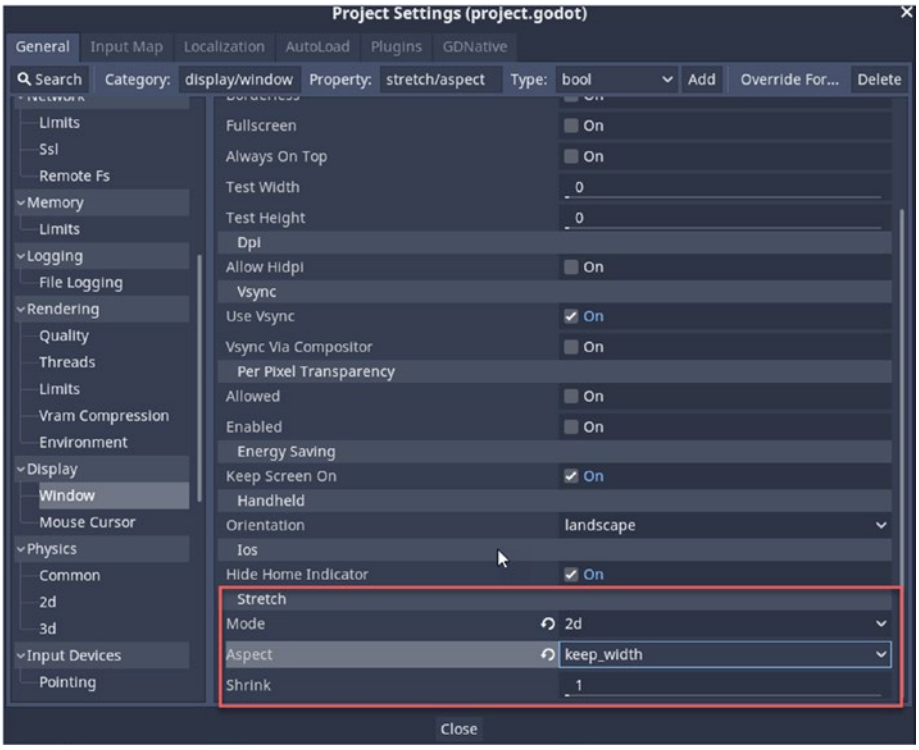

Figure 4-3. *Setting the Stretch Properties*

Our game will use *Tilemaps* to build the world, based on 2D tile images arranged in a grid. On some graphics cards – such as legacy NVIDIA cards – you may experience flickering from your tiles, especially as the camera moves around the scene. To avoid this bug, let's switch to the *Rendering* ➤ *Quality* tab and move to the *Depth* Section. From here, remove the check mark from the *HDR* field. See Figure 4-4.

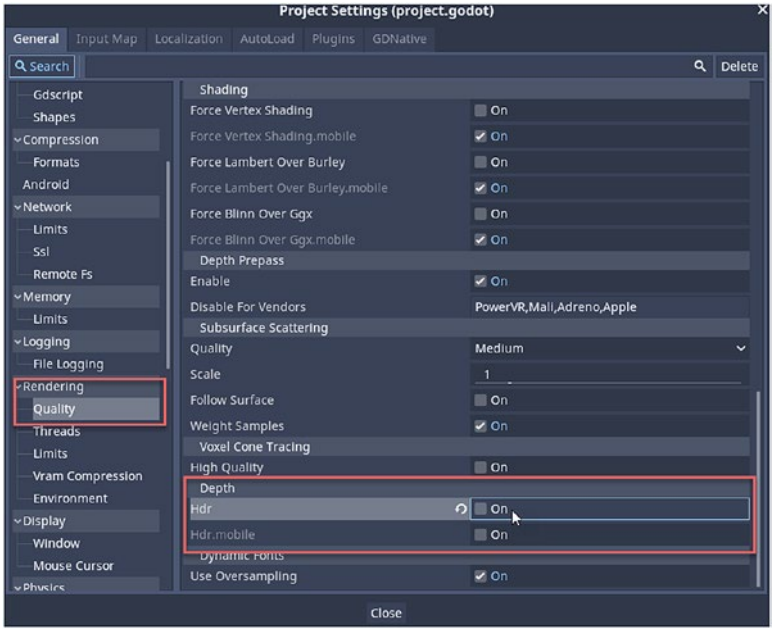

Figure 4-4. *Disabling HDR for Rendering Optimizations in 2D*

Now let's set up the keyboard input for up, down, left, and right controls. These span the Horizontal and Vertical axes, corresponding to the keys WASD (and the corresponding arrow keys). Chapter 3 explores how to configure these. Figure 4-5 shows the final key configuration for our 2D game.

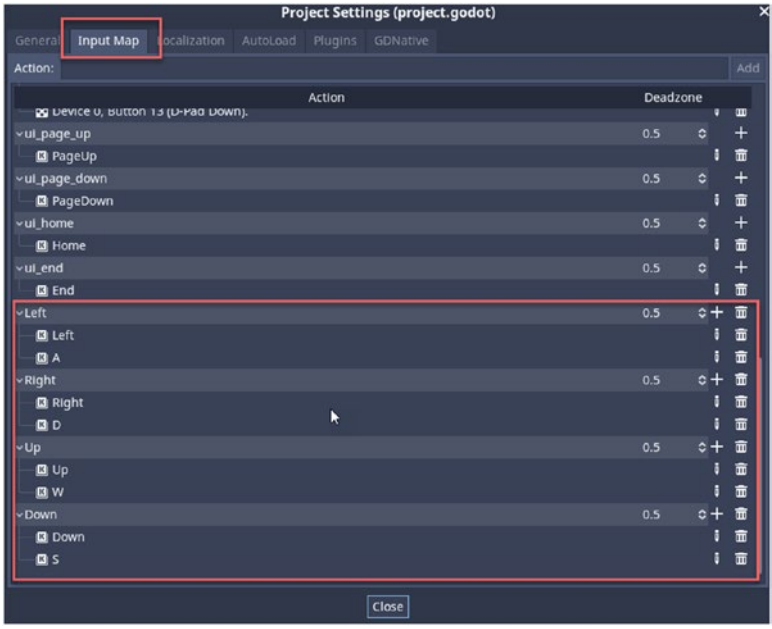

Figure 4-5. *Setting Input for the 2D Collection Game*

Importing Assets

For our 2D game, we'll use the freely available public domain assets from *kenney.nl*. For character sprites, we'll use `www.kenney.nl/assets/toon-characters-1` and for environment tile sets `www.kenney.nl/assets/topdown-shooter`. Later, we'll also use `www.kenney.nl/assets/generic-items` for pickups. Kenney is an excellent online source for quick prototypical 2D game art. Other great resources include `https://opengameart.org/`, `https://texturehaven.com/`, and hdrihaven.com. Each pack contains a selection of images in PNG format. And within each pack, there will be a single sprite sheet, with each smaller image composed into a single larger atlas image. We'll import the larger sprite sheet into Godot, and these are included in the book companion files too. See Figure 4-6.

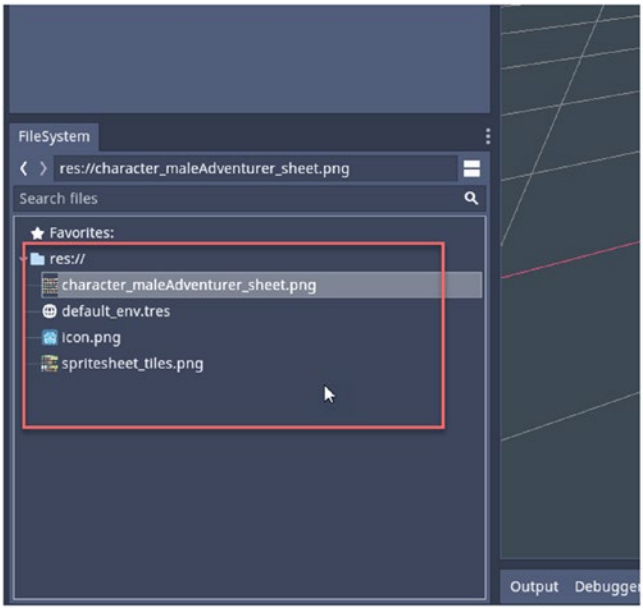

Figure 4-6. *Importing PNG Image Textures into the Project*

After importing the textures, select each one from the *FileSystem* panel and then move to the *Import* tab above. This allows you to change the import settings for the selected texture. For the Character sprite sheet – and for any objects that'll move or rotate during gameplay – ensure *Filter* is *enabled*. For environmental assets, and tile-sets, and other 2D assets that don't move during gameplay (walls, floors, tables, etc.), ensure *Filter* is *deactivated*. Don't forget to click the *Reimport* button if you make any changes. See Figure 4-7.

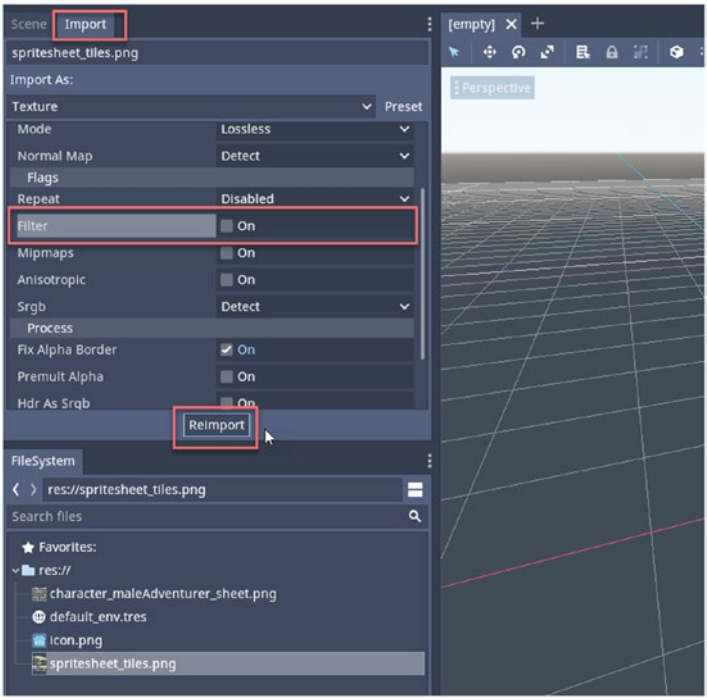

Figure 4-7. *Reimporting Textures*

Creating the Player Character

Let's start development by creating the player character. The character will be able to move around the environment using the WASD keys, collide with objects like walls and props, and collect power-up items. In Godot, *each distinct* gameplay element should be created *in its own scene*. For this reason, we'll need at least a player scene, an environment scene, and a collectible scene. During gameplay, these scenes will be merged, or linked, together to create a master scene containing our combined gameplay elements. Scenes, in this sense, then, should be conceived as stand-alone, self-contained gameplay units. Let's create the player scene by choosing *Scene* ➤ *New Scene* from the application menu and then click *2D Scene* from the *Scene* tab to create the initial starting node for the scene. See Figure 4-8.

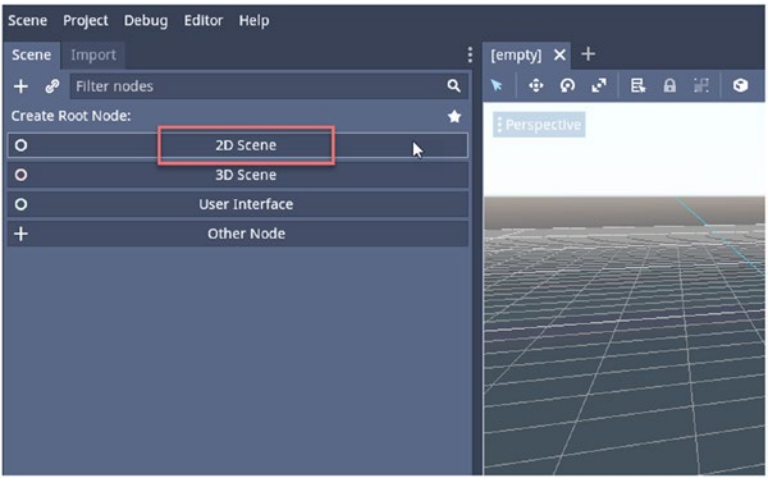

Figure 4-8. *Creating a 2D Scene for the Player Character*

The player root node should be a *KinematicBody2D*, because it'll be player controlled and should react with physical objects. Right-click the scene root, and then choose *Add Child Node*. From here, create a *KinematicBody2D*. See Figure 4-9.

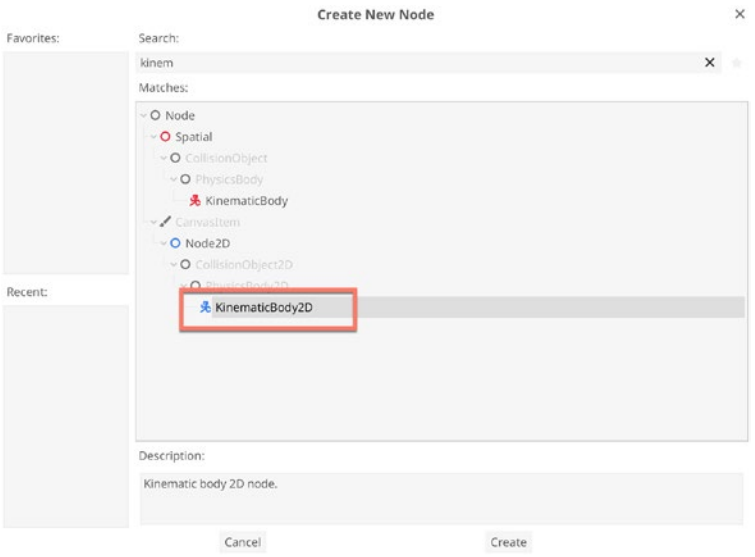

Figure 4-9. *Creating a Kinematic Body As the Player Root*

> You can make any node the scene root by right-clicking the node from the Scene Tree and then selecting *Make Scene Root* from the context menu. For this example, go ahead and make our newly created Kinematic Body the scene root.

The *KinematicBody2D* has no physical or visible presence when created on its own. It must be complemented by two additional child nodes: one to define the character's volume and size physically and the other to define its appearance to the player. To achieve this, let's create two child nodes: first, the *CollisionShape2D* node and, second, the *AnimatedSprite* node. See Figure 4-10 for the completed scene tree.

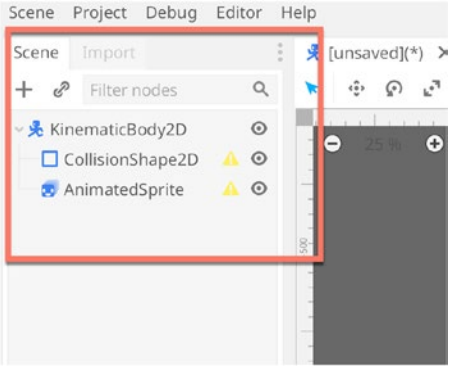

Figure 4-10. *Creating a Kinematic Body As the Player Root*

Let's proceed by defining the Sprite Appearance through the AnimatedSprite node, that is, the 2D appearance of the player character. We're using an AnimatedSprite node, as opposed to a regular Sprite node, because actions like walking should display a spite animation. Select the *AnimatedSprite* from the Scene Tree, and from the *Frames* field in the Object Inspector, choose *New SpriteFrames*. See Figure 4-11.

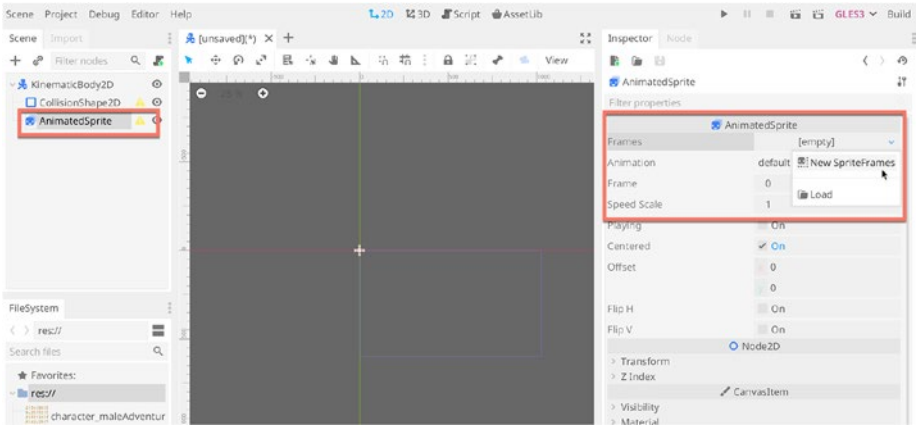

Figure 4-11. *Creating New Sprite Frames…*

By clicking the newly created *SpriteFrames* Resource in the Inspector, you'll display the SpriteFrames Window at the bottom center of the Godot interface. This lets you edit the SpriteFrames resource, adding animations and frames from a pixel-based sprite or sprite sheet. See Figure 4-12. To create our first animation, click and rename the default to *Idle*. And then click the *Add Frames from Sprite Sheet* button. From here, you can select a texture asset (*character_maleAdventurer_sheet.png*) from our available resources.

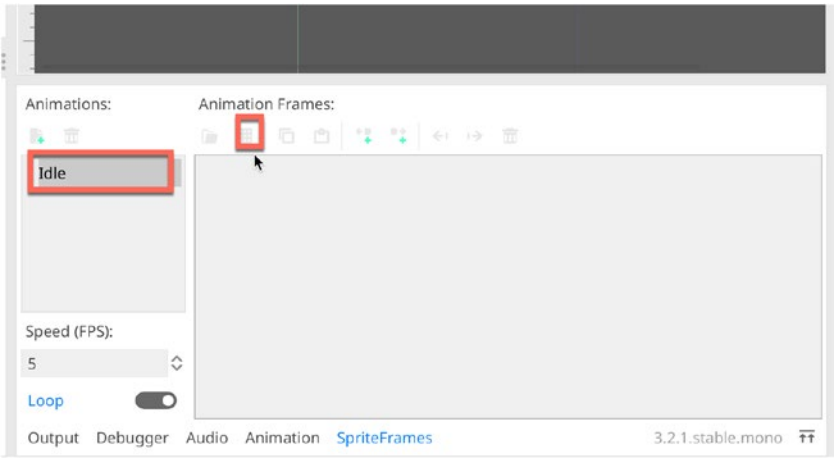

Figure 4-12. *Adding Frames from a Sprite Sheet*

Clicking the *Add Frames from Sprite Sheet* button displays the *Select Frames* Window, where you can slice the sprite sheet into even tiles, based on rows and columns. For our character sprite image, a horizontal slice of 9 and a vertical slice of 5 works. This divides the image successfully into evenly sliced tiles. Once sliced, simply hold down the Ctrl key (or *Cmd* key on a Mac) to *left-click* and select all frames to be included in the Idle animation. The *Idle* animation defines how the character will animate when standing still. See Figure 4-13. When done, click the *Add Frames* button.

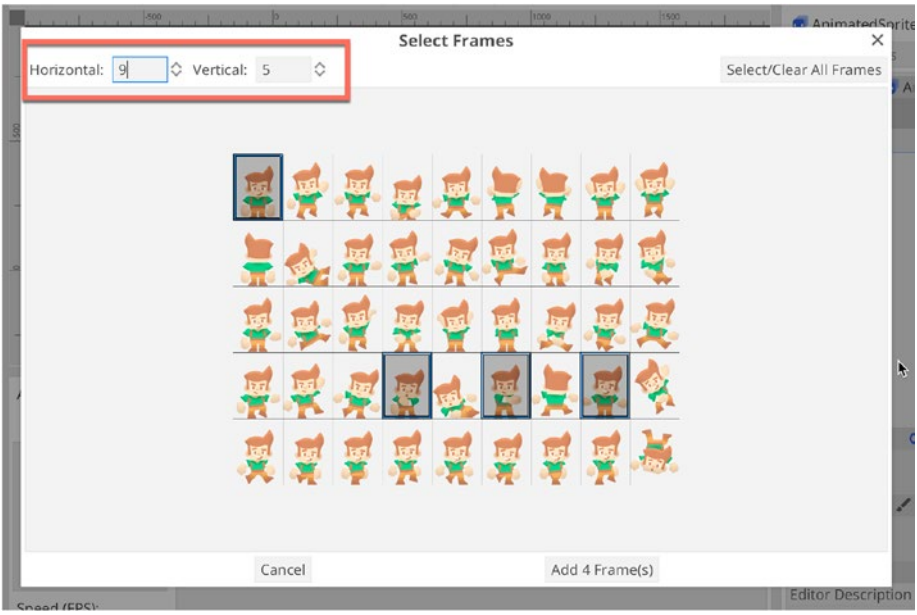

Figure 4-13. *Selecting Animation Frames*

Now you can use the *Sprite Frames Window* to arrange the frames and define a frame rate for the animation. Use the *Copy* and *Paste* buttons to duplicate frames where needed and the *Move Before* and *Move After* buttons to rearrange the order of frames. See Figure 4-14.

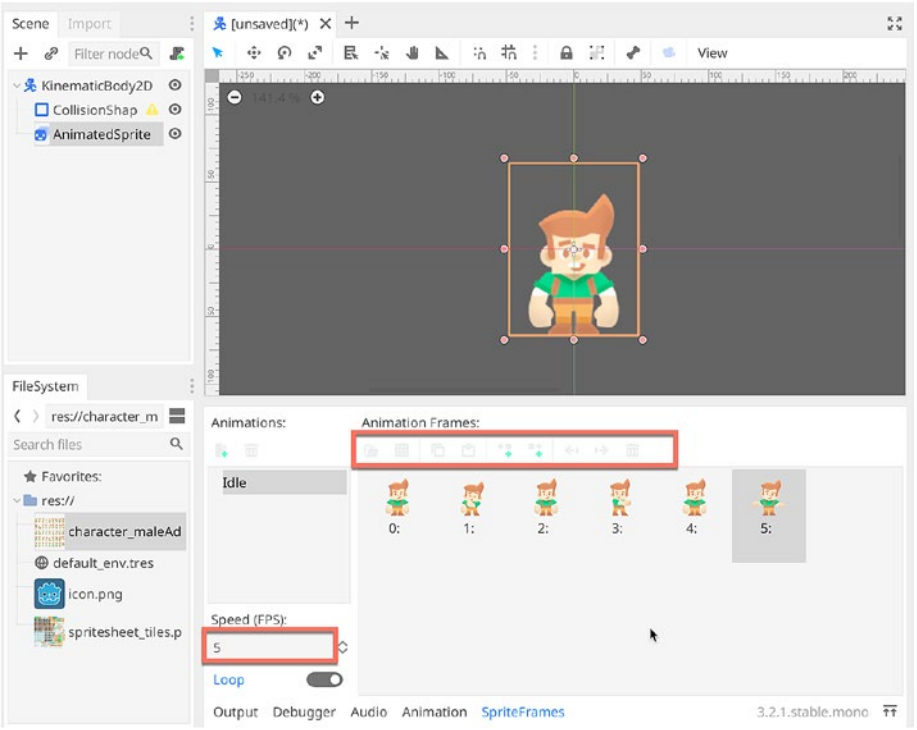

Figure 4-14. *Defining an Idle Animation Using the Sprite Frames Window*

Repeat this process now for creating a *Walk* animation, which will play whenever the player character moves. Simply hit the *New Animation* button to add a Walk animation, and then add frames via the preceding method. See Figure 4-15.

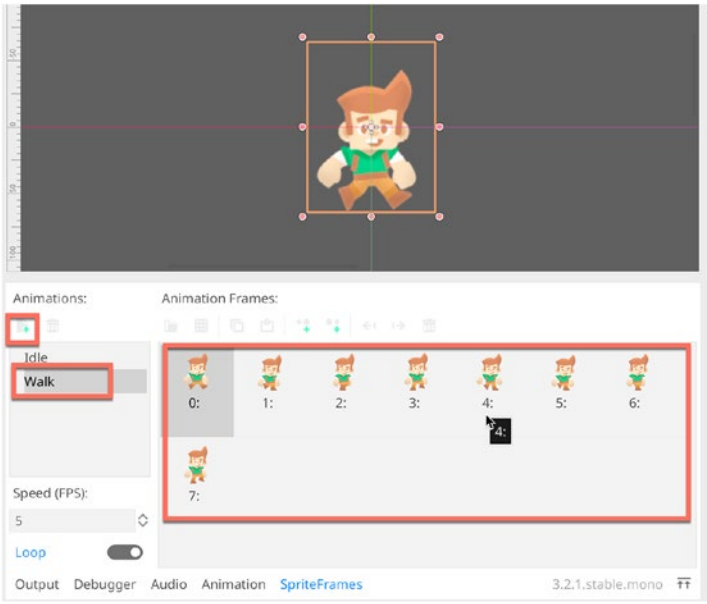

Figure 4-15. *Creating a Walk Animation*

You can preview each animation directly in the viewport. To do this, select the *AnimatedSprite* node, and from the Inspector, enable the *Playing* check box and select the relevant animation to preview from the *Animation* field. See Figure 4-16.

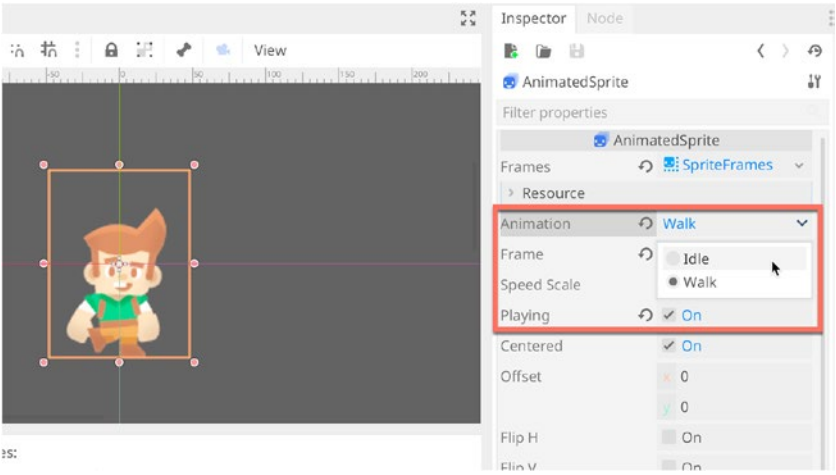

Figure 4-16. *Previewing Character Animations*

Excellent! We've now got an animated character, complete with two distinct animations. Of course, the *Idle* animation should be the default, as it'll always play unless the character is moving. However, the character still doesn't have any physical size, shape, or volume. Let's address that by selecting the *CollisionShape2D* node from the *Scene Tree* panel. With this object selected, click the *Shape* field from the Object Inspector, and choose *New CircleShape2D* from the context menu. See Figure 4-17.

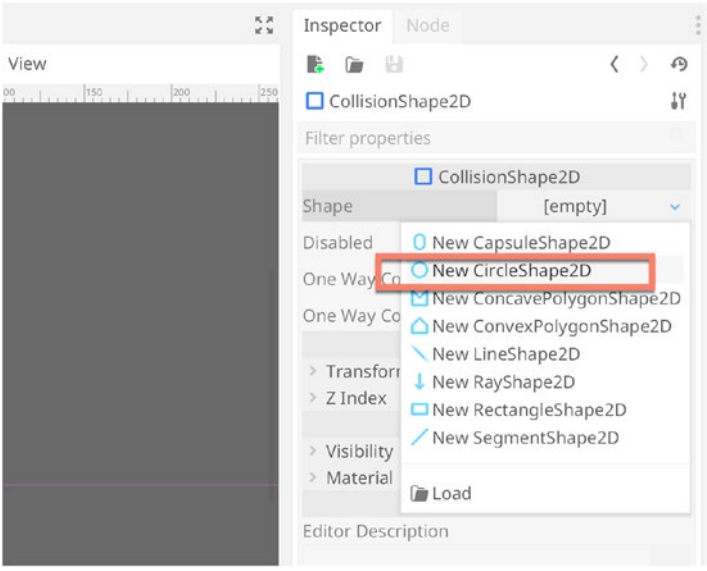

Figure 4-17. *Creating a Collision Shape for the Player Character*

After creating a new *CircleShape2D*, you'll be able to resize the radius to enlarge or shrink the circle, approximating the character shape and size. You can resize the circle by adjusting the *Radius* field or by directly dragging the circle gizmo in the editor. See Figure 4-18. As you adjust the Radius, you'll notice the *AnimatedSprite* is probably centered on the object pivot point, making the character pelvis or mid-region centered at the origin. This is not ideal. So, we'll change that next.

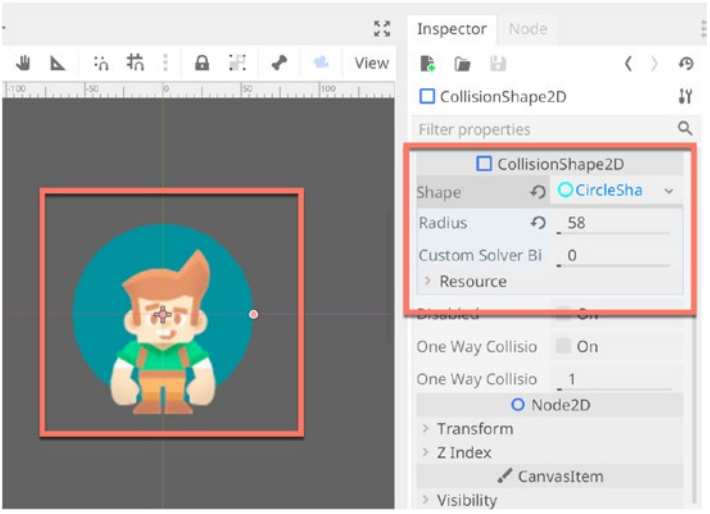

Figure 4-18. *Creating a Collision Shape for the Player Character*

Select the *AnimatedSprite* character, and then deactivate the *Centered* check box. Next, use the *Offset X* and *Y* fields to recenter the character, so his feet touch the origin instead. This'll make it easier to predict where the character will appear when we set his position during gameplay. See Figure 4-19.

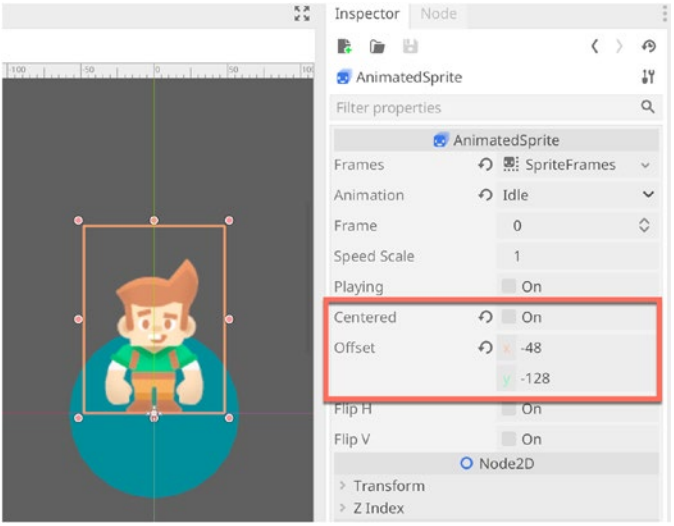

Figure 4-19. *Recentering the Player Character*

Now, reselect the *CollisionShape2D*, and adjust the collision shape to match the character's body. You don't need to enclose the head, because when seen from a top-down perspective in a regular 2D level, it's only the body that will collide with walls, doors, and objects. See Figure 4-20.

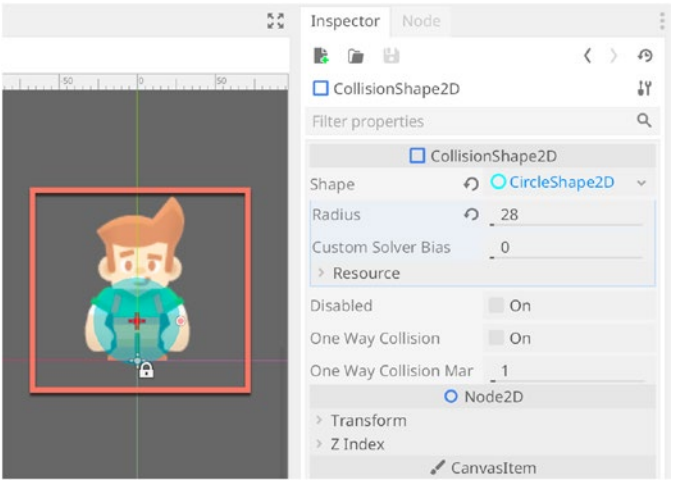

Figure 4-20. *Sizing the Circle Collider*

This is great. We've got a fully configured player character. And now, finally, we can code player controls. To do this, select the root *KinematicBody2D* node, and create a new C# script from the *Script* field in the Inspector. The script should inherit from *KinematicBody2D* and be called *PlayerControl*. See Figure 4-21.

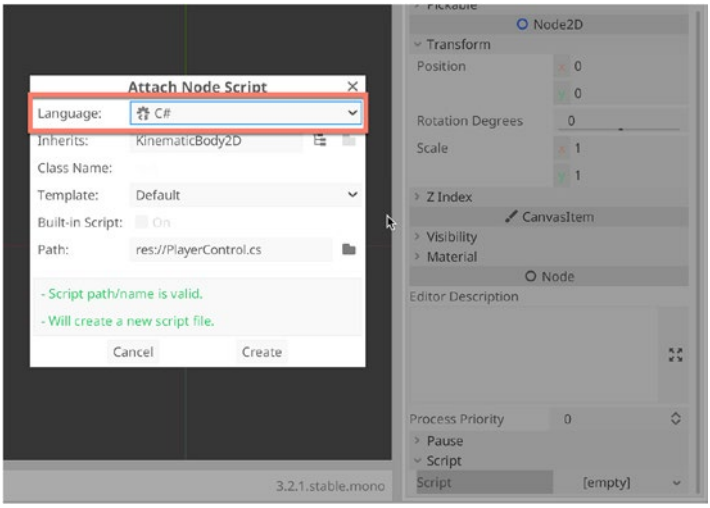

Figure 4-21. *Creating a New C# Script for Player Controls, Attached to the KinematicBody2D*

The purpose of the *PlayerControl* script is to respond directly to user input – *up*, *down*, *left*, and *right* – and to ensure the player interacts with physical obstacles in the scene, like walls and doors, and props. The complete source code for the player character is given in Listing 4-1. Comments follow.

Listing 4-1. Player Controls

```
using Godot;
using System;

public class PlayerControl : KinematicBody2D
{
    [Export]
    public float MoveSpeed = 2f;

    private Vector2 InputDir = Vector2.Zero;
    private AnimatedSprite AnimSprite = null;
    // Called when the node enters the scene tree for the first
       time.
    public override void _Ready()
    {
        AnimSprite = GetNode("AnimatedSprite") as
        AnimatedSprite;
    }

    public override void _Process(float delta)
    {
        InputDir.x = -Input.GetActionStrength("Left") + Input.
        GetActionStrength("Right");
        InputDir.y = Input.GetActionStrength("Down") + -Input.
        GetActionStrength("Up");

        if(InputDir.x > 0) AnimSprite.FlipH = false;
        if(InputDir.x < 0) AnimSprite.FlipH = true;

        InputDir = InputDir * MoveSpeed;

        if(InputDir.LengthSquared() <= 0)
            AnimSprite.Play("Idle");
```

```
        else
            AnimSprite.Play("Walk");
    }

    public override void _PhysicsProcess(float delta)
    {
        MoveAndSlide(InputDir);
    }
}
```

Listing 4-1 contains the full and complete *PlayerControl* script. The *_Ready* function executes on startup and finds the *AnimatedSprite* node. The *_Process* function runs on each frame to read input from the keyboard, to flip the sprite left or right if needed, and to trigger the appropriate animation in the *AnimatedSprite* node. The collected input is expressed as a 2D velocity in the Vector *InputDir*, which is used inside the *_PhysicsProcess* event by the *MoveAndSlide* function. To draw a parallel, Unity divides a "frame" across both *Update* (on each rendered frame) and *FixedUpdate* (each physics step). Godot, by contrast, uses *_Process* for rendered frames and *_PhysicsProcess* for physics steps. This means that all physics-based functions – like *MoveAndSlide* – must happen inside *_PhysicsProcess*, if they are always to behave as intended. Calling a physics-based function outside the *_PhysicsProcess* event may mean collisions aren't always detected, characters could pass through solid objects, and physics collisions could result in jitter and stutter.

Now, to complete the player character, let's add a follow camera. This will track the player as they move around the scene, ensuring the player is always visible. Right-click the *KinematicBody2D* node and add a *Camera2D* node. See Figure 4-22.

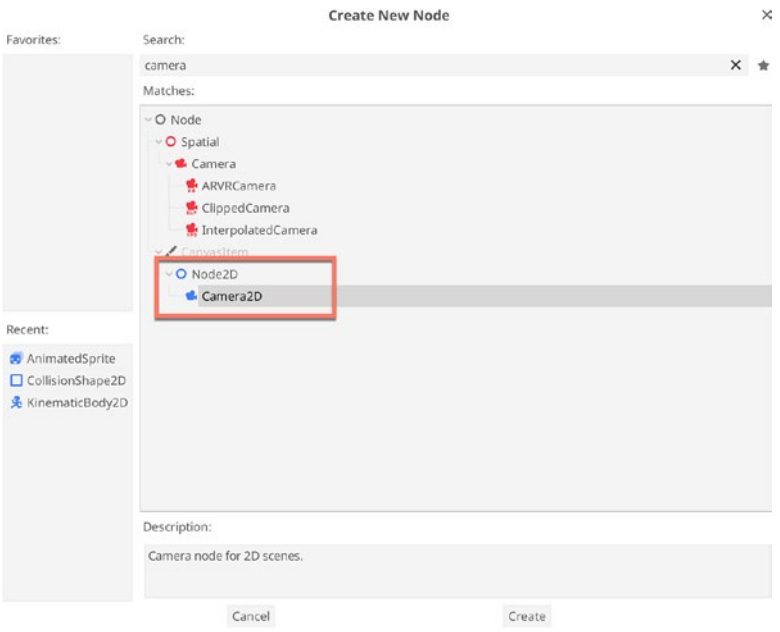

Figure 4-22. *Adding a Camera2D Node to Become a Follow Camera*

After adding a *Camera2D* node as a child of the *KinematicBody2D*, enable the *Current* boolean setting from the Inspector. This ensures the Camera is always the main active camera in the scene. In addition, enable *Smoothing* to create a smoothed camera motion as the camera tracks its target, and set its *Process* mode to *Physics*, since the camera motion is based entirely on player movement, which uses the physics system. See Figure 4-23. Great work! We've now got a fully functional player.

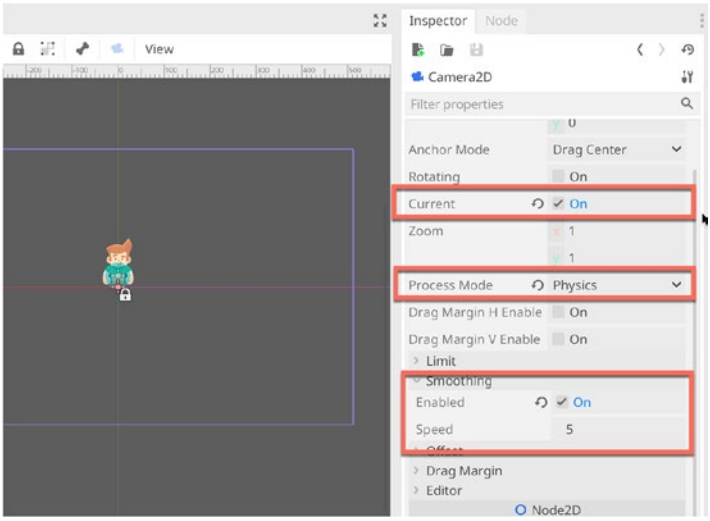

Figure 4-23. *Configuring the Camera to Follow the Player*

The player is done! Save your changes. Now create a new master scene, if you haven't already, and drag the player scene into it. The Master Scene will contain all top-level gameplay units of our game, including the player and the level. See Figure 4-24.

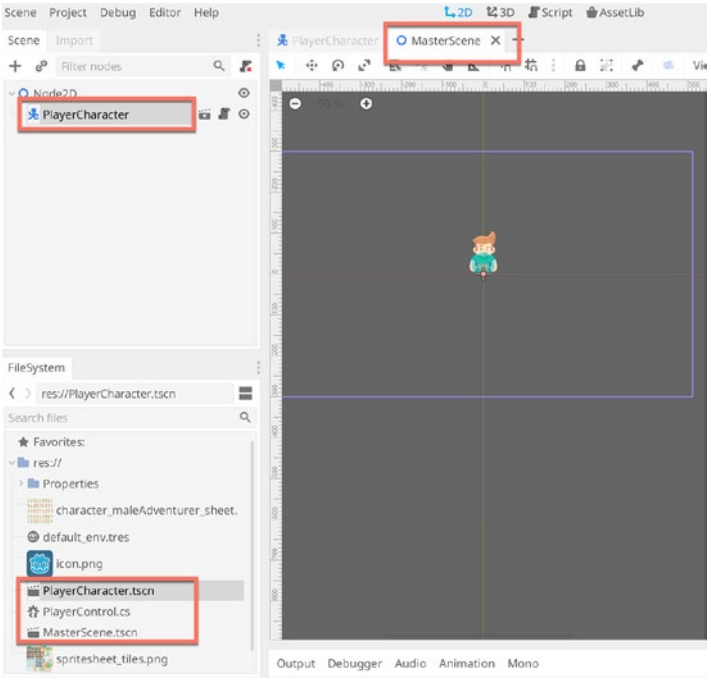

Figure 4-24. *Setting Up a Master Scene*

Building a Level – Tilemaps and Tilesets

Our 2D Collection game needs a world – or an environment – where the player can explore and find things to collect. In 2D, worlds can be made easily using a Tilemap, which is a grid of equally sized image tiles, arranged in rows and columns, and reused optimally to form a single complete world. Many famous games were constructed in this way, including the early *Zelda* and *Final Fantasy* games. Nearly every top-down RPG game is made from a tilemap, where each level one is simply one big tilemap. A Tilemap in Godot is composed from a Tileset, which is a palette of tile images. To build our level, let's create a new 2D scene and add a *Tilemap* node. See Figure 4-25.

Unity also supports TileMaps and offers feature equivalence to Godot. However, Godot also supports GridMaps, which are tilemaps for 3D scenes and meshes.

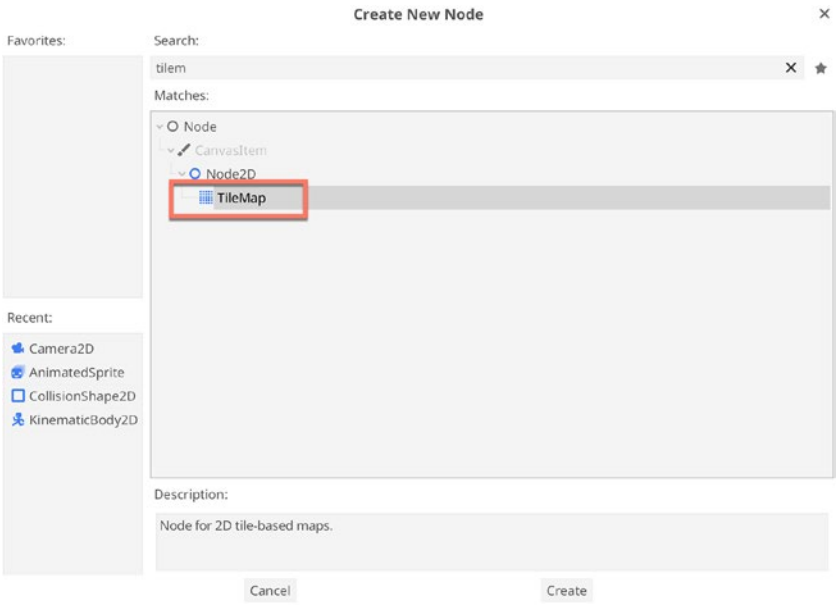

Figure 4-25. *Creating a Tilemap Object*

Newly created *Tilemaps* are generated empty. To start using them, you'll need to create a *Tileset*. A *Tileset* represents the palette or images, or the raw materials, from which you build a *Tilemap*. With the Tilemap object selected in the Scene Tree, click the *Tileset* drop-down from the inspector and select *New Tileset* from the context menu. This creates a new *Tileset Resource*, and this can be edited in the *Tileset Editor Window*, shown by default at the bottom center of the interface or whenever you click the *Tile Set* field in the Inspector. See Figure 4-26.

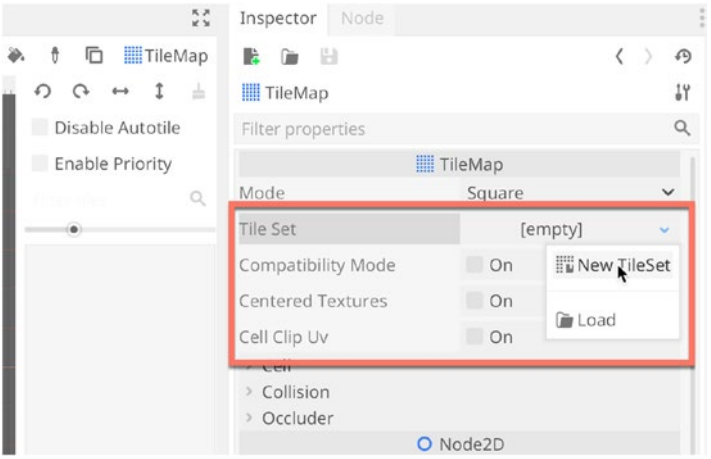

Figure 4-26. *Creating a Tileset Resource for the Tilemap*

The *Tile Set Editor* is used to create new Tile sets. To start, drag and drop a tile sheet texture from the *FileSystem* panel into the texture list of the TileSet Editor. For our level, we'll use the *spritesheet_tiles.png* file included in the book companion files and which are part of *Kenney Assets*. See Figure 4-27. To confirm the drag and drop was successful, the Tileset Window will populate with the sprite sheet image.

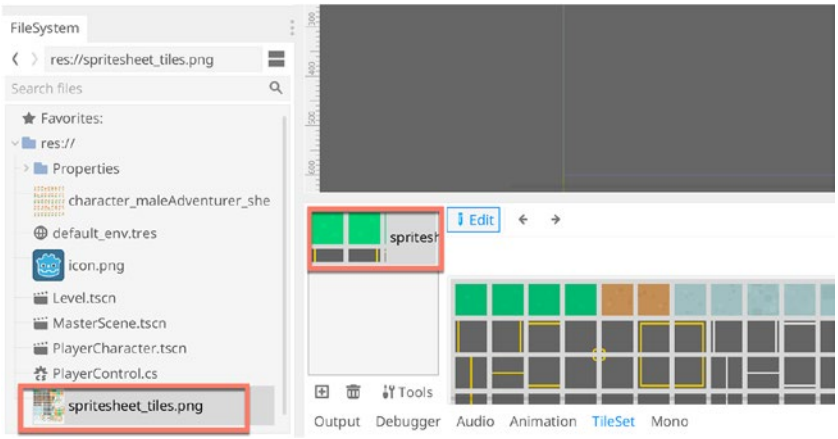

Figure 4-27. *Building a New Tileset*

Let's make our first two tile types, standard tiles for a wooden floor. First, click the *New Single Tile* button from the top menu of the Tile Set Window. Click and drag a region in the texture sheet area and then expand the *Snap* settings from the Inspector. If you don't see the Snap settings, be sure to enable Tile Snapping in the toolbar. This makes selecting and working with Tiles easier. I have chosen a tile width and height of 64 and a pixel separation value of 10. This simply means that each tile in the image is 64 pixels wide and high, and there is a gap of 10 pixels between tiles, both horizontally and vertically. See Figure 4-28.

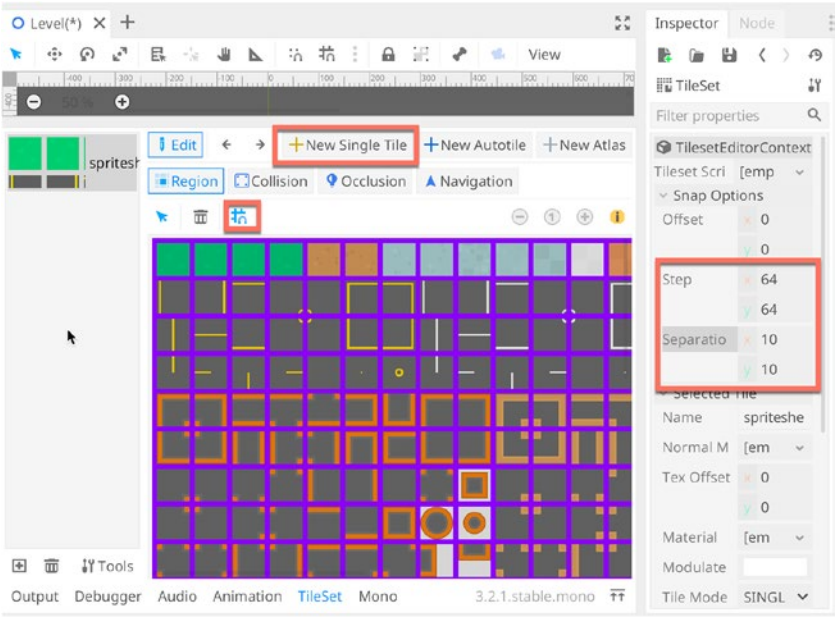

Figure 4-28. *Starting Our First Tile*

Now select the tile image to be used for our first tile, which will be wooden floorboards. To do this, click the floorboard tile image and assign the selected tile a name from the Inspector. This name will display in the Tilemap object, when shown in the tile palette. See Figure 4-29.

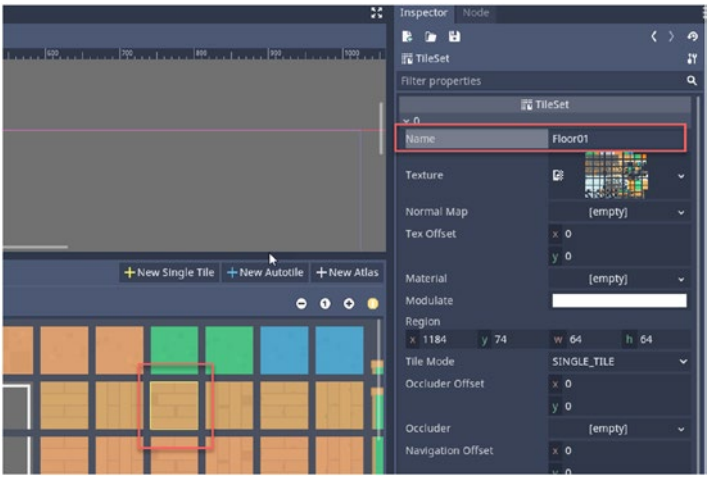

Figure 4-29. *Configuring the First Floor Tile*

You've created your first tile. Excellent. Now jump back to the Godot Tilemap node just by selecting it from the Scene Tree panel. On selecting it, the Tile Palette will be displayed in the Inspector, showing your newly created tile. See Figure 4-30.

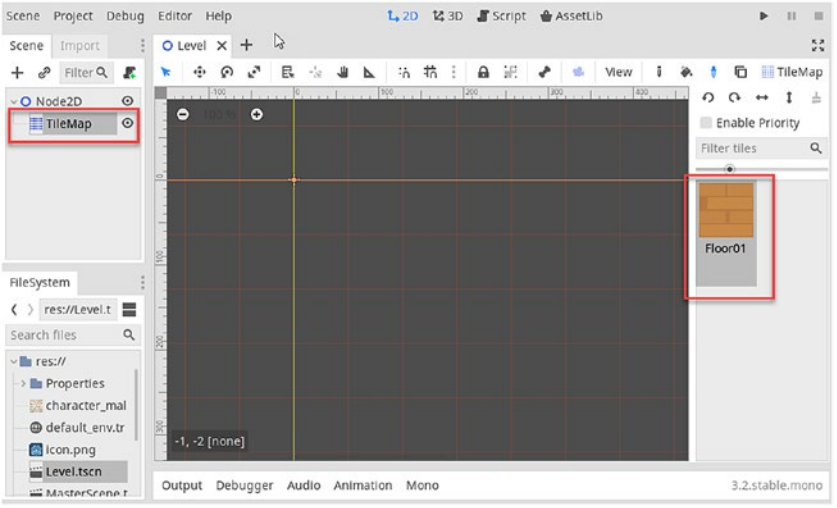

Figure 4-30. *A New Tile Map Node with Our First Tile*

Just by selecting the tile from the Inspector and then by clicking and dragging in the viewport like a brush, you can draw many instances of the tile in the scene to create a world. Right-clicking a tile erases it. You can also use the *rotate left* and *rotate right* controls to rotate your brush, drawing different tile variations. See Figure 4-31.

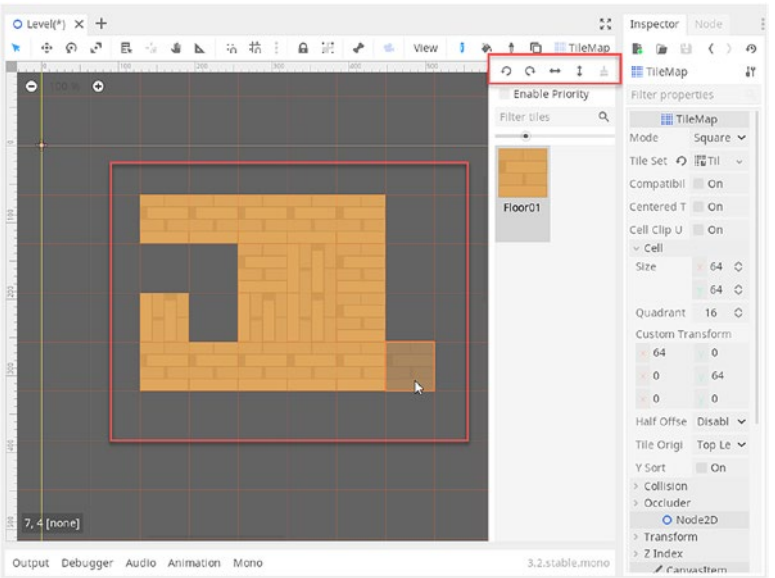

Figure 4-31. *Drawing a Scene from a Tile*

Next, we'll create a second tile (much like the first), which is a slightly different variation of the floorboard tile. This simply adds variety and diversity in the scene. For a third tile, we'll create an *Autotile*, which adds lots of interesting flexibility for wall tiles, tracks, roads, and other kinds of winding tiles that other connect together. An Autotile lets you select a collection of related tiles (e.g., different areas of wall) and will intelligently paint your tile into the level to connect seamlessly with surrounding areas. These kinds of tiles are best understood by trying them out. To get started, click the *Tileset* asset from the Inspector to return to the *TileSet Window*. Then click the *New Autotile* button. See Figure 4-32.

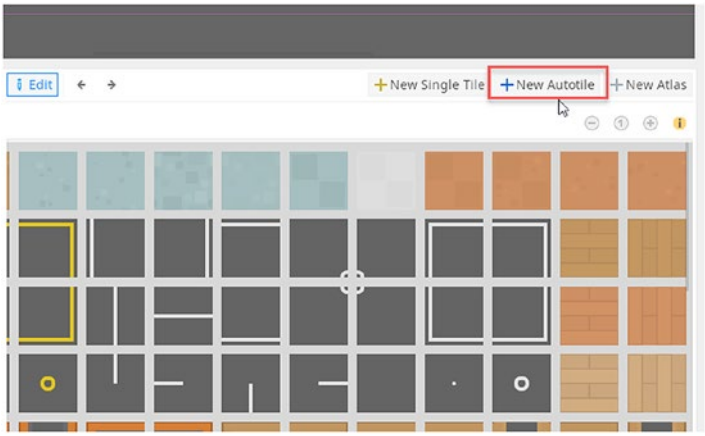

Figure 4-32. *Autotiles Let You Create Integrated Walls, Roads, and Other Elements*

Now click and drag over a tile set region to select all tiles to be included as part of the autotile. Be sure to set your tile snap settings from the Inspector to 64x64 and a separation of 10 pixels, as we did earlier. The purpose of our autotile will be to create wall elements for the environment: corner sections, intersections, and straight sections. So I'll click and drag a complete rectangular region around all the wall elements. See Figure 4-33.

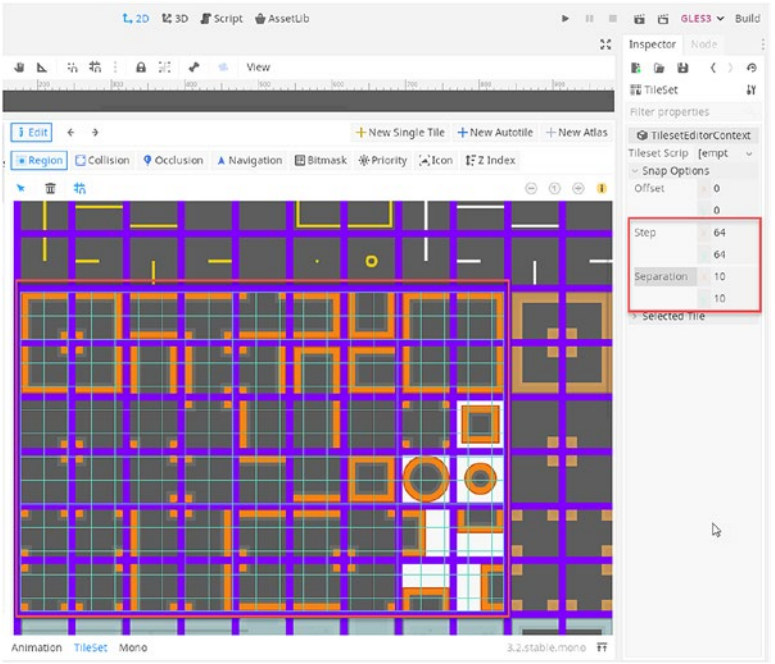

Figure 4-33. *Select an Auto-Tile Region. This Should Contain All Tiles Related to a Specific Architectural Element: Walls, Floors, Roads, Fences, and Others*

Now click the *Bitmask* button from the *Tileset* toolbar. From the Inspector, choose 3x3 (minimal) for the *Autotile Bitmask field*. Then set 64x64 for the *Subtile field, X and Y*. Once these settings are provided, we're ready to complete our auto-tile by drawing out the "walkable" regions, that is, the continuous, inside regions that follow the direction of the wall. See Figure 4-34.

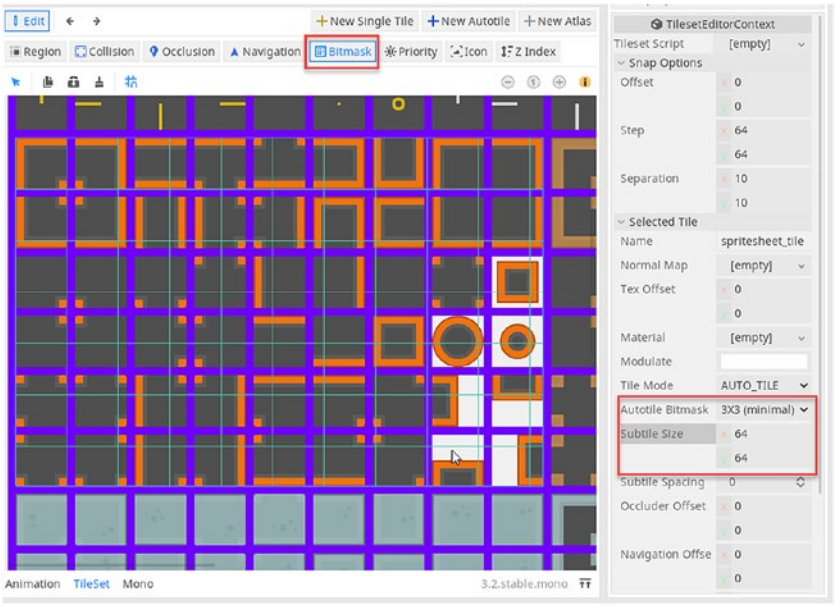

Figure 4-34. *Preparing to Select the Autotile Walkable Region*

Next, click and drag over the "empty" region of the wall tiles. These will be marked in red. Ensure you cover all areas that are "walkable." See Figure 4-35.

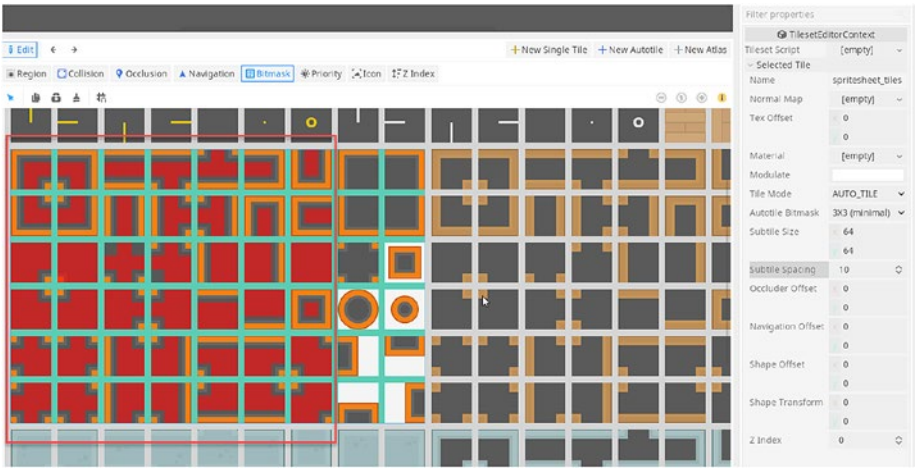

Figure 4-35. *Drawing the Walkable Regions of an Autotile*

Name your *Autotile*, and then switch to the *Icon* tab. From there, click to select the most appropriate icon. In this case, I selected the completely enclosed wall section. See Figure 4-36.

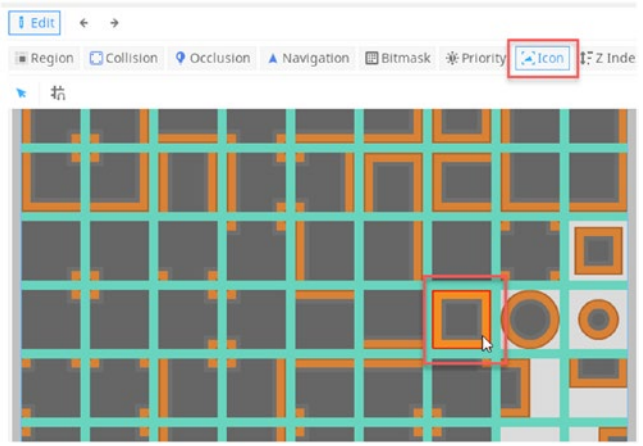

Figure 4-36. *Setting the Auto-Tile Icon*

Now you're ready to start drawing interconnected walls with the auto-tile brush. Jump over to the Tilemap node in the scene, and then select the auto-tile brush, which is represented in the palette by your selected icon. You'll notice that by clicking and dragging the brush around the scene, inside the viewport, the walls will be drawn and connected together correctly, automatically, and seamlessly. See Figure 4-37.

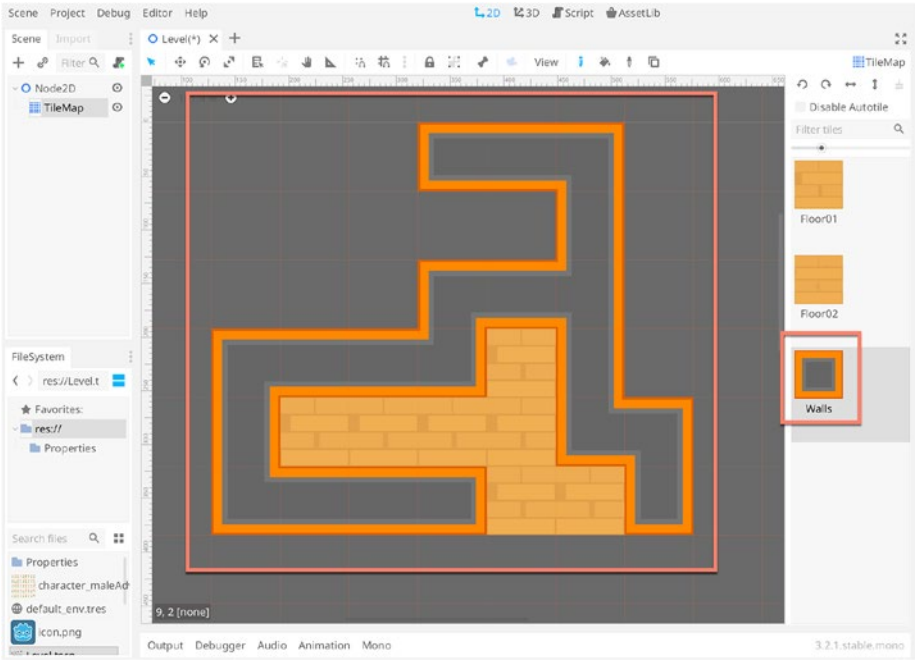

Figure 4-37. *Drawing an Auto-Tile Wall*

Great! Now use your three tiles (two regular tiles and one auto-tile) to build a complete level. A floor layout surrounded by walls. See Figure 4-38.

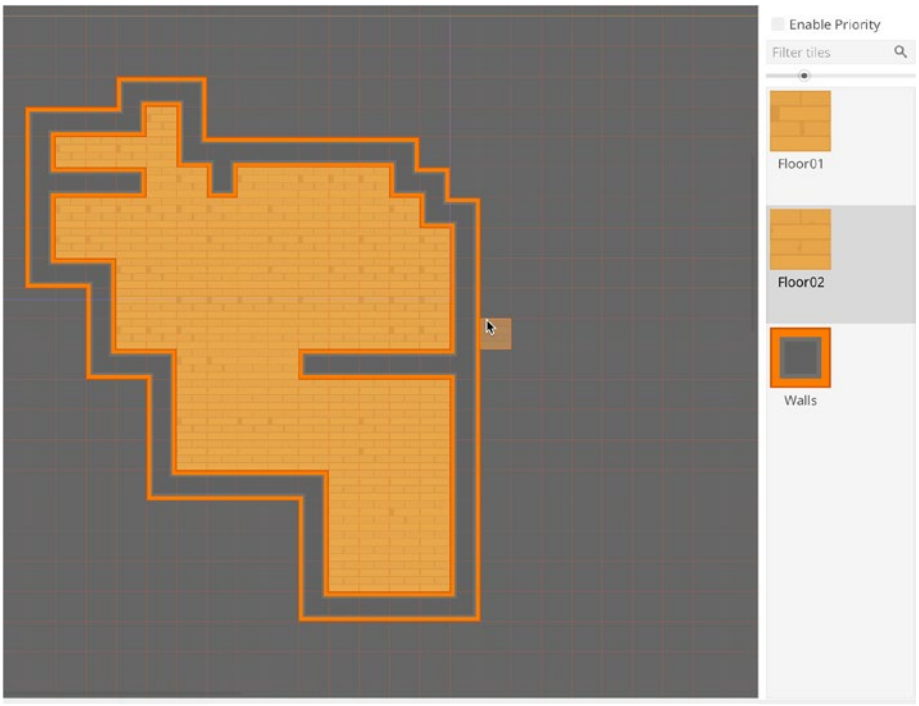

Figure 4-38. *Designing a Complete Level*

World Collisions

We've built both a character and a world. Let's test them. Open the Master Scene file created earlier – or create a new master scene, if you haven't already – and drop both the *player scene* and the *level01* scene together into the master. You may notice that the level overlaps, and hides, the player character. If you don't, check out the Scene Tree Panel. By default, higher objects are rendered behind lower objects.

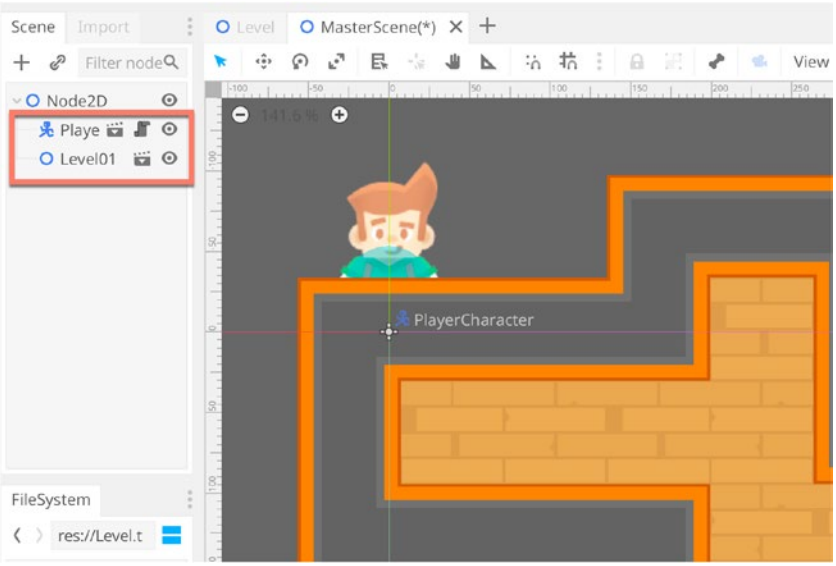

Figure 4-39. *Lower-Order Nodes in the Hierarchy Render on Top of Higher-Order Nodes*

We could solve the render order problem by rearranging the nodes, moving the player below the level node. However, this order really shouldn't matter when it comes to rendering. So, we can tweak it easily instead by jumping to the level scene and selecting the Tilemap node. We should set its *Z Index* to *–1000*, leaving *Z as Relative* check box enabled. When enabled, the Z Order of the *Tilemap* will always be set relative to its parent. See Figure 3-40. And now the level appears behind the player.

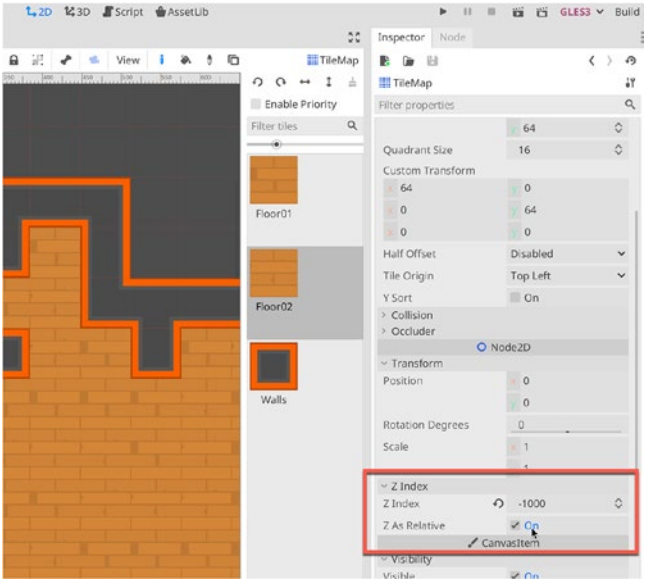

Figure 4-40. *Setting a Node's Render Order (Z Index)*

The main problem however is that our game, when played, doesn't support any collisions even though our player controller uses the *MoveAndSlide* function inside the *_PhysicsProcess* event. By pressing *Play* or *Play Scene*, the player will happily walk through walls and any solid objects. See Figure 4-41.

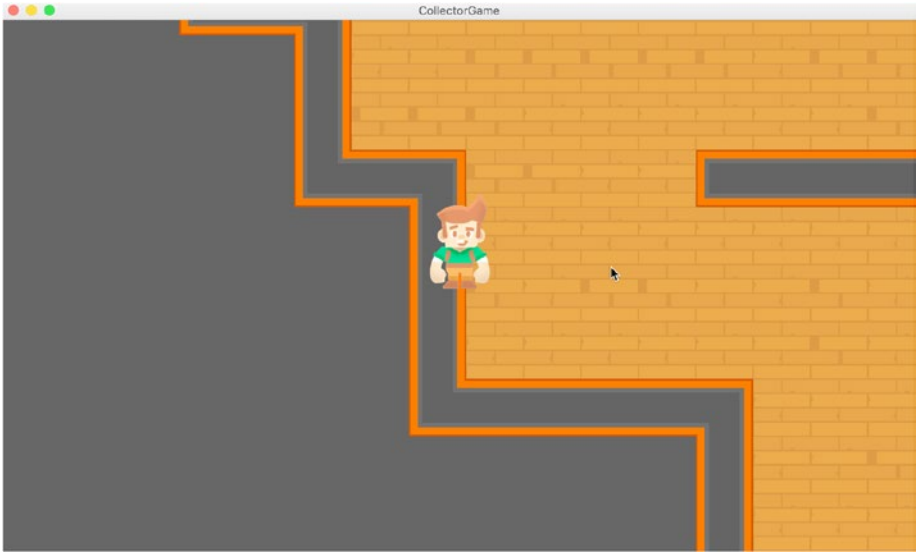

Figure 4-41. *The Player Walks Through Walls!*

To fix this, we'll need to create Collision information for the level. This information defines the borders and extents of solid, impassable objects. Open up the Level Scene file and create a new *StaticBody2D* node. See Figure 4-42. Static Bodies represent solid objects that never move during gameplay. Typically, this includes walls, ceilings, floors, trees, pillars, and others.

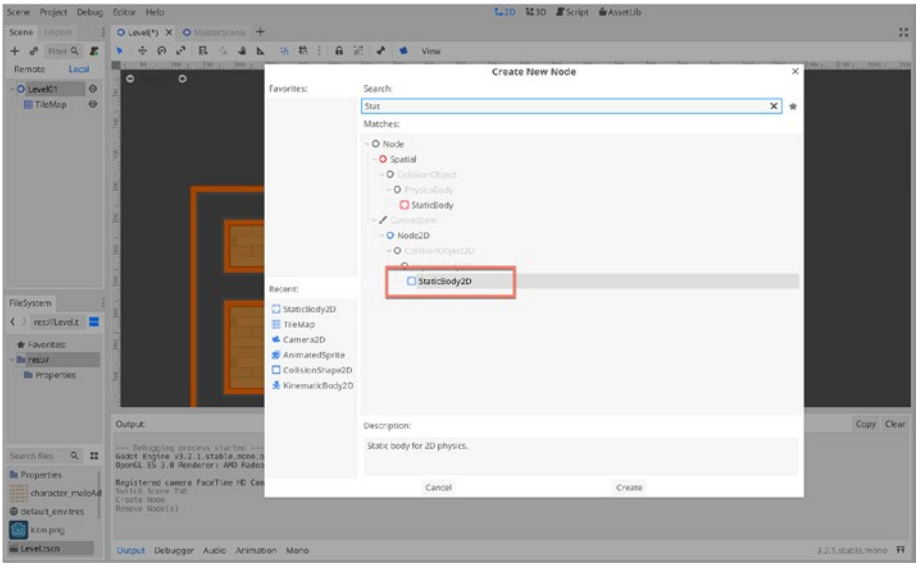

Figure 4-42. *Create a New Static Body for Non-movable Objects*

As with the *KinematicBody2D* created earlier for the player character, the *StaticBody2D* has no extension or volume by default. We need to create its size and shape using a *CollisionPolygon2D*. To do this, select the newly created *StaticBody2D* node, and create a child *CollisionPolygon2D* node. Once added, click the *Create Points* icon in the toolbar, and then click inside the viewport to start placing points around the level walls. See Figure 4-43.

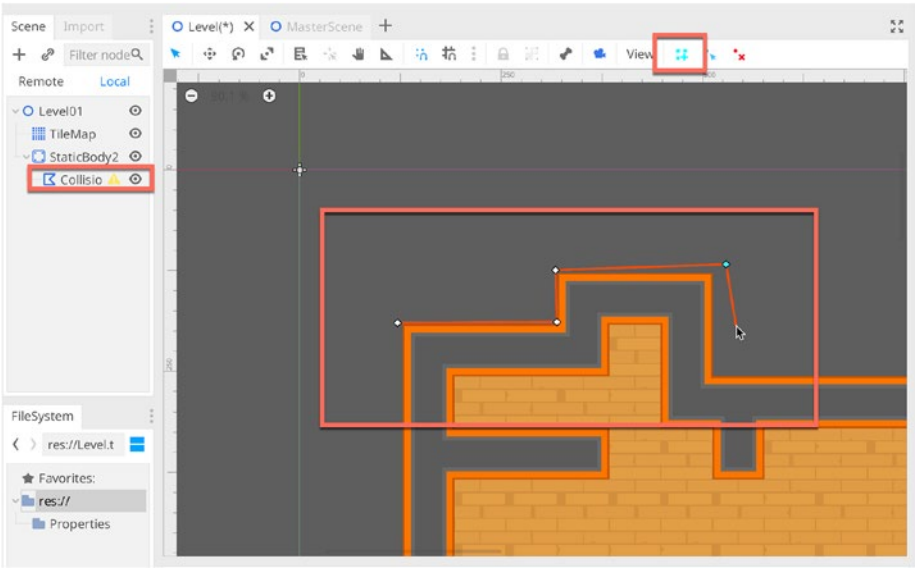

Figure 4-43. *Build a Collision Polygon for the Scene Static Body*

By default, point placement is free and loose, and you'll likely want to snap points to the world grid to ensure they align with the walls easily. To do this, click the Toggle Grid Snap icon from the top toolbar. A Grid is then placed over the complete map and drawn in the viewport. It will not, of course, be visible to the player – it's for our reference only. See Figure 4-44.

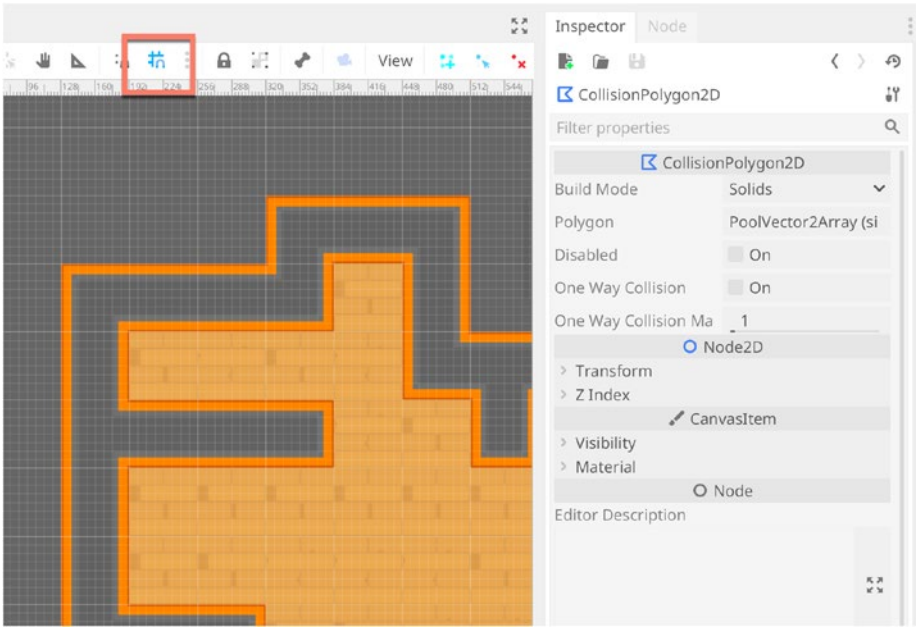

Figure 4-44. *Enabling Grid Snap for Precise Point Placement*

The created grid will typically be tightly packed, spaced to 8 pixels per grid line. We can change this to suit our tiles better. To do this, click the *Snap Options* button (the three dots beside the Snap Toggle), and choose *Configure Snap* from the menu. See Figure 4-45.

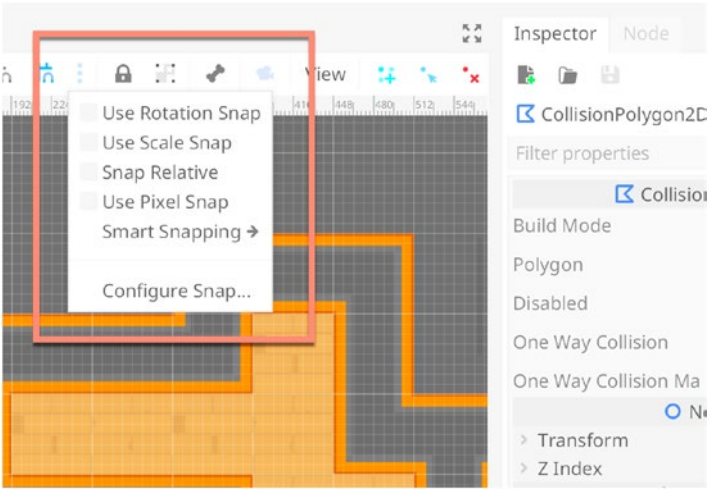

Figure 4-45. *Accessing the Grid Snap Options*

For our TileSet, let's use a value of 32x32 pixels for X and Y. These will realign the gridlines and match our tile sizes while also giving us appropriate steps within the grid. Excellent. See Figure 4-46.

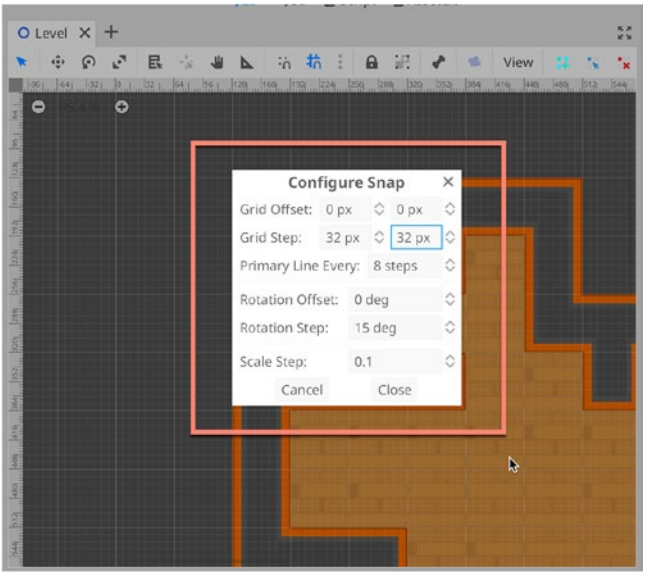

Figure 4-46. *Setting Grid Increments*

Now you can easily click inside the level to place points for the *CollisionPolygon2D*, building a Collision Shape that represents the scene walls. Be sure to add a point at every corner. Take a look at Figure 4-47, which shows a half-completed wall arrangement.

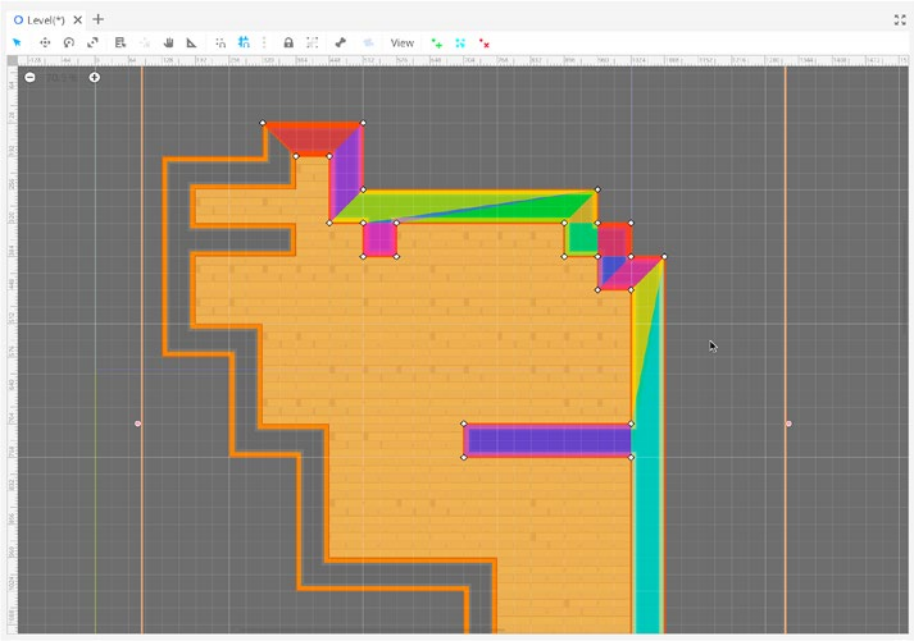

Figure 4-47. *Building the Wall Collision Polygon*

Great. You've now constructed a full collision polygon that surrounds the walls. By opening the Master Scene and running the game, the player character will not pass through the walls and will remain within the scene boundaries. This is good progress, but there's more to do still. See Figure 4-48. Now, let's see how to progress in the next section!

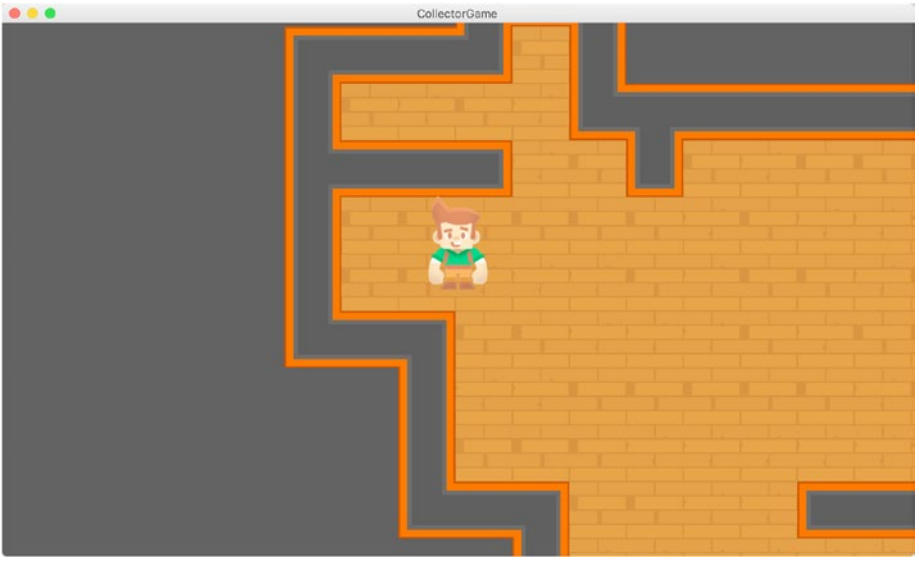

Figure 4-48. *Supporting Player Collisions…*

2D Lighting

In this section, we'll add drama to the scene with 2D lighting. Specifically, the room will begin dark, and the player will carry a torch, illuminating any areas he moves to. The light will also support full shadow casting, allowing different 2D objects to cast shadows and obscure light. To get started, let's open up our environment level scene. To make it dark, add a *CanvasModulate* node. And from the Inspector, set its Color property to a very dark gray, close to black. Suddenly, the entire level darkens. The CanvasModulate node control multiplies the color values for all Canvas-based objects, including 2D objects and user interface elements. See Figure 4-49.

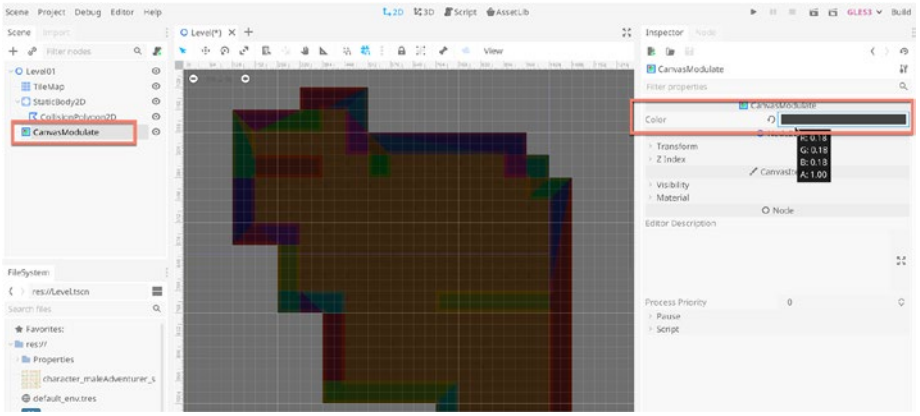

Figure 4-49. *Supporting Player Collisions…*

To create Shadows and to ensure the world responds correctly to any lights added, let's create a *LightOccluder2D* node. This works like the *CollisionPolygon2D* node created in the previous section to define the collision extents of the walls. By comparison, the Occluder node defines which objects will "block" light. Since light won't travel through walls, the Occluder polygon should be the same as the wall collision. Take a look at Figure 4-50.

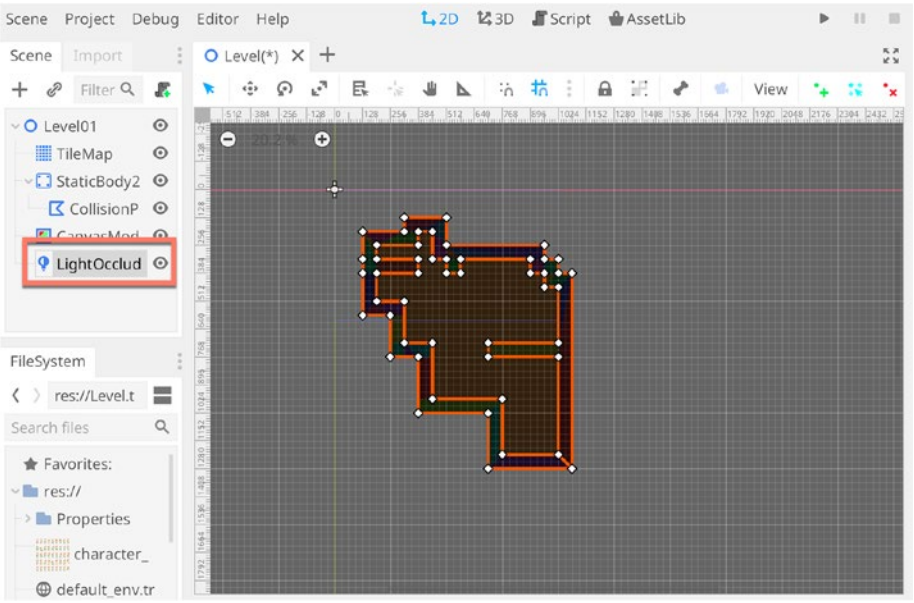

Figure 4-50. *Building Light Occluder Polygons*

Now let's switch over to the player scene. Select the root node and add a new *Light2D* child. This represents a light object. It requires a 2D texture to define the shape, scale, and brightness. Let's use the light texture provided by Godot, in its Light2D tutorial, here: `https://docs.godotengine.org/en/3.2/tutorials/2d/2d_lights_and_shadows.html`. The image is Light.png, which is also included in the course companion files for your convenience. Please do check out this tutorial, and others, on the excellent Godot website: `https://docs.godotengine.org/en/3.2/tutorials/2d/`. In this section, we'll import the light.png texture into the Godot *FileSystem* panel. And then assign it to the Texture slot of the Light2D. See Figure 4-51.

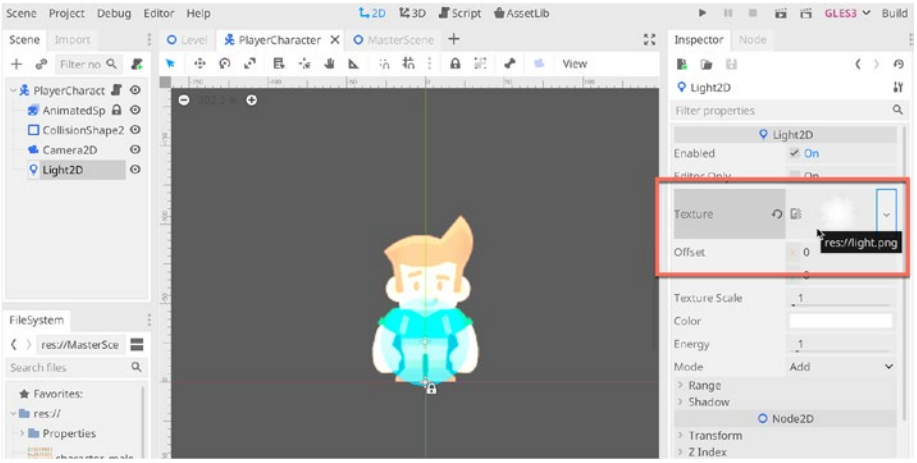

Figure 4-51. *Configuring a Light2D Texture*

Next, let's configure Light Shadow casting to interact with the world. To do this, select the *Light2D* node from the Scene Tree and then expand the *Shadows* tab from the Inspector. From there, click the *Enabled* check box. And now test the *MasterScene*. See Figure 4-52. Great, the Light2D now interacts with, and casts shadows from, the main environment walls.

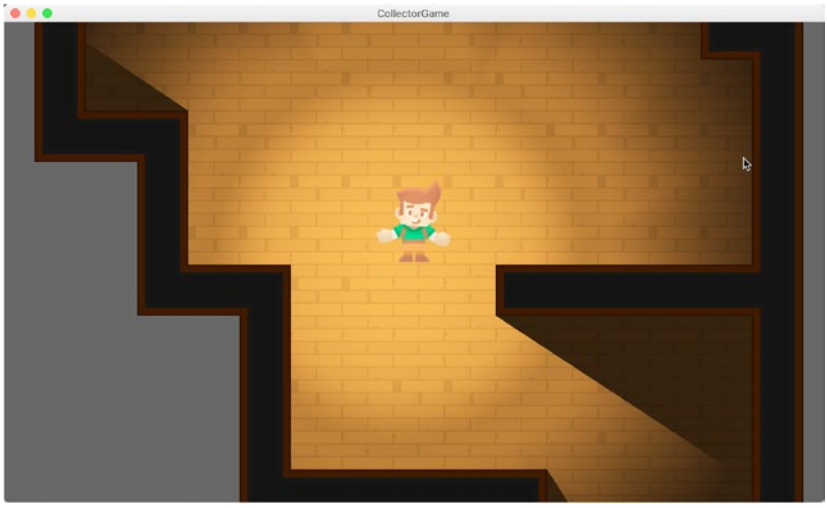

Figure 4-52. *Operational Shadows! Great Work…*

Pickups

In our collection game, the player needs to collect something! The idea is that the player must collect all pickups before the time expires. In our case, the player will collect gamepad sprites, and these can be downloaded from the excellent texture resource, Kenney Assets, available for free here, as part of the Generic Items pack, `www.kenney.nl/assets/generic-items`, specifically the PNG image file *genericItem_color_082.png*. See Figure 4-53.

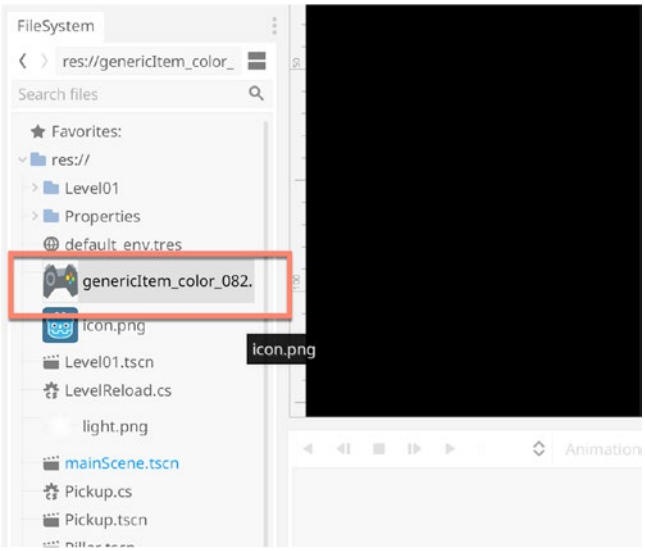

Figure 4-53. *Importing a Gamepad Texture for the Pickup*

Next, create a new 2D Scene. This will be used for the Pickup object. Then create an *Area2D* node and make this the scene root. The *Area2D* node behaves like a Unity Trigger collider. It defines an area or a volume within the scene, inside which events can be detected. But, it doesn't represent a physical, solid object like a *StaticBody* or a *Kinematic* body. Areas are useful for lava pits, water, poisonous atmospheres, and cut-scene triggers – whenever you must detect the entry or exit of objects in areas. See Figure 4-54 for the original setup.

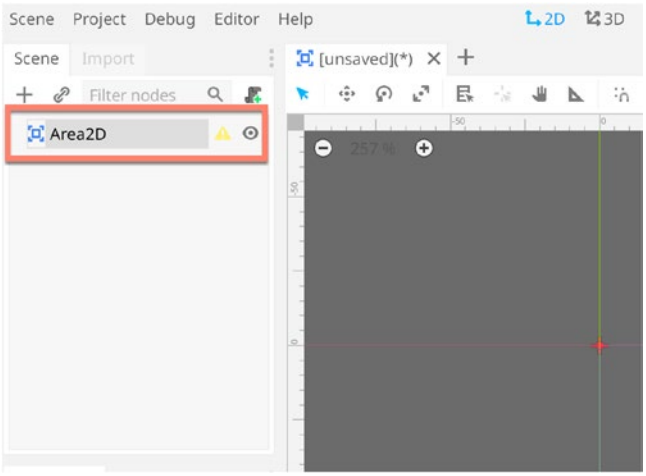

Figure 4-54. *Make an Area2D Node the Scene Root*

Let's proceed by adding both a Sprite Node and a *CollisionShape* node. The former defines the object appearance and the latter its volume. We've seen this process before when creating both the player and the world. Consider Figure 4-55 for the final configuration.

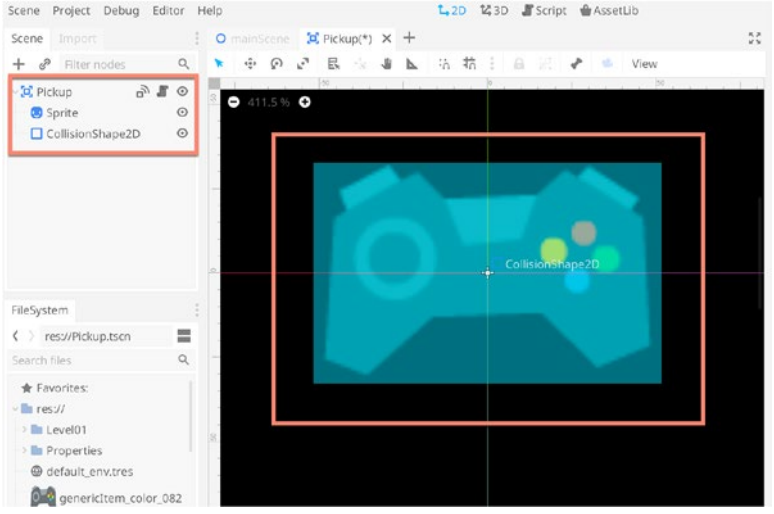

Figure 4-55. *Configuring a Pickup*

Great. Finally, let's add the Pickup to a "Pickup" group and use the Nodes tab from the Inspector (see Chapter 2 on how to use Groups), and then we can add some code to the pickup. The code will make the object disappear when it touches the player and will check to see if there are any remaining collectible objects. If not, the level is completed. See Listing 4-2. This code should be attached to the topmost Area2D node of the Pickup scene, and it assumes the player character belongs to the *Player* Group.

Listing 4-2. Responding to Pickup Collection

```
using Godot;
using System;

public class Pickup : Area2D
{
    public void _on_Pickup_body_entered(Node N)
    {
        if(!N.IsInGroup("Player"))return;

        //Check pickups remaining
        if(GetTree().GetNodesInGroup("Pickup").Count <= 1)
        {
            //You completed the level
            GD.Print("Level Completed");
        }

        CallDeferred("RemoveObject");
    }

    public void RemoveObject()
    {
        GetParent().RemoveChild(this);
    }
}
```

Now jump back to the Level Scene and add multiple instances of the Pickup object around the level for the player to collect. See Figure 4-56.

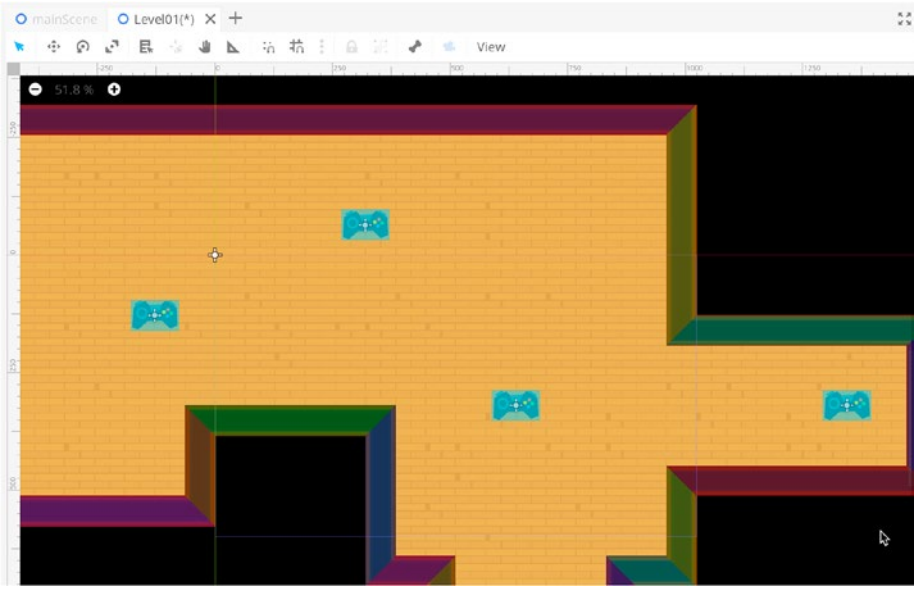

Figure 4-56. *Placing Pickups*

Now you can play the Master Scene and watch the player collect all items. This is excellent. We're nearly there. Only one more feature left to create: the timer! See Figure 4-57 for the completed level with collectibles.

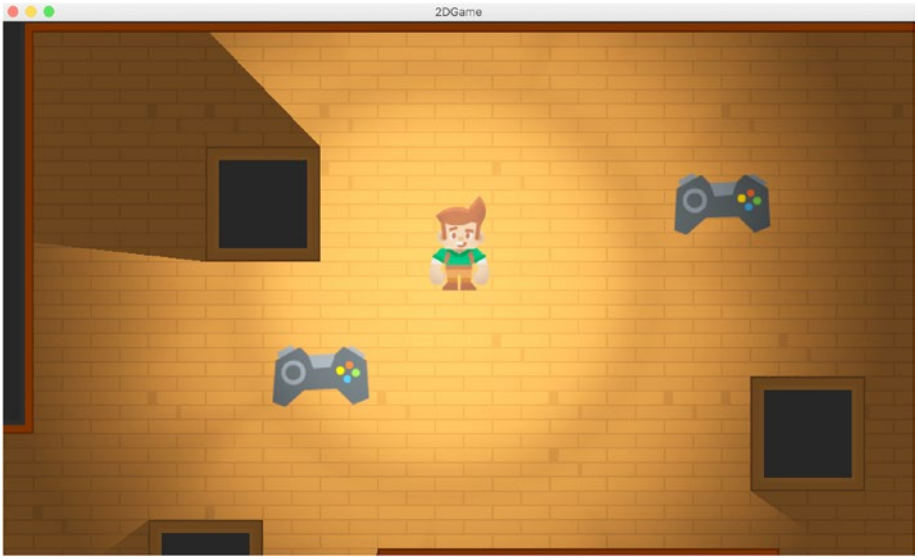

Figure 4-57. *Catching Those Pickups!*

Timers and Countdowns

So, the final step is adding a timer countdown. When the timer expires, the level will restart. It's that simple, and Godot makes it incredibly easy to add timers. To start, open the Level Scene, since the countdown is part of the level and not the player. Although the countdown *applies to* the player, in terms of game mechanics, its properties are specific to the level. So, let's add a Timer node. See Figure 4-58.

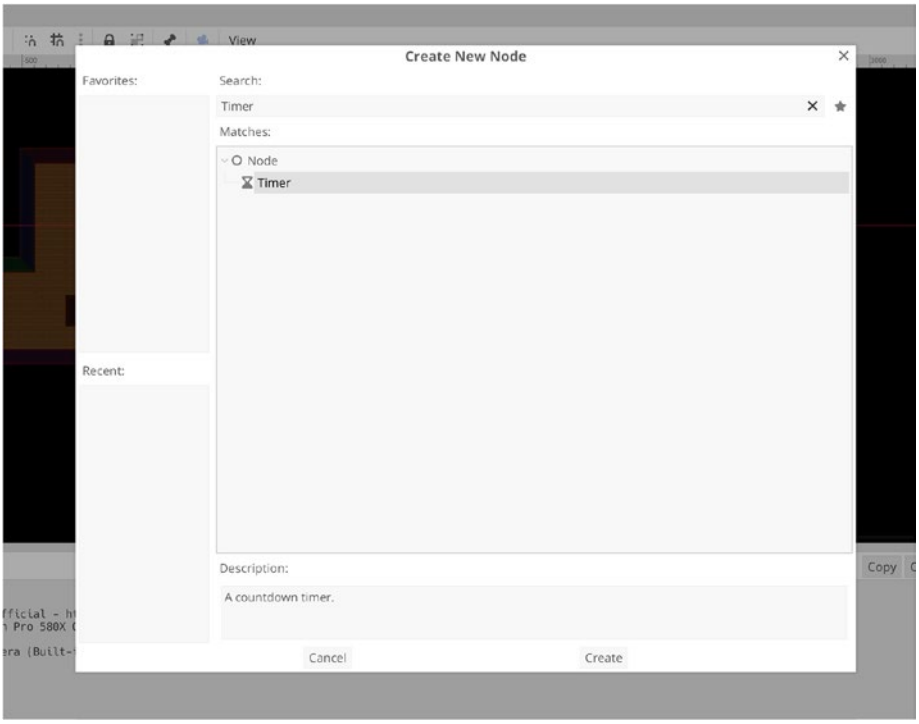

Figure 4-58. *Adding a Timer Node…*

Set the Timer *Process Mode* to *Physics*, to update along with the physics cycle, and specify a Wait Time of 20 seconds – we can easily change this – which represents the total time available for the player to collect the pickups. Enable *One Shot* and *Autostart* to begin the timer as the level begins. See Figure 4-59.

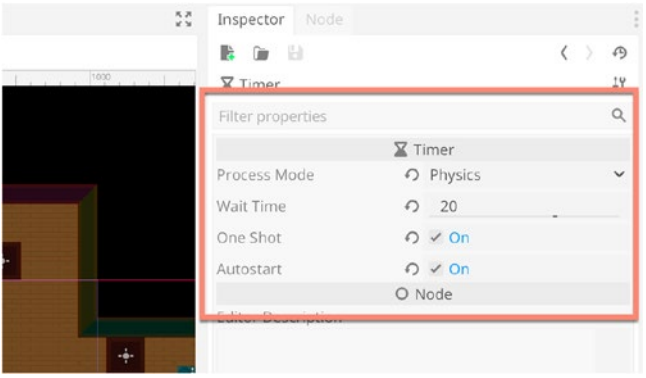

Figure 4-59. *Configuring a Timer*

Now we'll create a new C# script file to handle the level restart behavior for when the timer actually expires. It's pretty simple code. This script should be attached to the Timer node. Take a look at Listing 4-3.

Listing 4-3. Level Reloading for Expired Timers

```
using Godot;
using System;

public class LevelReload : Timer
{
    public void _on_Timer_timeout()
    {
        GetTree().ReloadCurrentScene();
    }
}
```

We should connect this code to the Timer expired signal and allow the code to execute when the timer expires. To do this, select the *Timer* node and switch to the *Node* tab on the Inspector. From there, double-click the Timeout signal and connect it to our *_on_Time_timeout* function, as we've seen before. In the end, your signal panel will look as shown in Figure 4-60.

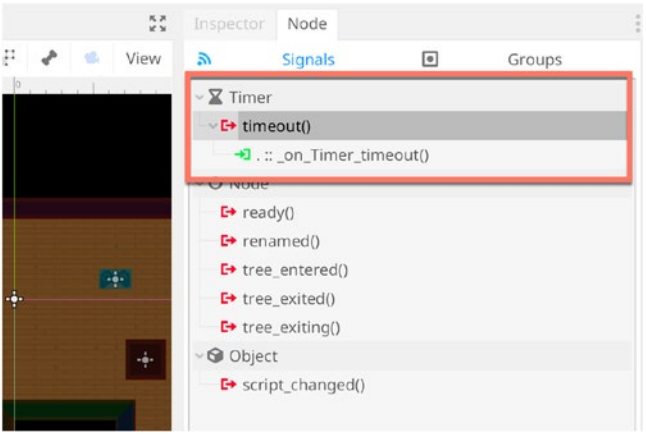

Figure 4-60. *Responding to Timeout Behaviors*

And that's it! You've now completed a simple, but functional collection game using the Godot engine. Excellent. Go ahead; take it for a test run and behold your work! See Figure 4-61.

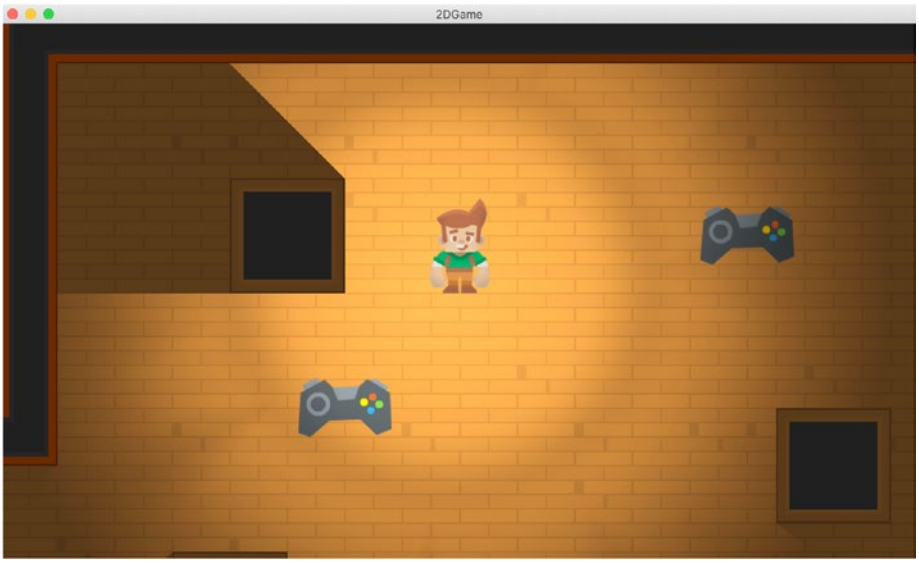

Figure 4-61. *The Completed Game!*

Summary

Great work. In this chapter, we created a complete, comprehensive, and fun 2D game with a variety of behaviors, showcasing many Godot 2D features. Godot has an amazing and easy-to-use 2D feature set, which is easy to pick up, use, and reuse. Godot's clever hierarchical scene framework makes it easy to create gameplay elements, like levels and player characters, in isolation and then bring them together seamlessly into a single world.

CHAPTER 5

3D Lighting and Materials

This chapter focuses on Lighting and Materials in Godot, specifically for 3D. Here, we'll focus on Godot's standard material and light types and then explore its two major lighting systems: *Global Illumination* and *Light Baking*. By understanding and using these techniques, you'll effectively transition from Unity to Godot's lighting system, and you'll be able to configure Godot scenes to achieve high-quality visual results in 3D. It must first be admitted, however, that Godot's lighting system is rudimentary compared to the newer *HDRP* (*High Definition Render Pipeline*) in Unity. In the Unity HDRP, we see an extensive, real-time PBR system for easily creating high-quality scenes, by tweaking just a few settings and materials. Godot, by contrast, is more limited in its lighting features out of the box. But still, that being said, Godot is capable of delivering high-quality lighting that makes a 3D scene compelling. Although a PBR workflow is supported and features exist to achieve a photo-realistic finish, the results are often performance intensive for many systems. This means that Godot may not be the right choice, right now, for a "photo-realistic" game. But that still leaves us many options and styles. We'll explore lighting in this context throughout the chapter.

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5_5

Lighting Fundamentals

Let's explore the Godot Light types together. We'll do this using a simple scene to see the lighting results in isolation, in a stand-alone scene, specifically a cube on a plane. This is created easily. Just make a new scene, and then add two *MeshInstance* nodes: one created as a *Cube Mesh* and the other as a *Plane Mesh*. See Figure 5-1.

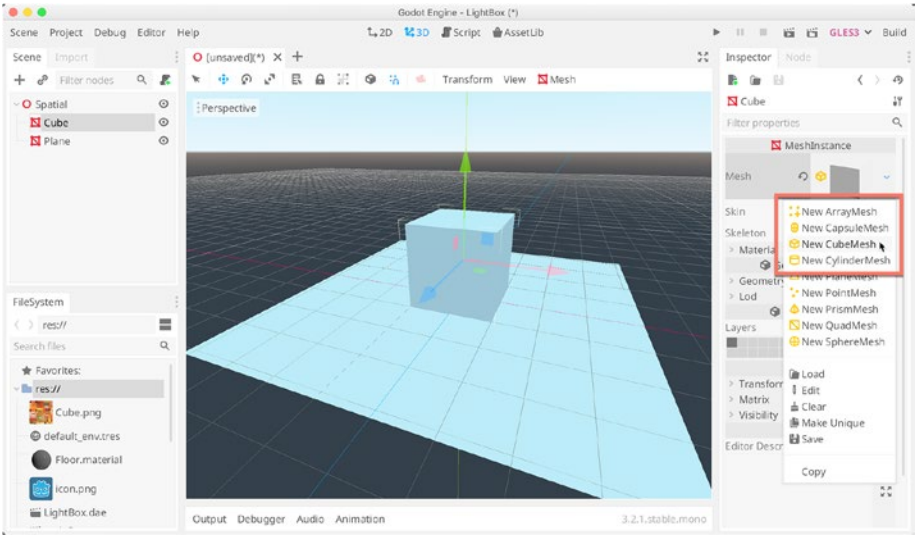

Figure 5-1. *Creating a Basic Cube/Plane Scene Setup for Exploring Lights*

You'll notice that every newly created Godot 3D scene features lighting by default. Even though there's no light listed in the scene tree, we still see a sky in the viewport, and both the cube and plane meshes are illuminated. In a scene truly absent of light, we'd see only black or darkness. But in our scene – as with all new 3D scenes – we have lighting already. This is because every new 3D scene is automatically associated with a default *WorldEnvironment*, defining the lighting properties of the "world" – the natural, environment light from the sun or the moon, or surrounding

areas –including the sky and fog. For our purpose, let's completely switch off the environment lighting. To do this, we'll override the default world environment. There are multiple ways to do this. We'll add a new node, namely, a *WorldEnvironment Node*. See Figure 5-2.

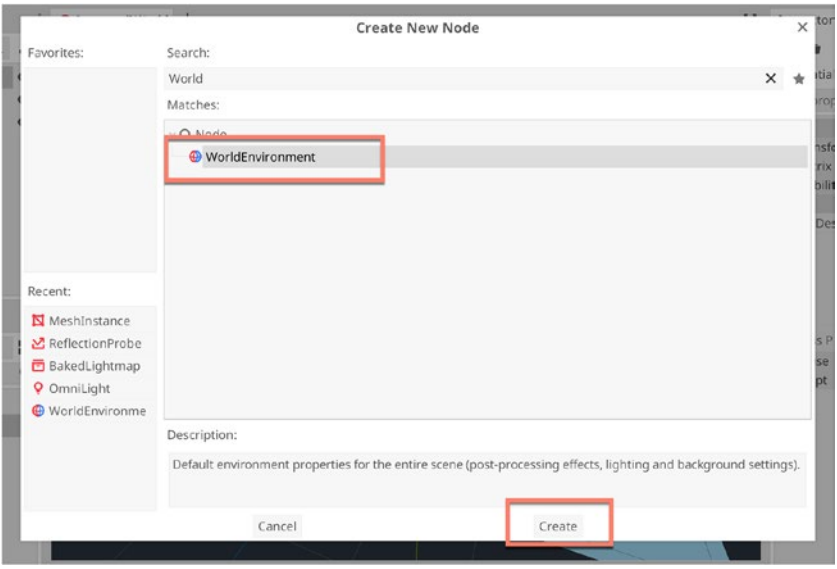

Figure 5-2. *A World Environment Node Overrides Ambient Light*

After creating the *WorldEnvironment* node, select *New Environment* for the *Environment* field in the Inspector. This creates a new environment and the world turns dark, overriding the original settings. See Figure 5-3.

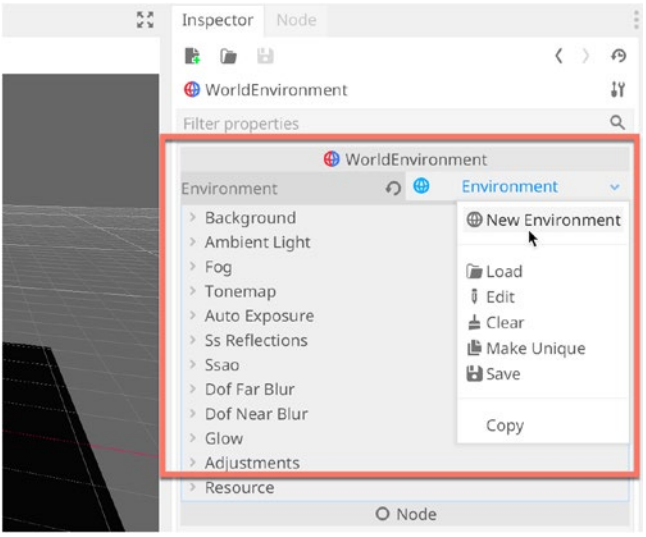

Figure 5-3. *Configuring the World Environment Node…*

Now expand the *Background* and *Ambient Light* sections, and then turn their *Energy* field to 0. This removes all ambient light from the scene. See Figure 5-4.

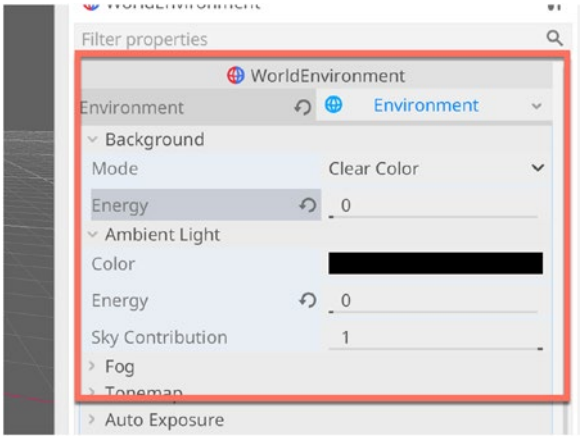

Figure 5-4. *Removing All Ambient Light*

Exploring Light Types

We've now succeeded in removing any and all ambient light from the scene by overriding the *WorldEnvironment* lighting data using our own newly created *WorldEnvironment Node*. This turns the scene – and indeed, any scene without lights – completely black. Doing this is useful for exploring lights, because we can create lights, edit their properties, and see them in isolation. Let's start by creating a ***Directional Light***. To do this, add a *Directional Light* node. See Figure 5-5.

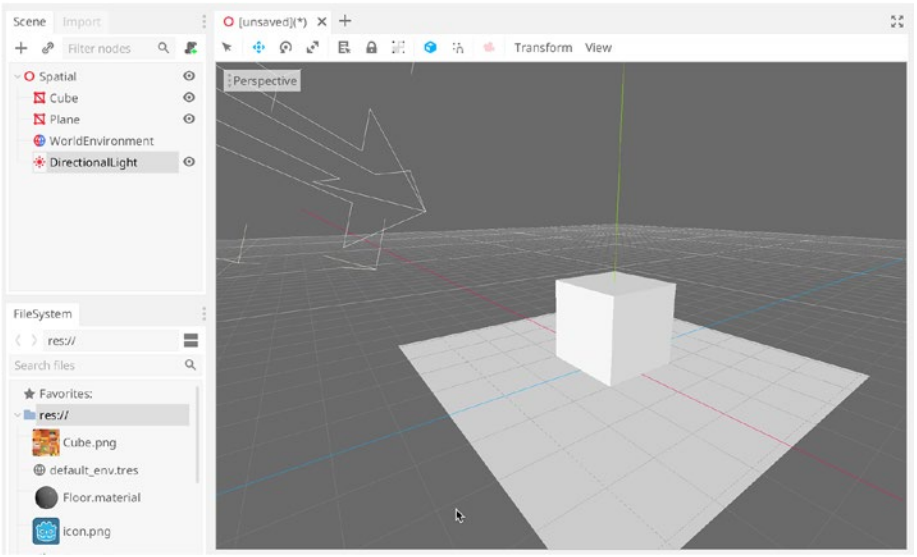

Figure 5-5. *Creating a Directional Light*

Directional Lights are truly "positionless" insofar as their location within the scene makes no difference to their lighting impact. It's only their orientation – their direction – that determines what they illuminate and how. Directional Lights are useful for simulating the sun, the moon, and other huge natural light sources. They are also used to simulate bounced light and ambient light. You can enable Light Shadows – for all standard

light types – by expanding the *Shadow* tab in the inspector and setting *Enabled* to *On*. By default, Shadows will appear completely black. You can change their intensity by changing the Shadow Color, for example, from black to gray. See Figure 5-6 for Directional Light Shadows.

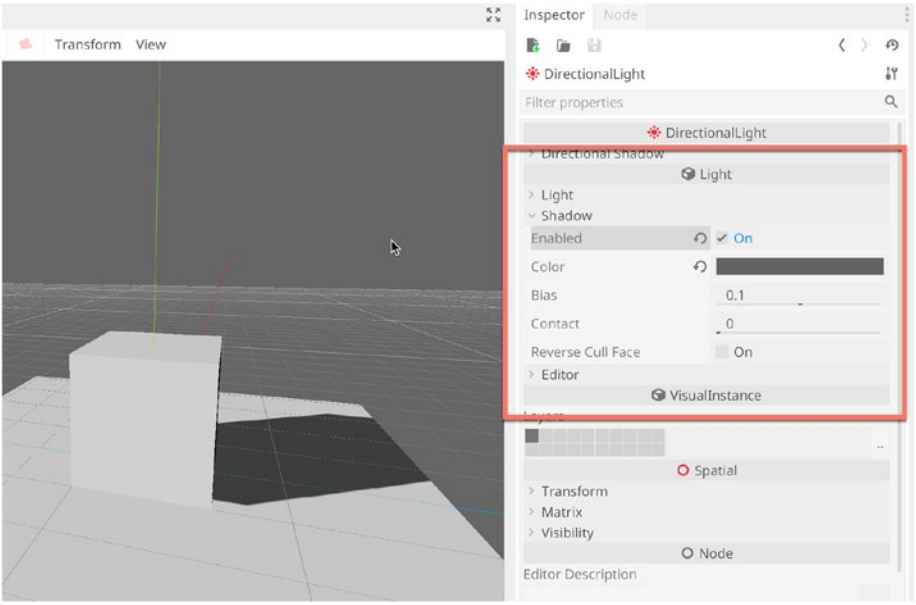

Figure 5-6. *Controlling Directional Light Shadows*

Next, let's delete the *Directional Light* and try a ***Spot Light***. The Spot Light is a computationally expensive Light Source, which has a position, a range, and a cone of influence. It's useful for creating car headlights, ceiling lights, and other artificial Light Sources. The Light tab can be used to control a light's color and brightness. See Figure 5-7.

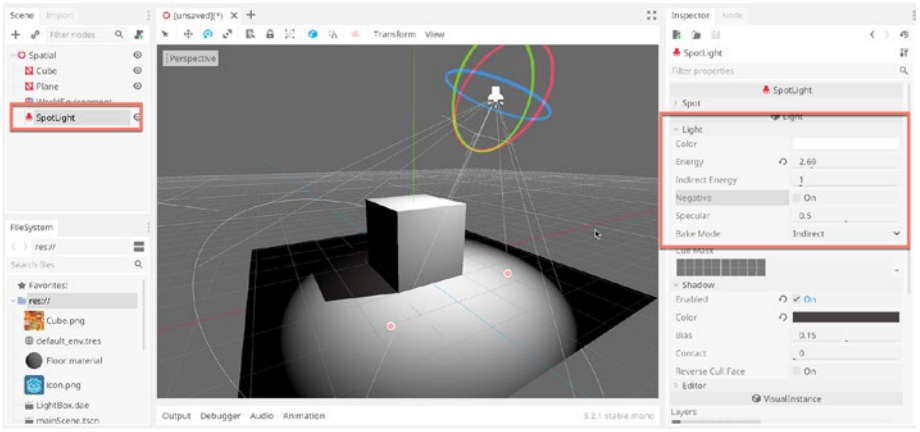

Figure 5-7. *Adding Spot Lights*

The third and final main light type is the ***Omni Light***, sometimes referred to as a Point Light. This light is defined by a location and a range. From its position, it casts light in all direction for a specified range. This light simulates lamps, bulbs, and other artificial sources. See Figure 5-8.

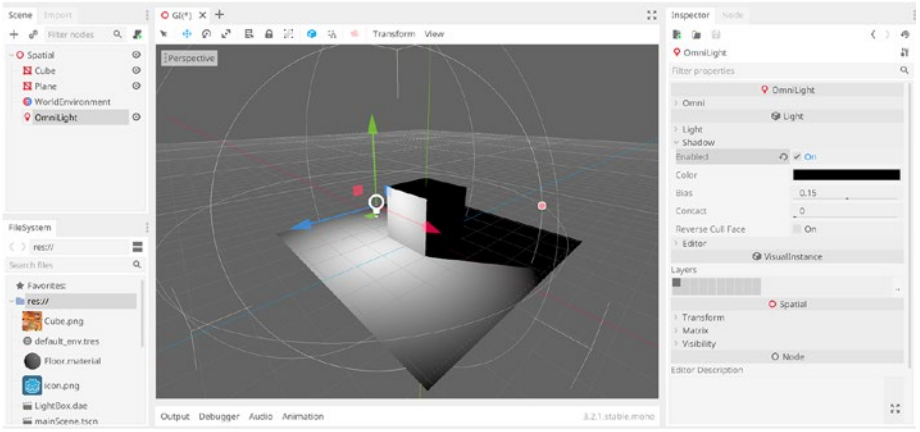

Figure 5-8. *Creating an Omni Light*

Together the Directional, Spot, and Omni lights are the three main light types for multi-purpose lighting in a Godot seen. As we'll see shortly, there are more types that are not immediately obvious and which can be used to great effect.

Materials

Lights are an important aspect of making scenes look great, but so are materials. A material defines how the surface of an object *will* appear, when lighting and environmental conditions are all considered. Materials let you apply textures, color, and other effects to your objects. Normally, your object needs correct UVs – added inside your 3D modeling software – for textures to appear correctly on its surface. In this section, we'll create a simple material and apply it to a cube in our sample scene. To do this, right-click the *Res//* path inside the *FileSystem* Window, and then choose *New Resource* from the context menu. See Figure 5-9.

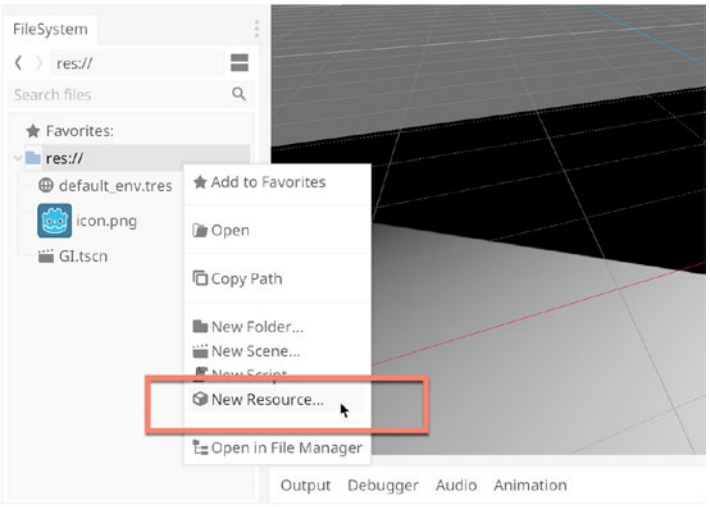

Figure 5-9. *Making a New Resource*

Afterward, a *New Resource* Window appears. From here, search and select a *SpatialMaterial*. These are materials that are created and added to 3D objects. It is Godot's equivalent to the *Unity Standard Shader*, or to the *HDRP* ➤ *Lit* material, or the *URP* ➤ *Lit* material. It aims to provide a diversity of shader features needed to illuminate most 3D objects. See Figure 5-10.

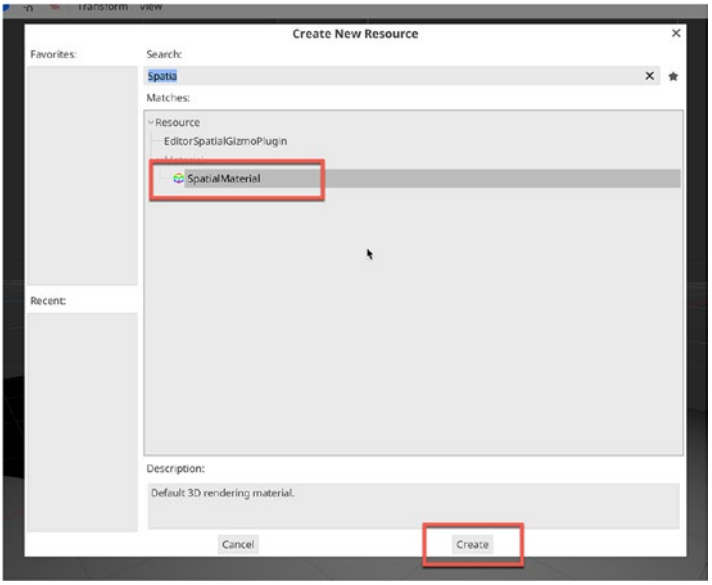

Figure 5-10. *Creating a New Spatial Material*

Name your material *matCube*; and it will appear as a Resource in the FileSystem panel. You can select and view the material properties by double-clicking it. When selected, the material properties display for editing inside the Inspector. To change the main material color, expand the Albedo section, and click the Color swatch. This lets you select a color for the material. For our example, I'll make the material red, and it will soon be applied to the cube. See Figure 5-11.

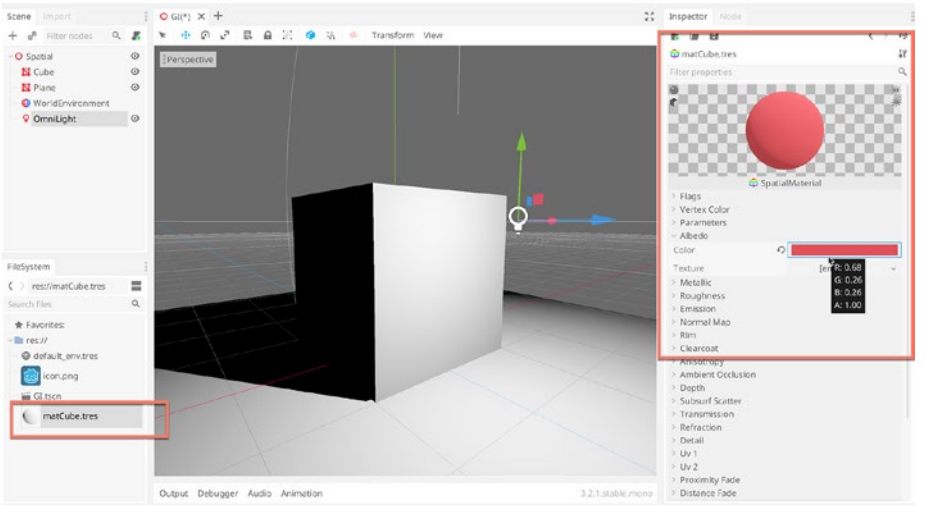

Figure 5-11. *Building a Red Material*

You can use the *Texture* slot to select an image texture for applying to the mesh. This option only works as intended if the 3D mesh has appropriate UVs.

To assign the material to the cube, select the cube in the scene, and then expand the *Material* section from the Inspector. Click the *Empty* drop-down, and then choose *Load* from the context menu. See Figure 5-12.

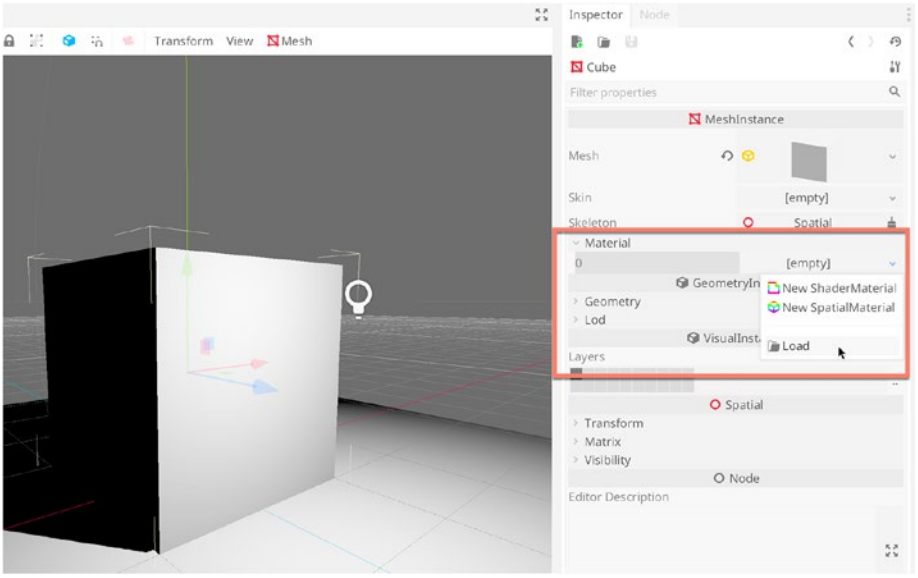

Figure 5-12. *Assigning a Spatial Material*

From the *Resource Selection Window*, choose the newly created *Red Material*. Once assigned, the cube will turn red to reflect the material assignment. Now double-click the material from the File System panel to view its properties in the Inspector again. Expand the Parameters section and check the *Cull Mode* field. This should be set to *Back*. Back means that *Backface Culling* is enabled. This is a performance optimization that saves the Renderer from having to render the "back faces" of a mesh, that is, the reverse side of a surface that is normally facing away from the camera. For most meshes in a 3D scene, you'll want this enabled. See Figure 5-13.

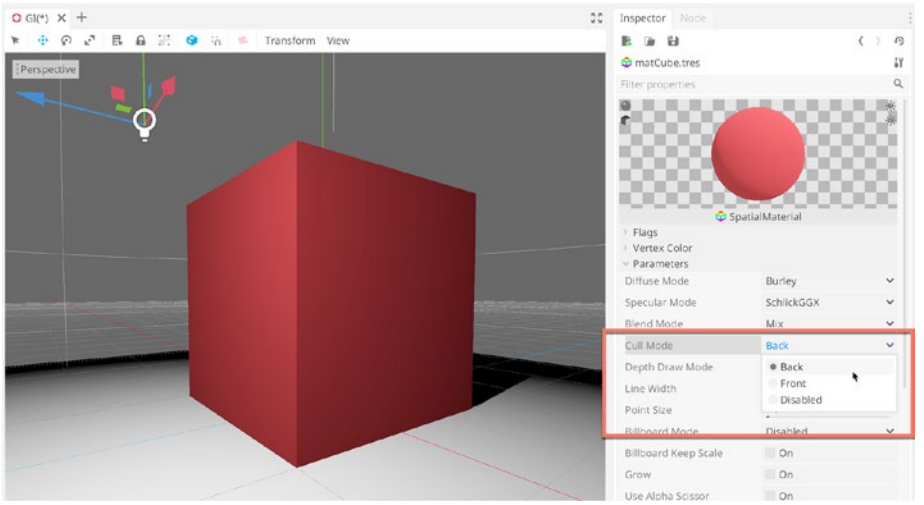

Figure 5-13. *Enabling Backface Culling on a Material*

You can make a material partially transparent by expanding the Flags Section, enabling the Transparent check box, and then by assigning a Alpha Value to the Albedo Color.

Global Illumination – Light Baking

We've surveyed Godot's most basic and fundamental tools for building effective 3D scenes, complete with lighting. But we're missing ***Indirect Illumination***, sometimes known as Global Illumination (GI), that is, light that strikes a surface – such as a wall – and then bounces off and continues. Perhaps it bounces off directly, as it does with a mirror (*Specular Reflections*), or perhaps it gets absorbed into the object and then reflected back after passing through it (a *Diffuse Interreflection*), or sometimes it even passes into an object and exits at a point quite distant from the entry (a *Subsurface Scatter)*. In all of these cases, light is reflected after making contact with an object; and this results in the continued transport of light

through the scene, potentially illuminating areas that are not directly in view of the original light source. By default, when using Godot's three major light types, no indirect illumination is calculated because it's computationally expensive to do so in real time. This is evident whenever we add a light to the scene and observe the results in the viewport. Consider Figure 5-14. In this figure, light directly hits the cube from a point light and then illuminates its faces; but any faces turned away from the light will simply render completely black.

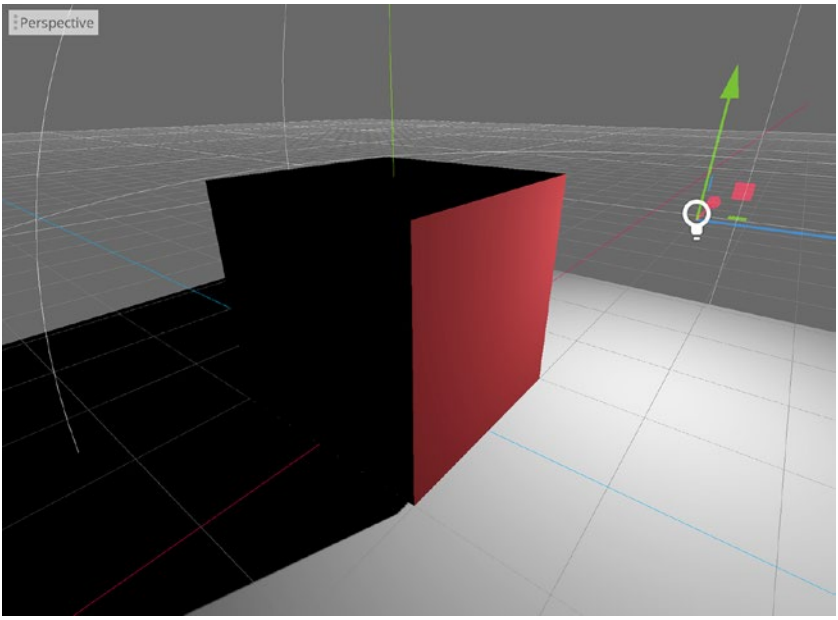

Figure 5-14. *By Default, Godot Only Calculates Direct Illumination*

Now, such *Direct Illumination* behavior is simply not physically accurate. Polygons facing away from a light should normally receive illumination from that light. We can solve this problem by using either one of Godot's *GI lighting* methods. The first method, considered here, is *Light Baking*. The other, considered in the next section, is the *GIProbe* system. Both systems simulate the transport of light through scene, beyond

direct illumination. So, both systems are capable of achieving great levels of realism. Light Baking should be preferred when creating games for a diversity of hardware, new and old. It's for when you need good-looking lighting at a low-performance cost. Light Baking saves most or all lighting data to textures ahead of runtime to simulate the effect of bounced lighting. Let's see how to set up Light Baking. We'll start by opening the *LightBox Godot* project included in the book companion files for this chapter. It features a basic scene and a mesh with all lighting removed using a *WorldEnvironment* node, as we've seen previously. See Figure 5-15.

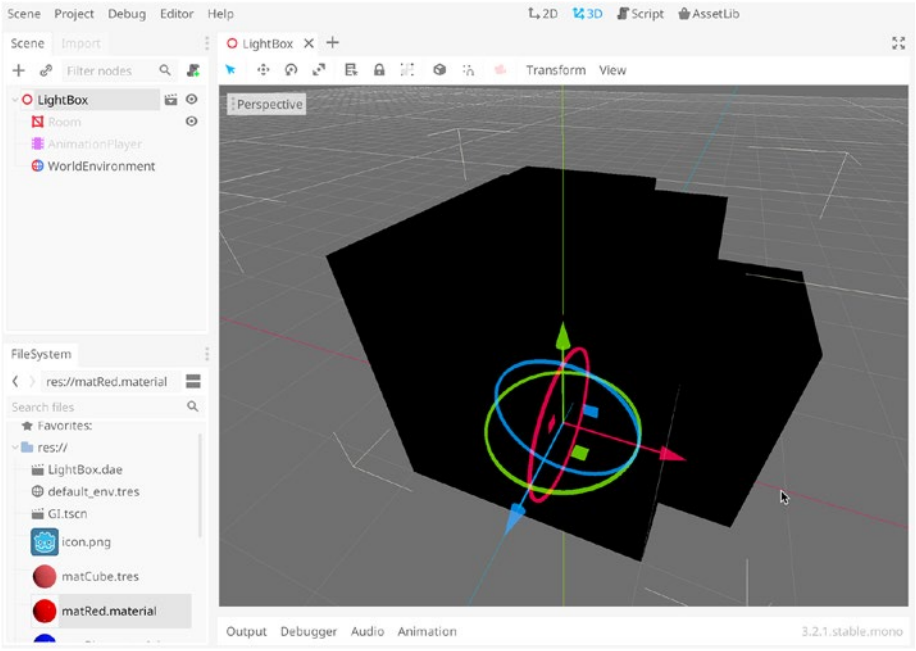

Figure 5-15. *Lightbox Room with Lighting Removed*

Next, let's add an *Omni Light* to the scene and position it close to the ceiling for simulating a ceiling light. By default, indirect illumination will not be calculated, as we've seen already. Don't worry, we'll take care of that shortly! See Figure 5-16.

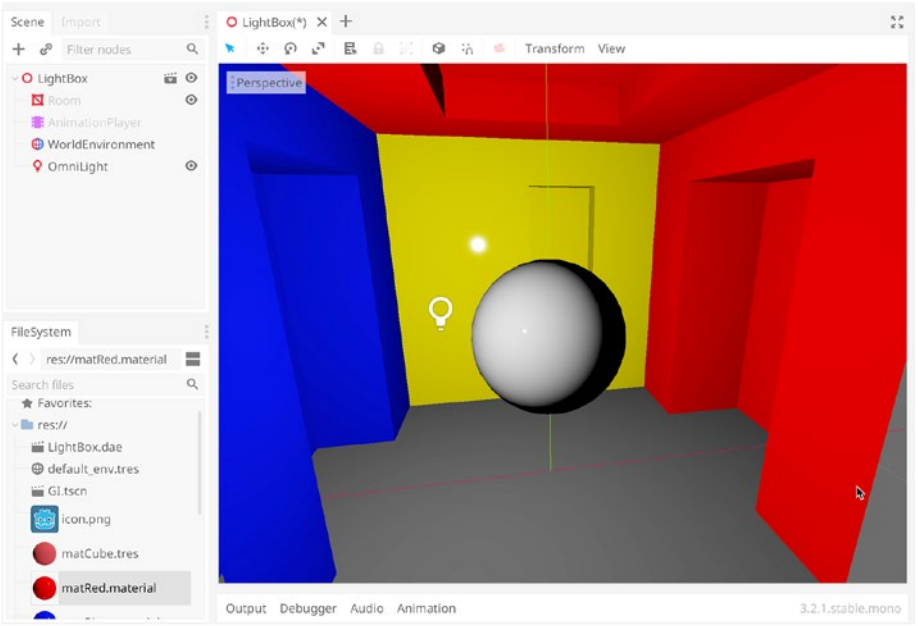

Figure 5-16. *Adding an Omni Light to the Scene*

Now let's add some shadows. Select the Omni Light from the Scene Tree and enable the Shadow field via the Inspector. From there, select a mid-tone color to create a believable look and feel. Also make sure that *Shadow Casting* is enabled on the world mesh instance too. See Figure 5-17.

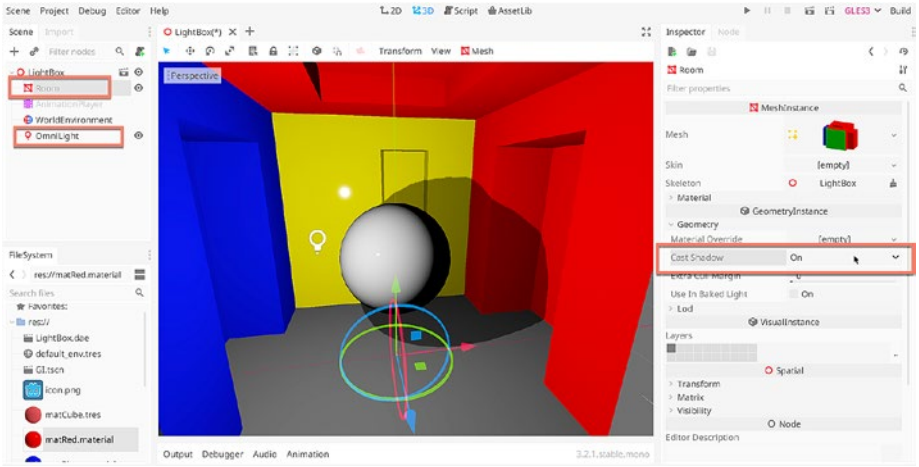

Figure 5-17. *Adding Shadows*

Wow. What a difference shadows can already make to a scene. Now, we'll configure the Baked Lighting. To do this, start by selecting the environment mesh instance. In our sample scene, we have only one mesh. But in larger scenes, you'd select all environment instances – walls, floors, ceilings, trees, and so on. Then choose *Mesh* ➤ *Unwrap UV2 for Lightmap/AO* from the viewport context menu. See Figure 5-18. This creates a new second set of UVs for the selected mesh. In this set, no UV islands will overlap within the UV space, ensuring that lighting and pixels can be rendered safely to a Lightmap texture and encode all scene lighting, resulting in no overlaps, conflicts, or aberrant effects.

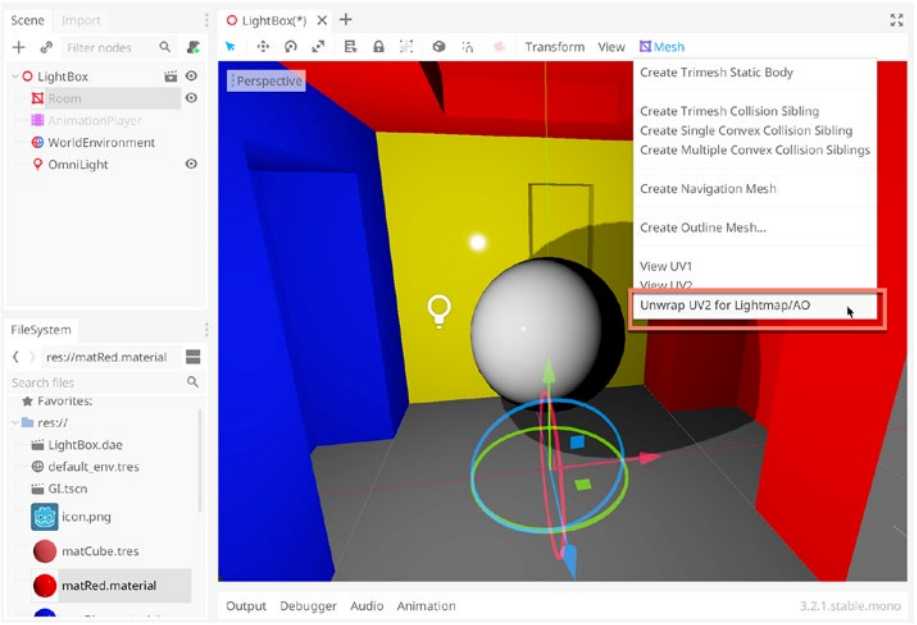

Figure 5-18. *Creating Lightmap UVs for the World Environment*

Now, right-click the scene root from the Scene Tree, and then add a new Baked Lightmap node using the Add Node dialog. The newly created node will eventually encode all lighting information. See Figure 5-19.

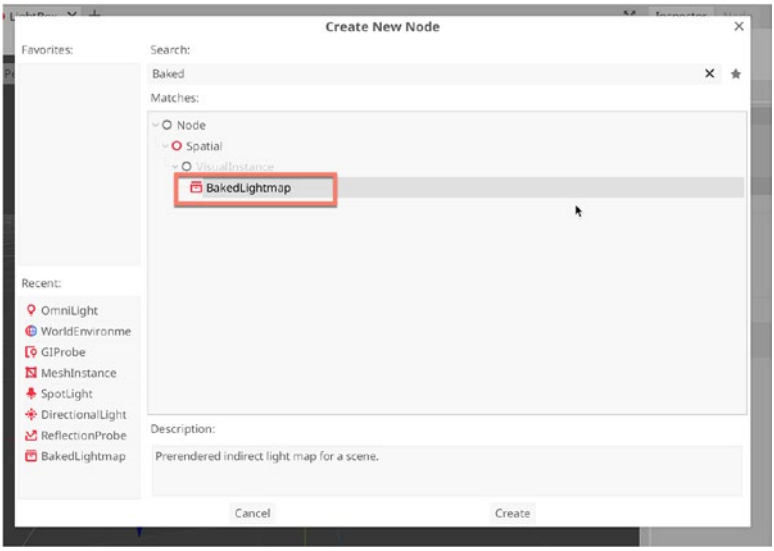

Figure 5-19. *Adding a Baked Lightmap Node*

The newly created object surrounds the scene with a highlighted cube gizmo, representing the total volume of the scene to be included in the Baked Lighting. Ideally, this box should be as small as possible while containing all areas of the scene. You can resize the box by clicking and dragging the dotted handles at the edges of the box or by adjusting the *X*, *Y*, and *Z* extents field from the *Bake* section of the Inspector. See Figure 5-20.

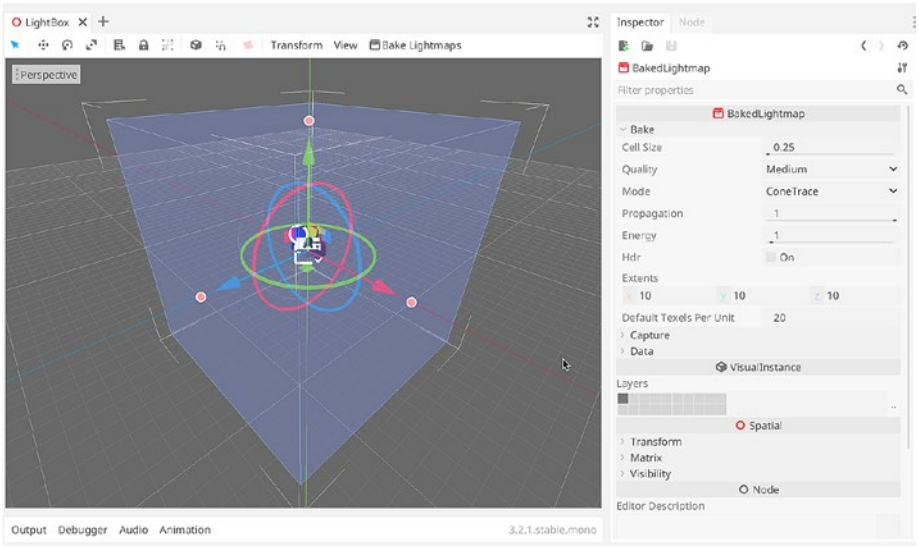

Figure 5-20. *Defining a Baking Volume…*

Next, define the light quality for capturing by adjusting the *Default Texels Per Unit* field from the Inspector. This determines the ratio between linear meters to pixels in the Lightmap texture. The default value is 20, meaning 20 Lightmap pixels will be produced for every "meter of surface" in the scene. This means that a winding, wrapping, and contorted surface will generate more lightmap pixels than a simple, flat plane extending across the same region. For our example, I'll reduce the Texels to 10. See Figure 5-21.

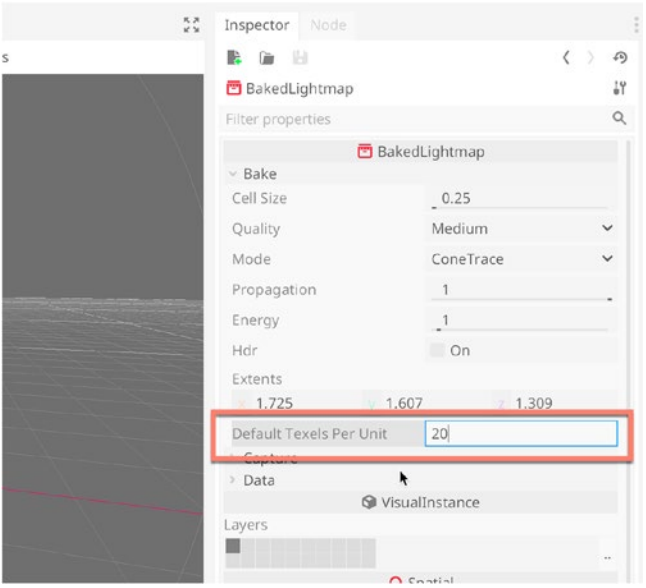

Figure 5-21. *Defining a Texel Ratio…*

Before starting the Baking Process, you'll need to mark each mesh instance to be included. Select the environment mesh. From the Inspector, expand the Geometry section, and then enable the *Use in Baked Light* field. See Figure 5-22.

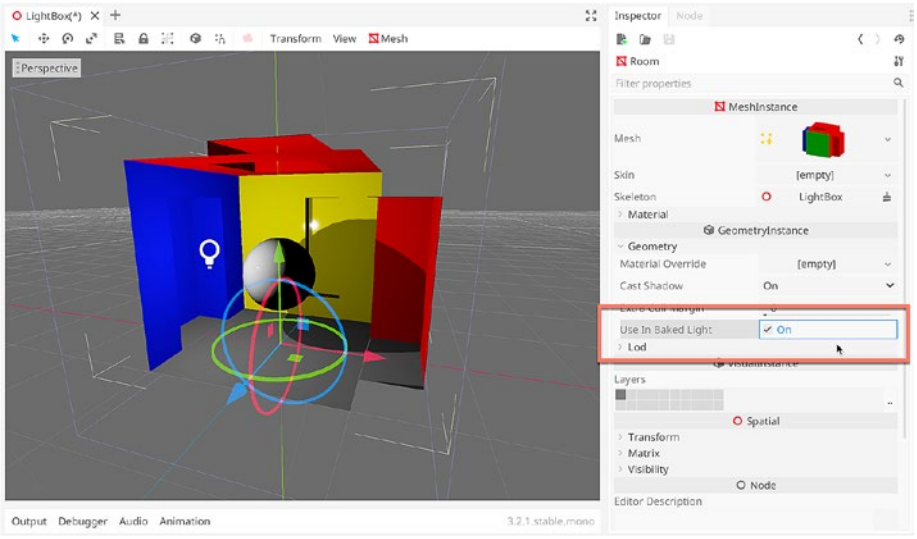

Figure 5-22. *Marking Baked Meshes*

In Unity, you mark geometry for light baking by enabling the *Static* check box, available on all Game Objects from the top-right hand side of Object Inspector. However, in Unity, static objects cannot move. In Godot, this limitation does not apply. Baked Objects can move. Moving objects automatically revert to an internal Light Probe system for calculating Indirect Illumination.

Finally, and optionally, let's disable the Gizmo Visibility for the Light Bake volume. Now that we've sized it appropriately for our scene, we don't need to continue viewing the volume itself. To do that, select *View* ➤ *Gizmos* ➤ *BakedLightmap* from the viewport context menu. See Figure 5-23.

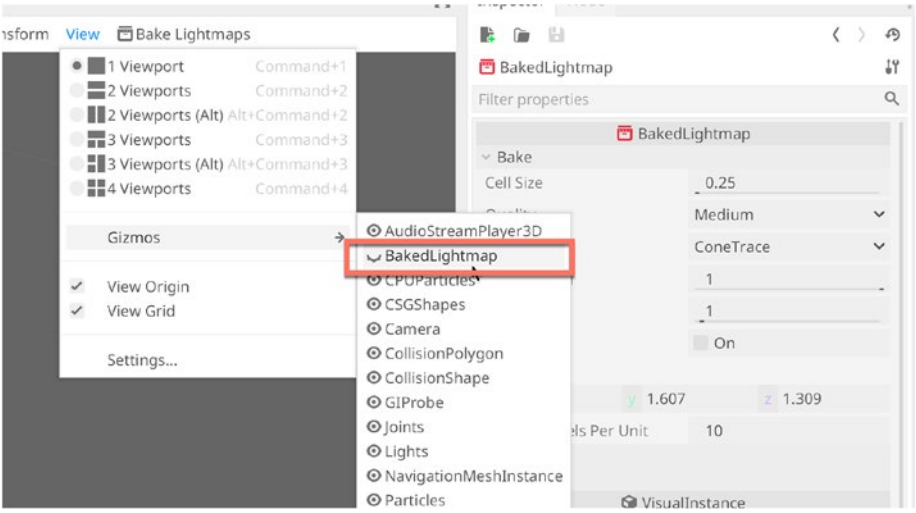

Figure 5-23. *Toggling BakedLightmap Visibility…*

Finally, you can bake the Lighting by choosing Bake Lightmaps from the viewport menu. Save the scene before doing so. This process may take a while, depending on the scene size. See Figure 5-24.

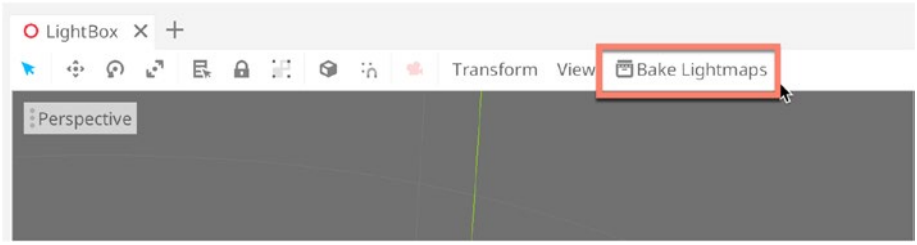

Figure 5-24. *Baking Lightmaps*

Notice the *dramatic* improvement on your scene lighting. What a difference indirect illumination makes to the scene realism. See Figure 5-25.

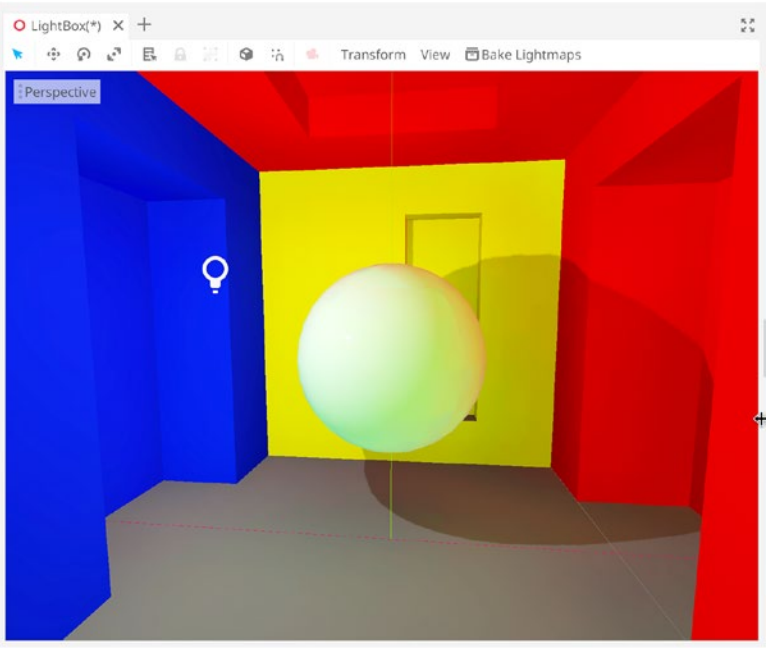

Figure 5-25. *Baking Lightmaps Increase the Believability of a Scene*

We can improve the lighting quality even further by using Post-Processing from the WorldEnvironment node. Let's try that now. Select the *WorldEnvironment* node from the Scene Tree and then expand the *Environment* section in the Inspector. This features lots of post-processing options. See Figure 5-26.

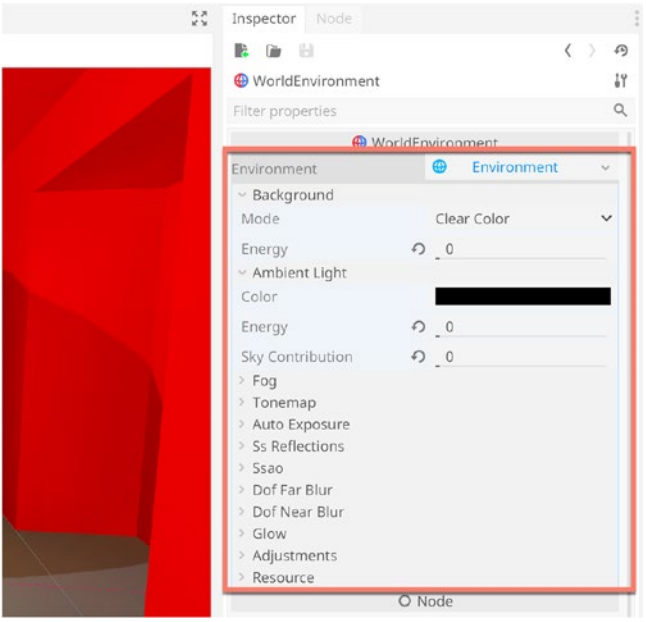

Figure 5-26. *Exploring Post-Processing*

First, expand the *Tonemap* group in the Inspector, and change the *Mode* to *Aces*. The different Tonemapping options adjust the brightness and saturation of your image dynamically, based on specific properties, to make your colors look punchier. See Figure 5-27.

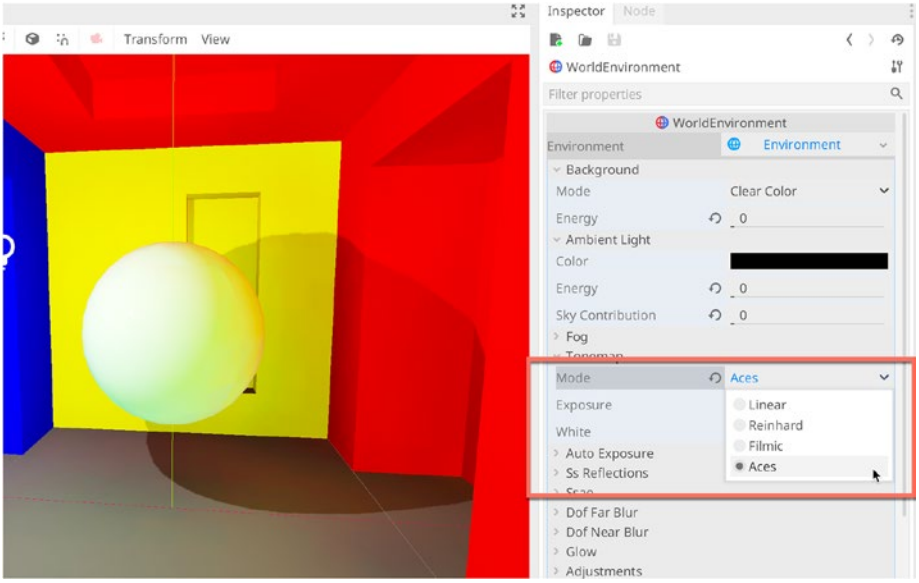

Figure 5-27. *Adding Tone Mapping*

Next, let's add *Contact Shadows*, also known as *Ambient Occlusion*. This refers to the darkening of objects at their edges, as they come into contact with other surfaces. To do this, expand the *SSAO* section (*Screen Space Ambient Occlusion*) and enable it. Then reduce the *Intensity* slider to *0.8*. See Figure 5-28.

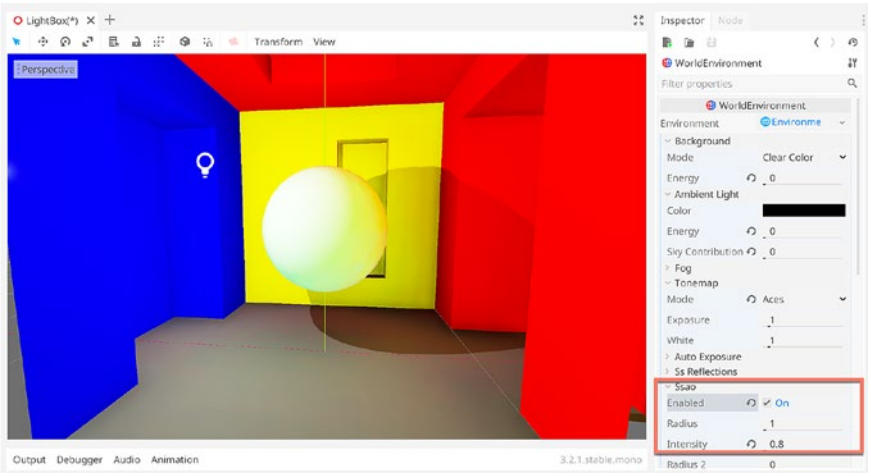

Figure 5-28. *Adding Contact Shadows*

Now let's add some *Bloom*. Bloom adds a subtle, but dreamy, blur to the render's highlight values. To do this, expand the *Glow* section. *Enable* the effect. Increase the *Intensity* and *Strength* fields to strengthen the effect overall, and then reduce the *HDR Threshold* to increase the range of elements affected. See Figure 5-29.

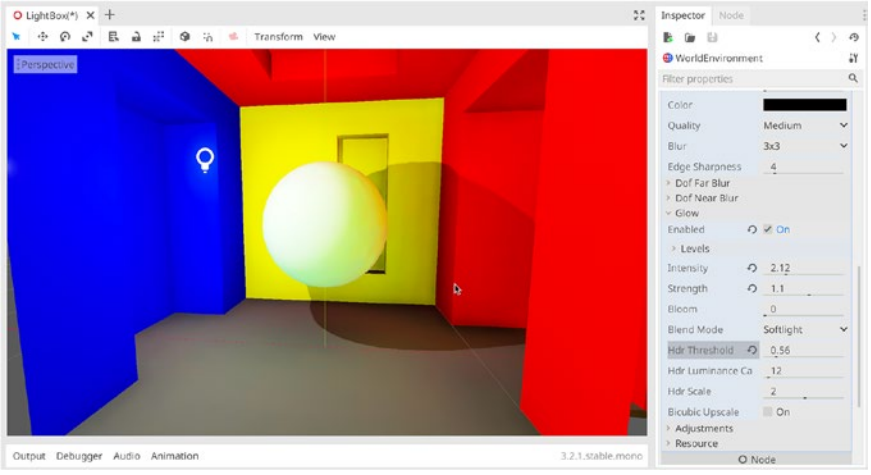

Figure 5-29. *Adding a Glow Effect*

This is looking good. Now, to finish things off, let's enable Anti-Aliasing to smooth off any sharp, jagged edges on the meshes. To enable this, select *Project* ➤ *Project Settings* from the application main menu. From there, display the Rendering ➤ Quality section, enable 4x for the *MSAA* setting (*Multi Sampling Anti-Aliasing*). See Figure 5-30.

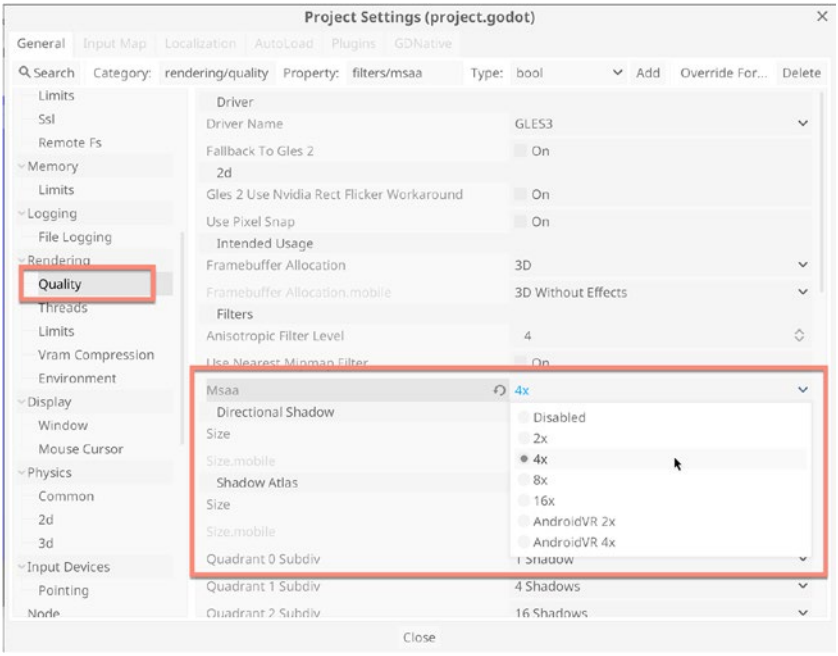

Figure 5-30. *Adding a Glow Effect…*

Great! And now you've successfully illuminated a 3D scene, complete with Light Baking and global illumination. Godot makes it easy. See Figure 5-31. Now, in the next section, we'll try it again, but using the *GIProbe* system!

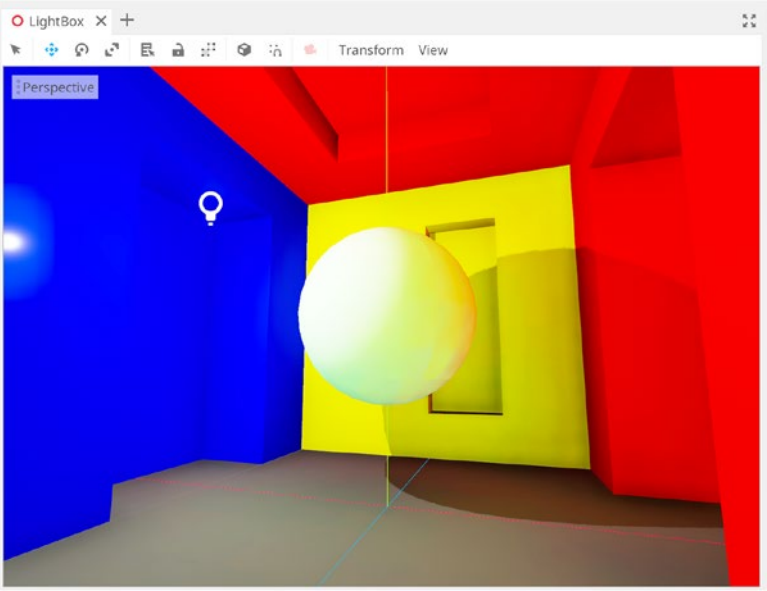

Figure 5-31. *A Light Baked Scene*

Global Illumination – GI Probes

In this section, we'll illuminate our Light Box room, the same as from the previous section. We'll begin again from the point shown in Figure 5-32. This room is included in the book companion files too, in the Scene *GIProbes*. This scene features only a Lightbox Room mesh, an Omni Light, and a default *WorldEnvironment* node. You'll notice again that our scene features only Direct Illumination. Throughout this section, we'll add Indirect Illumination, but by using the *GIProbe* system. Unlike Light Baking, which uses Image Textures to save scene lighting, the GIProbe system is more sophisticated and can bake a greater variety of lighting data to create realistic effects quickly and easily. However, the GIProbe is computationally expensive and is suitable for high-end PCs, consoles, and other more recent powerful devices.

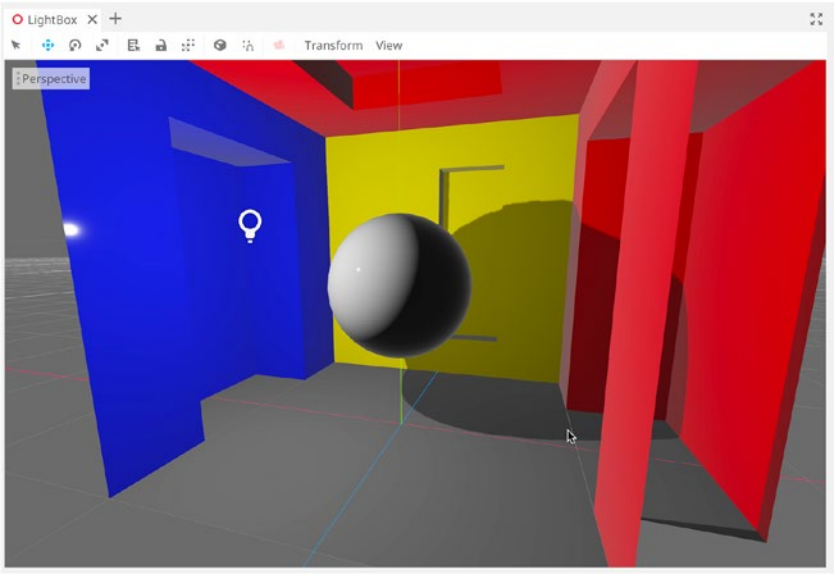

Figure 5-32. *Getting Ready for GIProbes*

As before, we must select all meshes and enable the *Use in Baked Lighting* option from the *Geometry* section of the Inspector. This ensures all meshes will be included in, and effected by, the GIProbe system. See Figure 5-33.

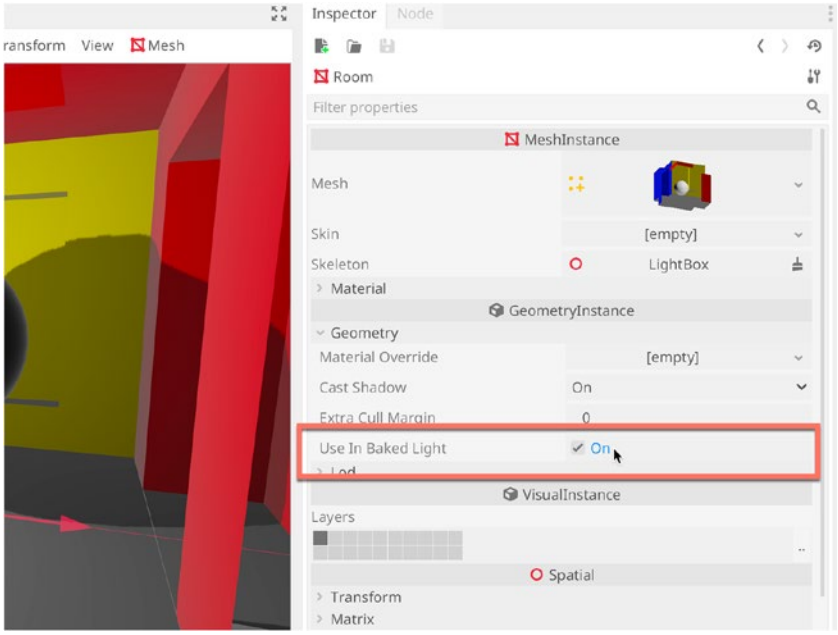

Figure 5-33. *Enabling Baked Lighting for Mesh Instances*

Next, add a *GIProbe* node to the scene from the Node dialog. Your scene needs only one GIProbe node for calculating indirect illumination in the scene. See Figure 5-34.

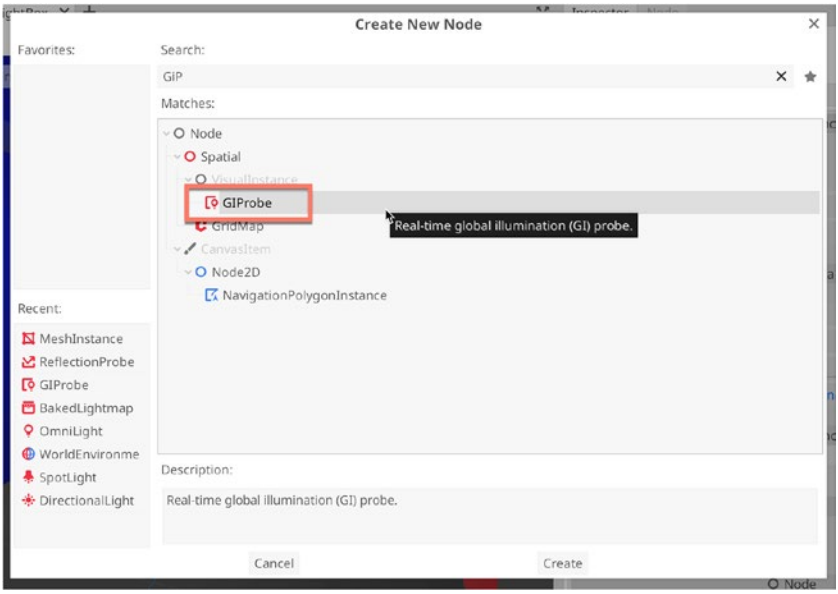

Figure 5-34. *Adding a GIProbe Node*

Initially, your GIProbe will probably be larger than your scene and will appear as a subdivided cube gizmo. The idea is to resize the gizmo to completely contain your scene, being as tightly sized as possible. You can resize the Gizmo by clicking and dragging the edge handles, inside the viewport, or by adjusting the X, Y, and Z fields for Extents group in the Inspector. See Figure 5-35.

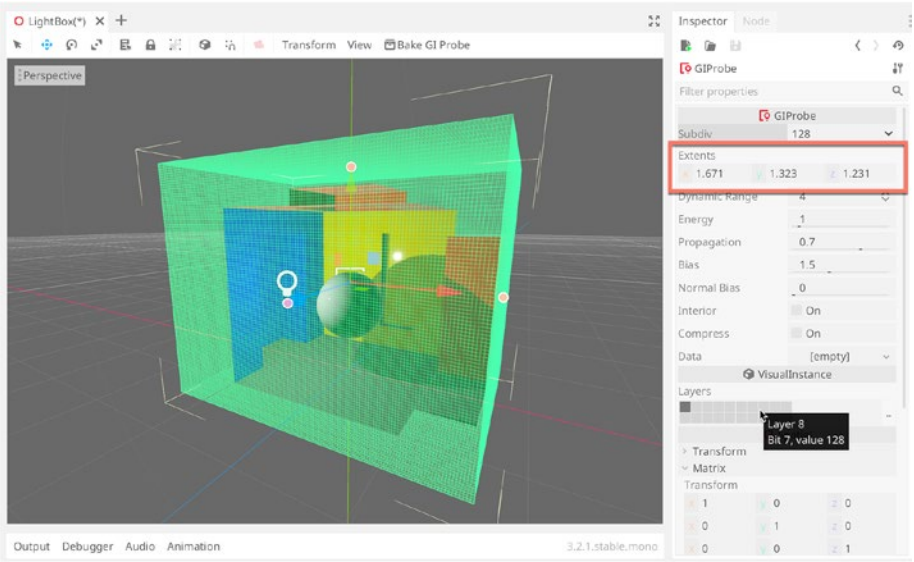

Figure 5-35. *Resizing the GI Probe*

Make sure the GIProbe node is selected in the Scene Tree, and then click *Bake GI Probe* from the viewport menu. When you do this, Godot calculates scene lighting and automatically adds indirect illumination. See Figure 5-36.

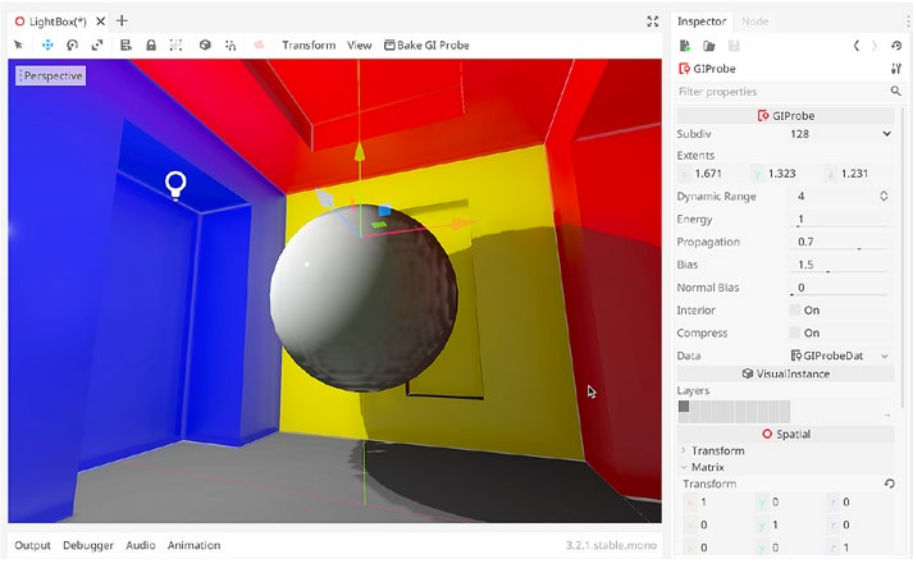

Figure 5-36. *Baked Scene with GI*

The GI will probably need tweaking if you're using the default settings. For example, by viewing the backside of the sphere in our scene (as featured in Figure 5-37), you'll notice some aberrant shadows or shading. See Figure 5-37. You may notice similar problems in other places too.

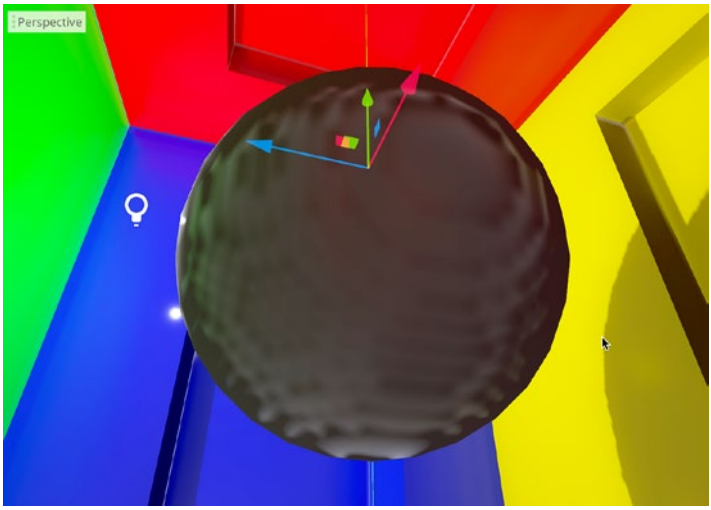

Figure 5-37. *Baked "Errors"*

Let's address these issues. First, mark our current scene as an "Interior," as opposed to an exterior scene, like a forest or a beach. This ensures that the sky or any skyboxes don't get baked into the scene lighting. You can do this by selecting the GIProbe node and then enabling the Interior check box from the Inspector. See Figure 5-38.

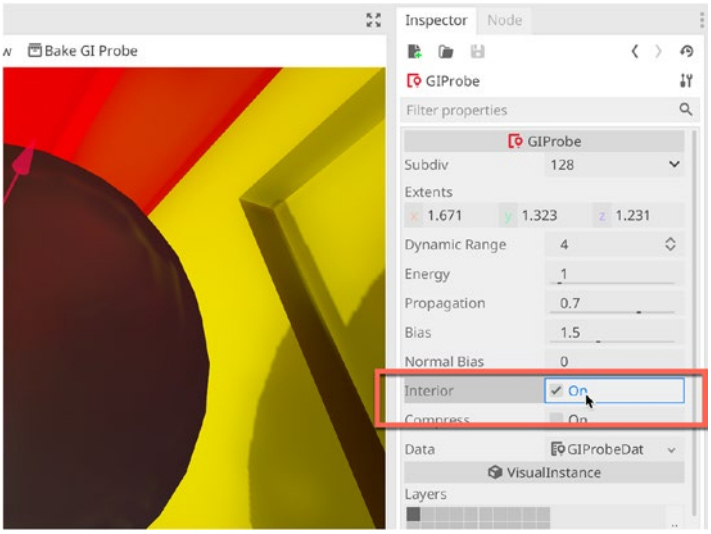

Figure 5-38. *Marking a Scene As Interior*

Next, let's adjust the *Normal Bias* field from the Inspector. This adjusts the fidelity of shadows and light bounces. Usually, 0 results in lower-quality shadows for smaller objects. Try adjusting the value to *0.16*. This will instantly have an effect in the viewport, and already the lighting will look much better. Great work! See Figure 5-39.

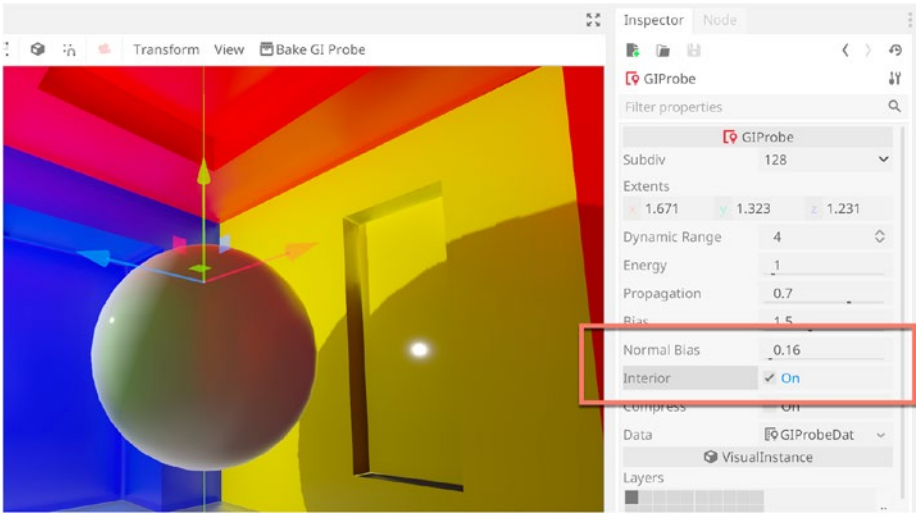

Figure 5-39. *Adjusting the Normal Bias for Better Shadows*

The *Subdiv* field at the top of the Inspector (when the GI Probe is selected) can be raised to higher values to improve lighting quality at the cost of performance. Here, we'll leave the GI Probe set to the default value of *128*. Ideally, this value should be as low as possible while still maintaining the results you need. See Figure 5-40.

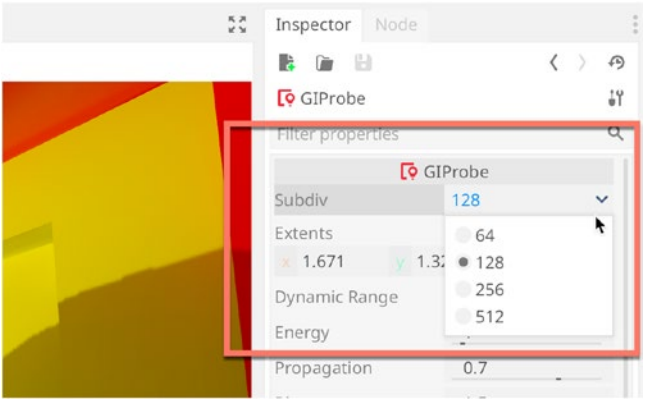

Figure 5-40. *Setting the GIProbe Subdivisions…*

Great. This scene is looking good. Now let's add Reflections. To do this, add a new Node, a *Reflection Probe Node*. This node acts like a camera; it captures images of the surrounding environment as a HDR map, and the captured data is assigned as a reflection to reflective objects. See Figure 5-41 for adding a Reflection Probe.

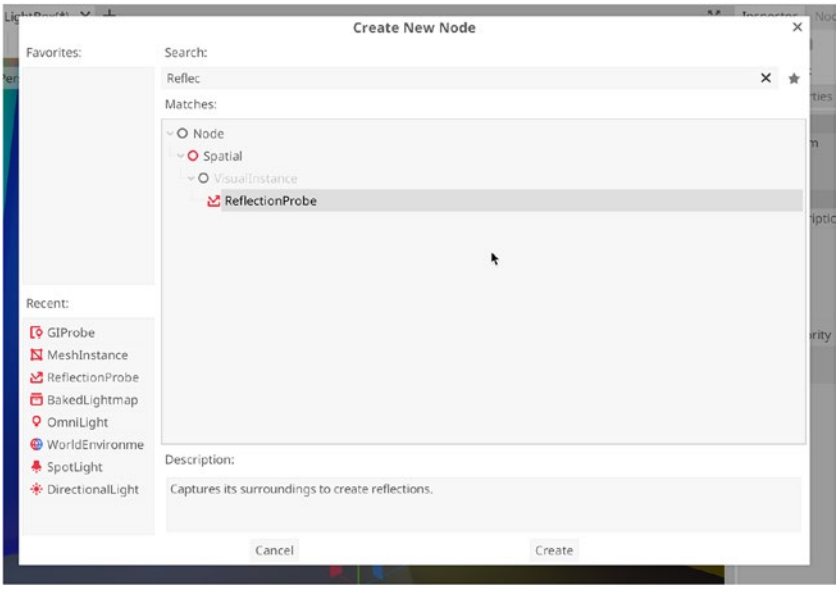

Figure 5-41. *Adding a GI Probe to the Scene…*

Next, use the Extents field of the Reflection Probe (from the inspector) or use the viewport gizmo handles to resize the reflection volume so it contains the scene completely. Again, this should be resized as tightly as possible. See Figure 5-42.

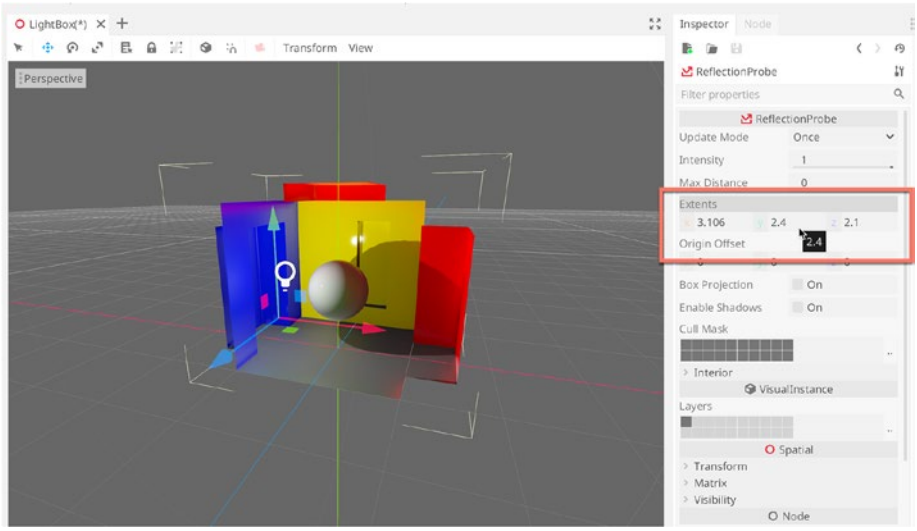

Figure 5-42. *Setting the Reflection Probe Extents to Contain the Scene*

And that's it! The scene is set up and ready to use Reflections. The *Update* Mode is specified as *Once* by default. This means the scene will be captured once and the image continually used as a reflection map. If your scene changes dramatically often – or its lighting changes often – you many need to change the mode to *Always*. Be careful, this is highly intensive. See Figure 5-43 for the complete scene with reflections.

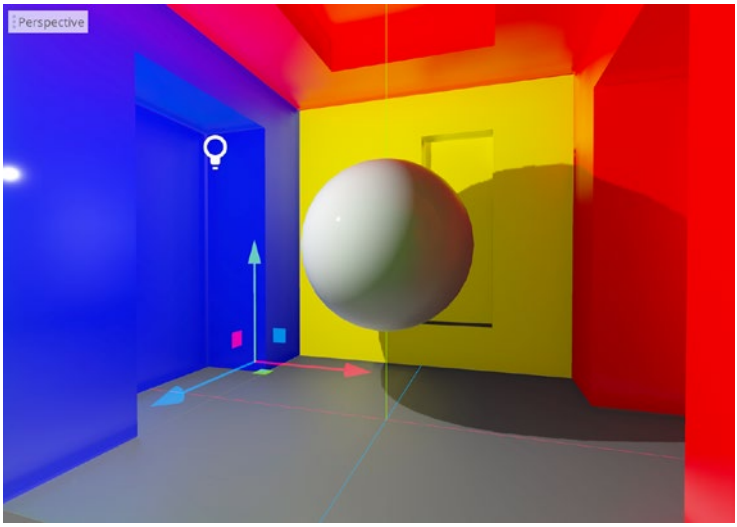

Figure 5-43. *Looking Good with Reflections*

Finally, let's enable our Post-Process effects from the WorldEnvironment node, as demonstrated in the preceding section. This includes Ambient Occlusion, Tone mapping, and Glows. See Figure 5-44.

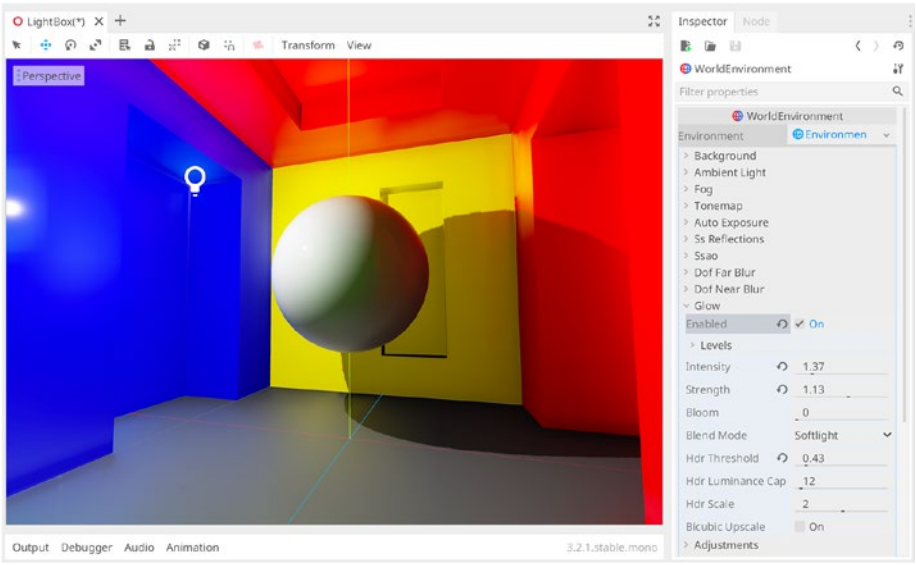

Figure 5-44. *Completed Scene with Post-Processing*

Summary

This chapter completes our analysis of lighting in 3D using Godot's two major lighting systems, Baking and GI Probes. For your projects, you'll need to choose the right one. Both are capable of great results: Baking trades quality for performance, and GI Probes trades in the opposite direction.

CHAPTER 6

Coding a First-Person Controller in C#

Unlike Unity, Godot doesn't ship with any built-in character controllers, neither third-person nor first-person. So, in this chapter, we'll create a reusable first-person controller rig from start to end in C# for Godot. Once created, we'll be able to drag and drop first-person controls directly into any scene. The WASD keyboard keys will move the camera forward, left, backward, and right, respectively. Holding down the Shift key enables run mode and the space bar initiates a jump. Mouse movement will control head orientation, and the first-person controller overall will support many physical interactions, including gravity, collisions, ramp movement, and more. See Figure 6-1.

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5_6

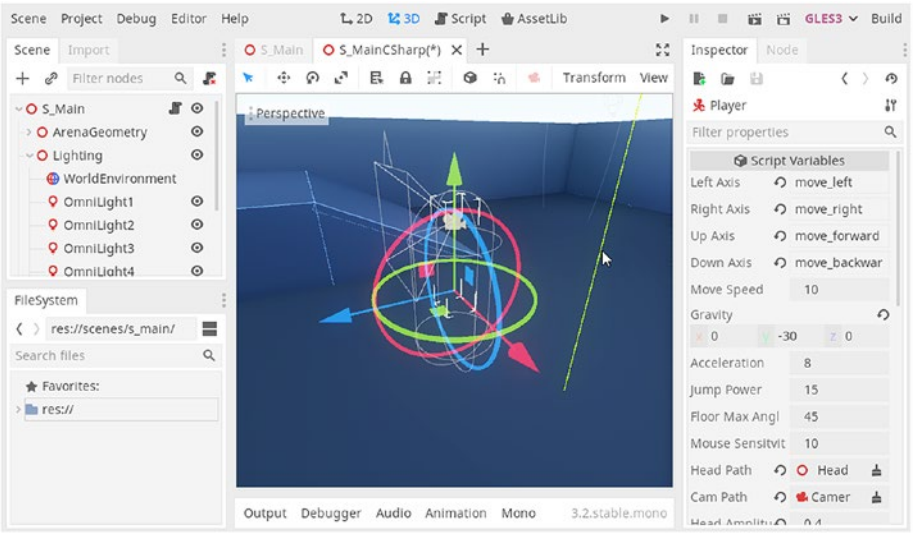

Figure 6-1. *Creating a First-Person Controller Rig*

Getting Started – Creating a Camera Scene

Let's begin the first-person character controller by creating a new scene to contain our objects. In Godot, a Scene behaves like a Unity Prefab. It allows us to create an asset in isolation, which can be added and reused across multiple scenes. To create a new Scene, select *Scene* ➤ *New Scene* from the application menu, and then choose 3D scene from the Create Root Node menu. See Figure 6-2.

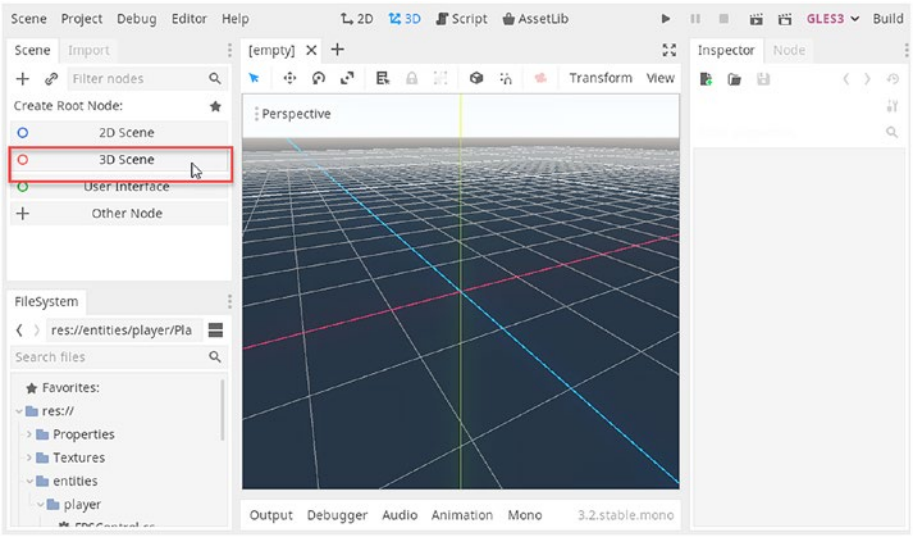

Figure 6-2. *Creating a 3D Scene*

Next, right-click the root node from the *Scene Tree*, and then choose *Add Child Node* to show the Create Node Dialog. From here, create a *KinematicBody Node (similar to a character controller in Unity – or a Rigidbody set to Kinematic Mode)*. A Kinematic Body is useful for building player-controlled characters or AI-controlled characters that need to react with physics and collision bodies. This node will become the root node – or main node – of the player character. See Figure 6-3.

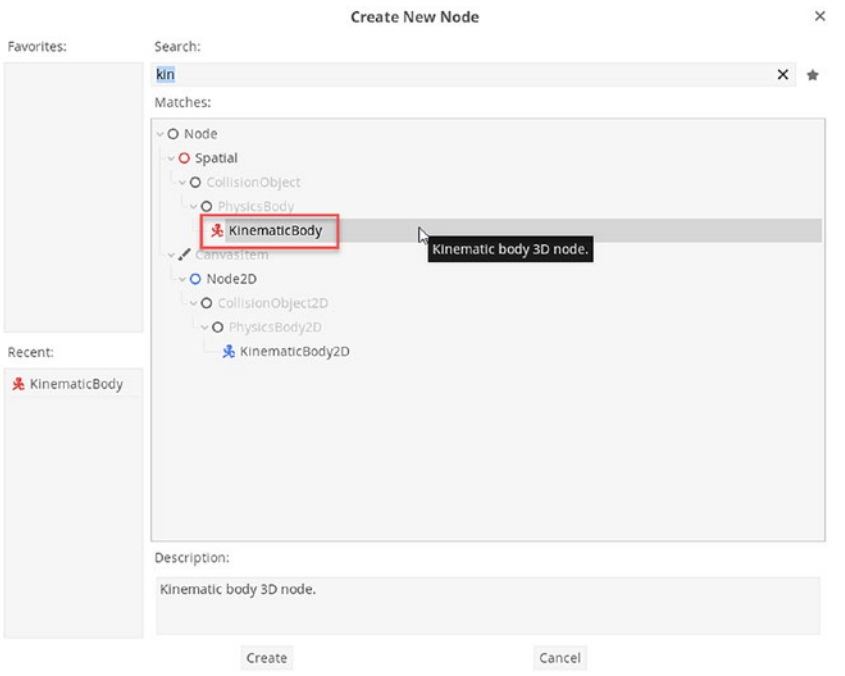

Figure 6-3. *Creating a Kinematic Body Node*

The newly created Kinematic Body node should become the root of the scene, that is, the topmost node. It'll be the topmost node of the player character, and there'll be several child nodes. Our current scene should contain only the player character – it's not intended to be played stand-alone but should be embedded into another scene, instantiated as a node, wherever first-person controls are needed. To do this, right-click the *KinematicBody* node in the Scene Tree and then select *Make Scene Root* from the context menu.

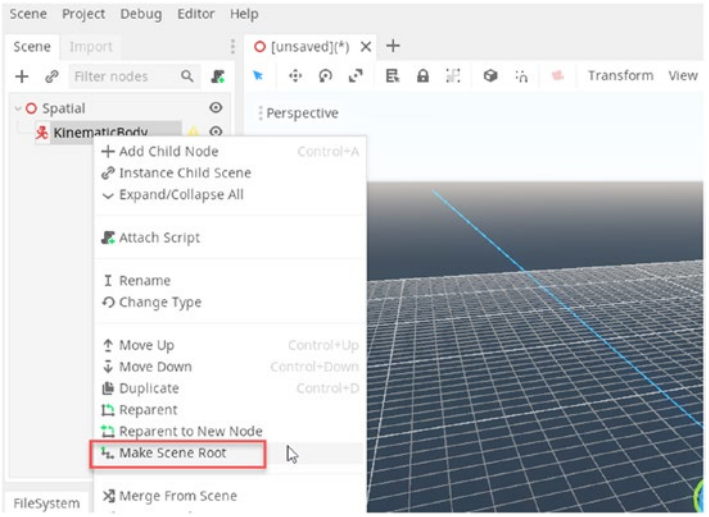

Figure 6-4. *Making the Kinematic Body the Scene Root*

As the Kinematic Body becomes the root, the previous root node (a 3D Spatial) will now become a child, and it may be safely deleted or left as is. We'll need a spatial node for later anyway. Every *KinematicBody* needs a *CollisionShape* child node to express the character's volume and size (equivalent to a Unity Collider). Let's add one now, specifically a *CollisionShape* node. See Figure 6-5.

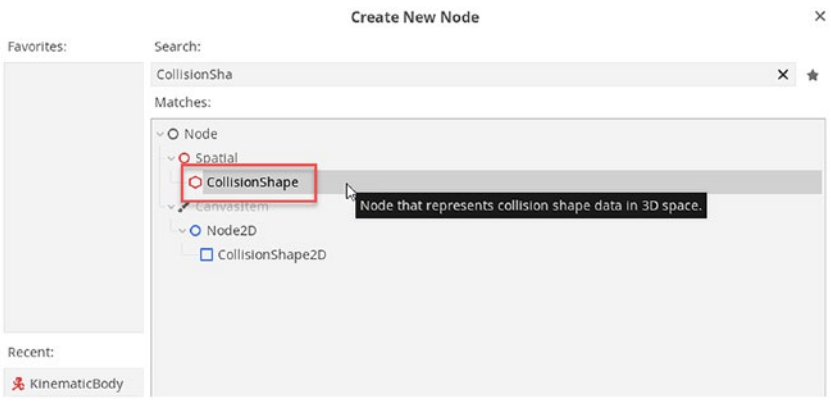

Figure 6-5. *Adding a CollisionShape Node*

The *CollisionShape* node begins as an empty spatial object with no width, height, or depth. To assign it shape, volume, and form, simply select the *CollisionShape* node and then move to the *Shape* field in the Inspector. From here, choose *New CapsuleShape*. The Capsule best approximates the volume of a humanoid character. See Figure 6-6.

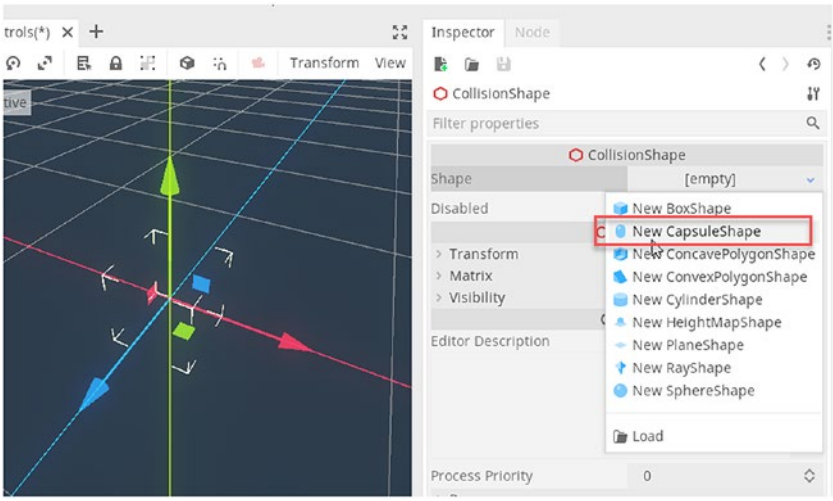

Figure 6-6. *Creating a Capsule Shape for the Kinematic Body*

You may need to rotate the *CollisionShape* by 90 degrees on the X axis – or a different axis (depending on your object's orientation) – and then tweak the CapsuleShape's settings to adjust its *Radius* and *Height*. The idea is to resize the CollisionShape to approximate a humanoid character, such as the player or an NPC. See Figure 6-7.

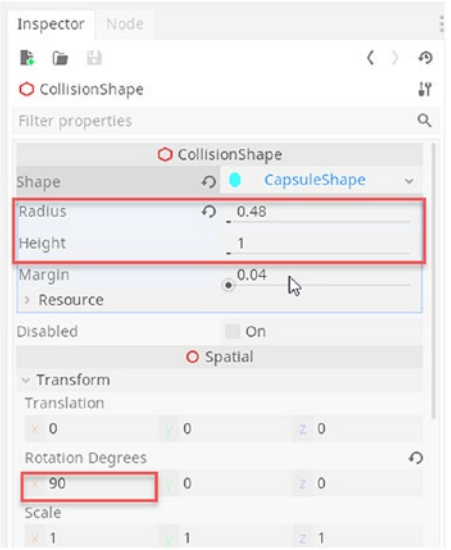

Figure 6-7. *Resizing the CapsuleShape*

Now let's focus on building the character's head, which will be the center of the camera's view. To start, create an empty spatial node – a child of the root – or use an existing spatial if you have one. Rename the node to "Head" and position it toward the top of the capsule at the center of the head location. Remember, the blue *forward arrow* should always point in the direction of the character's view. See Figure 6-8.

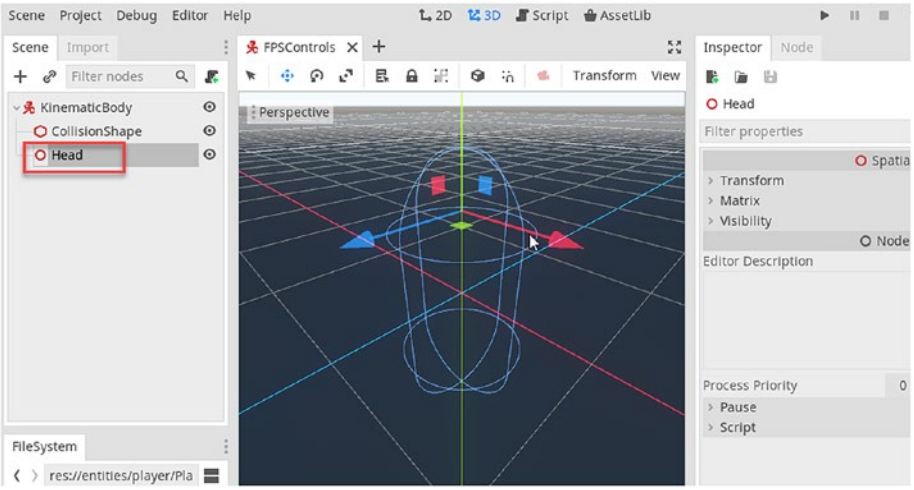

Figure 6-8. *Positioning the Character's Head*

Note The blue arrow of the transform gizmo is the forward vector. It must represent the direction in which the character is facing. All nodes within the player hierarchy should share the same orientation, with the forward vector pointing forward.

Having now used a spatial node to mark the head position, we should add a camera as a child object. Simply add a camera node using the *Node Creation Window*, and then rotate its transform if needed by 180 degrees around the *Y axis* to face forward. See Figure 6-9. The Player Rig is now completed and ready to code. Excellent!

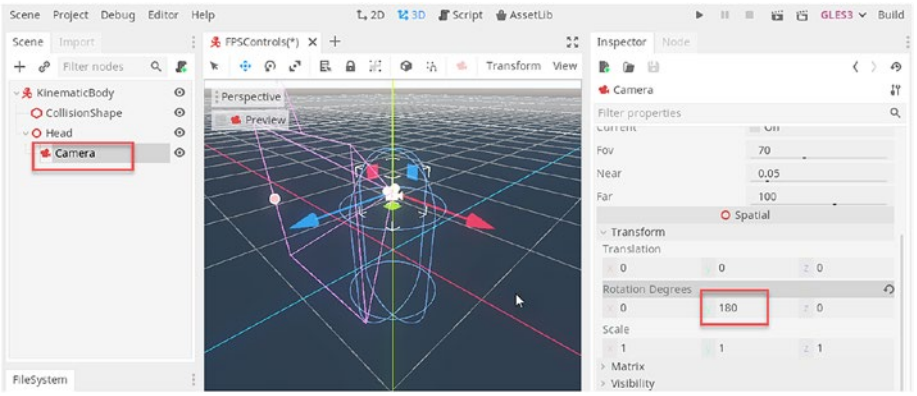

Figure 6-9. *Completing the Character Rig*

Player Movement and Key Bindings

One of the most important mechanics supported by any first-person controller is movement, both walking and running, specifically moving forward and backward using the *W* and *S* keys and strafing – or side stepping – left and right with *A* and *D*. To read input from these keys on the keyboard or to read input from buttons on a gamepad, we must first configure the game's Input Actions. This is simple to achieve. To do this, select *Project* ➤ *Project Settings* from the application menu, and then switch to the *Input Map* tab. See Figure 6-10.

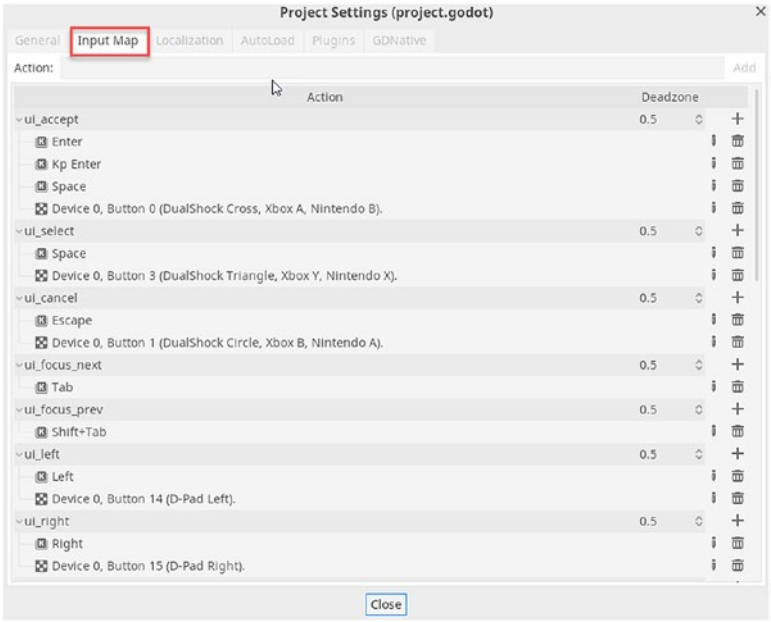

Figure 6-10. *Accessing the Input Action Settings from Project ➤ Project Settings*

We'll need to create four distinct Actions to read input from WASD: left, right, up, and down. In Godot, an Action is a form of key binding. It converts a button press or an analog control into numerical data that drives gameplay. To add Actions, simply type a name for each action into the Action field, and then click the *Add* button for each action to add four new actions in total. I've used the names *move_forward*, *move_backward*, *move_left*, and *move_right*, respectively. See Figure 6-11.

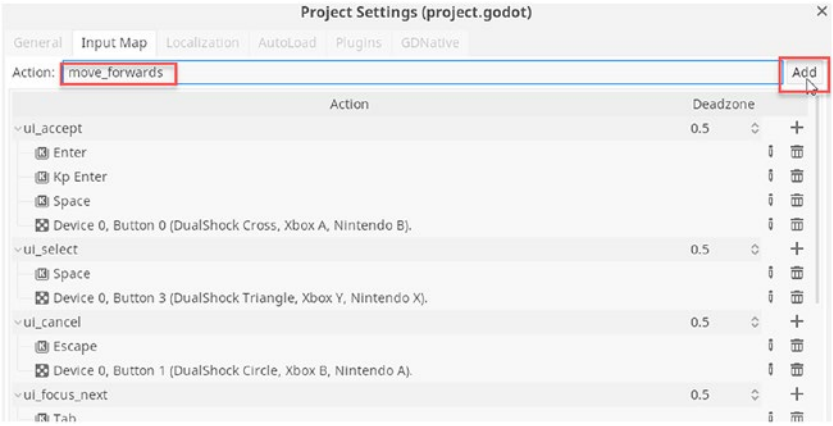

Figure 6-11. *Adding Input Actions from the Project Settings Dialog*

Each Input Action defines one or more key bindings or mappings, a conversion from a button press to axis data. Let's explore how to create bindings for the *move_forward* Action; and the remaining Actions are a repeat of that process. Click the *Add Event* button (+ icon), located beside the Input Action name. This creates a new key binding. On clicking *Add Event*, select *Key* from the context menu. This will link a keyboard button to the Input Action. See Figure 6-12.

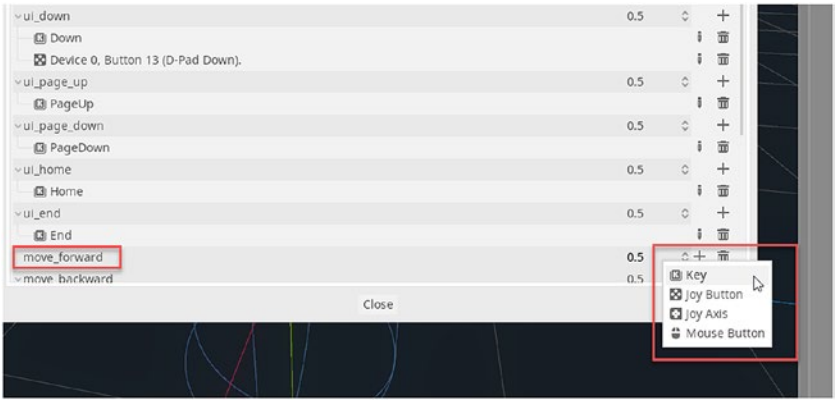

Figure 6-12. *Create a New Key Binding*

After clicking *Key* from the context menu, Godot prompts you to press the relevant keyboard key to form the key binding. For the *move_forward* Action, press the *W* key. See Figure 6-13.

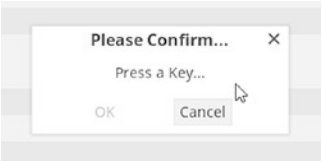

Figure 6-13. *Detecting a Keyboard Press for a Key Binding*

You can also add multiple keys to the same Action. Simply click *Add Event* again, and then add a new key binding. For the *move_forward* action, the *W* and *Up* arrow keys are most appropriate. See Figure 6-14 for the complete *move_forward* binding, along with all other actions, *move_backward*, *move_left*, and *move_right*. See Figure 6-14.

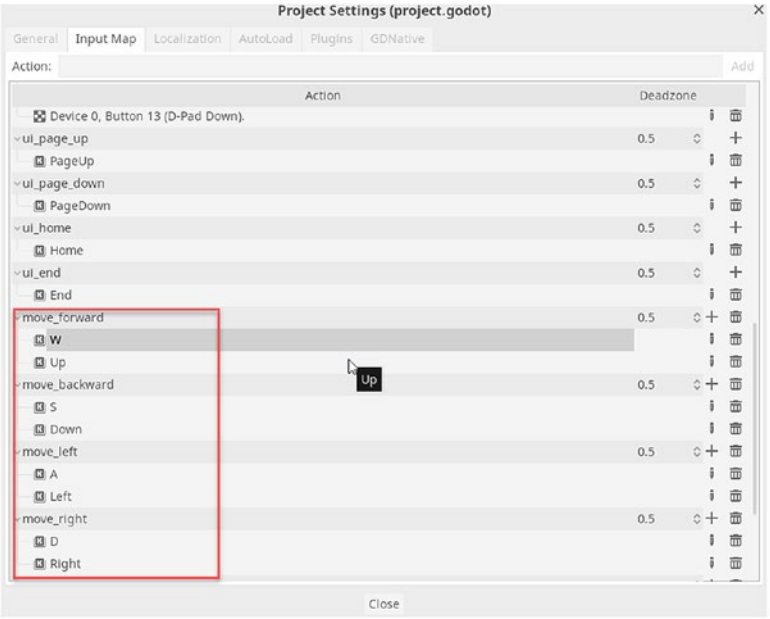

Figure 6-14. *Completing the Key Bindings for Forward, Backward, Left, and Right*

Reading Input Actions for Movement

The previous section demonstrated how to configure four input actions for player movement and all its associated key bindings. This links forward, backward, left, and right movement to the WASD keys and to the directional arrows. In this section, we'll create a new script file for the *PlayerCharacter*, which reads input through the Actions. To get started, create a new C# script file named *FPSControl.cs* and attach the script to the topmost KinematicBody node, as shown in Figure 6-15. The newly created script is shown in Listing 6-1.

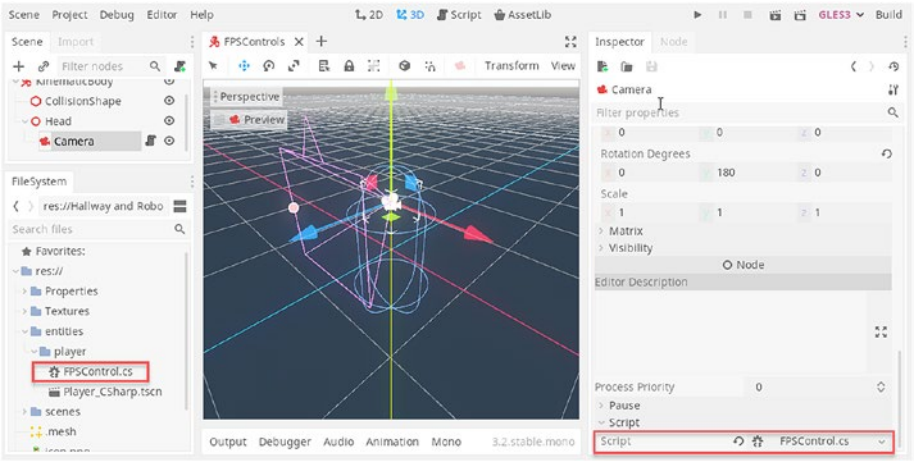

Figure 6-15. *Creating an FPSControl.cs Script File for Handling Player Movement*

Listing 6-1. Godot Auto-Generated FPSControl Script

```
using Godot;
using System;

public class FPSControl : KinematicBody
{
    // Declare member variables here. Examples:
    // private int a = 2;
    // private string b = "text";

    // Called when the node enters the scene tree for the first
       time.
    public override void _Ready()
    {

    }

//  // Called every frame. 'delta' is the elapsed time since
the previous frame.
//  public override void _Process(float delta)
//  {
//
//  }
}
```

Godot offers the *GetActionStrength* function to read input data from the specified Action. This function returns a smoothed floating-point value in the 0 to 1 range; 0 means "not pressed" and 1 means "fully pressed." These values, read from four Actions, can effectively be translated across two axes of movement. Specifically, *move_forward* and *move_backward* together represent the *Vertical* Axis. And *move_left* and *move_right* represent the *Horizontal* Axis. We can visualize both Horizontal and Vertical in the –1 to 1 range. –1 means down or left, and 1 means right or forward. 0 means

neither Action is pressed, or both are pressed together (because –1 + 1 = 0). To read input across the four Actions and to map them into our two axes of character movement, let's first add Export string variables to our class, allowing us to customize axis names from the inspector, which will be plugged into the function *GetActionStrength*. See the following code. Export is equivalent to the SerializedField attribute in Unity.

```
    [Export]
    private string LeftAxis, RightAxis,
   UpAxis, DownAxis = string.Empty;
```

Next, we'll use the *_process* event to read input from the actions and map them to the two axes. See Listing 6-2.

Listing 6-2. Converting Input Actions to 2D Movement, Horizontal and Vertical

```
public override void _Process(float delta)
{
        float Vertical = -Input.GetActionStrength(DownAxis) +
        Input.GetActionStrength(UpAxis);
        float Horizontal = Input.GetActionStrength(LeftAxis) +
        -Input.GetActionStrength(RightAxis);
}
```

Establishing Move Direction

Input read from Actions using *GetActionStrength* can be converted easily into two dimensions of local motion: horizontal and vertical – left and right and forward and backward, respectively. This motion may be expressed in two floating-point values only. This motion is local insofar as horizontal and vertical movement always relate to the *direction* in which the player is currently facing. Forward always means forward *for the player*, as opposed

to forward *in the world*, for example. For this reason, we'll need to convert the local movement strength to a world space direction for moving the player based on input. To do that, consider the revised *_process* event in Listing 6-3.

Listing 6-3. Converting Local Movements to World Velocity

```
public override void _Process(float delta)
{
        float Vertical = -Input.GetActionStrength(DownAxis) +
        Input.GetActionStrength(UpAxis);
        float Horizontal = Input.GetActionStrength(LeftAxis) +
        -Input.GetActionStrength(RightAxis);

        MoveDirection = Vector3.Zero;
        MoveDirection = HeadNode.Transform.basis.z * Vertical +
        HeadNode.Transform.basis.x * Horizontal;
        MoveDirection = MoveDirection.Normalized();
}
```

The code in Listing 6-2 depends on two additional class variables. The first is a Vector3 variable, *MoveDirection*, which always represents the normalized velocity of the player character, that is, the direction of movement. This variable is normalized and has a unit length, allowing it to be scaled by any speed needed. The other is the *HeadNode* variable, which refers to the empty spatial node that is the parent of the camera. Chapter 3 explores how to retrieve node references from a *NodePath* using the *GetNode* function. The head always represents the direction of travel for the player. This is because when moving forward and backward or left and right, the character is always moving in the direction of stare. Here's how you can read the *Head Node* and *Camera Node* objects at application startup, from NodePath variables defined in the Inspector.

```
public override void _Ready()
 {
     HeadNode = GetNode(HeadPath) as Spatial;
     CamNode = GetNode(CamPath) as Spatial;
     Input.SetMouseMode(Input.MouseMode.Captured);
 }
```

Note Notice the *Input.SetMouseMode* function is used to hide the system cursor, allowing for more intuitive head motion.

The *MoveDirection* variable is calculated each frame by using the *basis* member of the *Head Node*. The basis variable expresses three vectors, which are the local X, Y, and Z axes in world space. This means that the *basis.z* variable is the forward vector in world space and the *basis.x* is the right vector in world space.

Applying Gravity

The previous section detailed how to convert the player's move direction into world space using a normalized vector. This vector can be multiplied by a speed to offset the player along its velocity smoothly and seamlessly, frame by frame. However, this direction doesn't account for gravity, which is a force normally pulling everything downward toward the ground at an accelerated rate. To apply gravity, let's create a representation of it using a *Vector3* variable, expressing direction and strength.

```
[Export]
public Vector3 Gravity = Vector3.Zero;
```

A gravity vector of type Vector3.Zero will have no strength. For the player character, a vector of *(0, –30, 0)* works well in most cases. This can be specified in the Inspector. Other interesting values include *(0, 30, 0)*, which would push an object upward, like a perpetual jump, and *(–30, 0, 0)*, which would continually pull an object sideways.

Completing Player Movement

In this section, we'll complete the crucial functionality started in previous sections, namely, input, direction calculation, and gravity. Here, we'll complete the movement controls for the first-person character, specifically WASD motion. This code will span two functions, namely, *_Process* and *_PhysicsProcess*. *_Process* will be used to read input directly from the keyboard, and *_PhysicsProcess* will convert that input to physics-based movement for the *KinematicBody*. Consider the following *_Process* and *_PhysicsProcess* functions in Listing 6-4.

Listing 6-4. Reading Input and Moving the Kinematic Body

```
public override void _PhysicsProcess(float delta)
    {
        Vector3 TargetVel = Velocity;

        TargetVel.x = MoveDirection.x * SpeedMultiplier *
        MoveSpeed;
        TargetVel.z = MoveDirection.z * SpeedMultiplier *
        MoveSpeed;

        Velocity = Velocity.LinearInterpolate(TargetVel,
        Acceleration * delta);
        Velocity += Gravity * delta;
```

```
        Velocity.y = MoveAndSlideWithSnap(Velocity,
        Vector3.Down, Vector3.Up, true, 4, Mathf.
        Deg2Rad(FloorMaxAngle)).y;
    }

//  // Called every frame. 'delta' is the elapsed time since
the previous frame.
    public override void _Process(float delta)
    {
        float Vertical = -Input.GetActionStrength(DownAxis) +
        Input.GetActionStrength(UpAxis);
        float Horizontal = Input.GetActionStrength(LeftAxis) +
        -Input.GetActionStrength(RightAxis);

        MoveDirection = Vector3.Zero;
        MoveDirection = HeadNode.Transform.basis.z * Vertical +
        HeadNode.Transform.basis.x * Horizontal;
        MoveDirection = MoveDirection.Normalized();
    }
```

In Listing 6-4, the *_Process* event uses the *Input.GetActionStrength* function to read input data from the keyboard – on both the horizontal and vertical axes – and converts that to numerical data within the –1 to 1 range. This data is then constructed as a Vector3 structure, representing object velocity. The Transform.basis member expresses the three local axes of an object (Up, Forward, and Right) in world space. These are expressed in *basis.y*, *basis.z*, and *basis.x*, respectively.

Note More information on Input.GetActionStrength can be found at the Godot Documentation here: https://docs.godotengine.org/en/3.2/classes/class_input.html#class-input-method-get-action-strength.

More information on Transform.basis can be found at the Godot Documentation here: https://docs.godotengine.org/en/3.2/classes/class_transform.html#class-transform-property-basis.

Next, the *_PhysicsProcess* event uses the *Vector3.LinearInterpolate* function to smoothly transition the player's current position to a new position, based on our input velocity. The *MoveAndSlideWithSnap* function will move an object along its velocity but consider important physical properties about the world. Specifically, *MoveAndSlideWithSnap* ensures that, firstly, the player moves around successfully without passing through solid objects, like walls and enemies; secondly, the player can move up or down ramps and inclines; and thirdly, the player will move cleanly when standing on movable surfaces, like conveyor belts or ascending platforms. The *SlopeAngle* and *Floor Normal* parameters determine these behaviors.

Note More information on *MoveAndSlideWithSnap* can be found at the Godot Documentation here: https://docs.godotengine.org/en/3.2/classes/class_kinematicbody.html#class-kinematicbody-method-move-and-slide-with-snap.

Notice also that our code calculates gravity. That is, for each _ *PhysicsProcess*, our velocity is calculated and gravity applied to continually pull an object downward. This code uses the *Gravity* vector defined in the previous section.

Head Movement and Orientation

In the preceding section, the *_PhysicsProcess* function generated a velocity vector for the player character, based primarily on direct keyboard input. Pressing the *W* key or the *Up* arrow always moves the player *forward*, that is, forward *in the direction* in which the *camera is facing*. This section explores how to determine our look direction in code and how to control it using mouse movement. Let's start by reading input from the mouse, specifically mouse movement on the horizontal and vertical axes. This is important because Horizontal mouse movement (sliding left or right) controls head rotation around the Y axis, while movement along the Vertical direction controls head rotation around the X axis. The following code can be added to our *FPSControl* class. See Listing 6-5.

Listing 6-5. Reading Mouse Movement Input

```
public override void _Input(InputEvent motionUnknown)
{
    InputEventMouseMotion motion = motionUnknown as
    InputEventMouseMotion;

    if (motion != null)
        MouseMove = motion.Relative;
}
```

The *_Input* event is a native function of the *Node* class, which can be overridden by any script to handle input events, like key presses and mouse movement. This function will always fire when an input event changes. Here, this function is used simply to record the *Relative* mouse motion, that is, how much the mouse position has changed in screen space since the previous input event. The *MouseMove* variable is simply a *Vector2* class variable used to express the current mouse movement.

Note More information on *Input* event can be found online here at the Godot Documentation: https://docs.godotengine.org/en/3.2/tutorials/inputs/inputevent.html.

Next, having collected mouse movement from our *Input* event, we can re-code the *_Process* function as in Listing 6-6.

Listing 6-6. Handling Head Rotation

```
   public override void _Process(float delta)
    {
        float Vertical = -Input.GetActionStrength(DownAxis) +
        Input.GetActionStrength(UpAxis);
        float Horizontal = Input.GetActionStrength(LeftAxis) +
        -Input.GetActionStrength(RightAxis);

        MoveDirection = Vector3.Zero;
        MoveDirection = HeadNode.Transform.basis.z * Vertical +
        HeadNode.Transform.basis.x * Horizontal;
        MoveDirection = MoveDirection.Normalized();

if(MouseMove.Length() <= 0) return;

        Vector2 MouseResult = MouseMove * MouseSensitvity * delta;
        MouseResult = new Vector2(MouseResult.x, Mathf.
        Clamp(MouseResult.y, -PitchLimit, PitchLimit));
        HeadNode.Transform = HeadNode.Transform.
        Rotated(Vector3.Up, -Mathf.Deg2Rad(MouseResult.x));
        HeadNode.Transform = HeadNode.Transform.
        Rotated(HeadNode.Transform.basis.x,
        Mathf.Deg2Rad(MouseResult.y));
```

```
        HeadNode.RotationDegrees = new Vector3(Mathf.
        Clamp(HeadNode.RotationDegrees.x, -PitchLimit,PitchLimit),
              HeadNode.RotationDegrees.y, HeadNode.
              RotationDegrees.z);
        MouseMove = Vector2.Zero;
    }
```

The revised *_Process* function now supports head rotation based on mouse movement. The complete head rotation is calculated from the *MouseResult* Vector2 structure, where *MouseResult.X* expresses rotation around the camera's local Y axis and *MouseResult.Y* expresses rotation around the camera's local X axis. The *MouseSensitivity* floating-point variable is a multiplier to either strengthen or weaken the rotation effect. Notice the *Spatial* variable *HeadNode* references the *Spatial* parent of the camera. This node rotates according to mouse input.

Head rotation is a multistep process, iterating through *Pitch* (rotation around X) and *Yaw* (rotation around Y). First, the calculated *Pitch* is constrained by the *Mathf.Clamp* function and a *PitchLimit*. The *PitchLimit* variable is a *Vector2* object whose X and Y components define the minimum and maximum rotation in degrees, respectively, that the head may rotate from its starting orientation around the local X axis. This prevents the camera from rotating upside down, which would normally happen if the player continually moved the mouse up or down without stopping or reversing direction. Second, the *Transform.Rotated* function is called to rotate the Camera (or more specifically, it's empty parent) around its local Y and then its local X. Notice the utility function *Mathf.Deg2Rad* is called during this process, converting angles in degrees to radians, as expected by Godot's *Rotated* function.

Note More information on *Transform.Rotated* can be found at the Godot online documentation here: https://docs.godotengine.org/en/3.2/classes/class_transform.html#class-transform-method-rotated.

The final step is ensuring the *Spatial.RotationDegrees* variable (a Vector3 object) is clamped within our acceptable pitch range. *RotationDegrees* parallels Unity's *EulerAngles* variable. It's a Vector3 structure expressing Pitch, Yaw, and Roll – rotation around each local axis. So by rotating the camera based on mouse movement as we have done here, the *CameraNode.basis.z* variable always expresses our forward direction *in world space* – the direction we're facing.

Jumping and Being Grounded

Our first-person player character must support a jumping behavior, that is, the ability to jump in the air and then to fall down again by gravity. Jumping introduces two important states for the player, namely, *grounded* and *not grounded*. Grounded is true whenever the player is in contact with the floor and false when not. Knowing when a character is grounded is important for preventing double jumps – for not being able to jump while you're already jumping or falling and for enabling animations or character behaviors that change depending on whether the character is jumping, falling, or standing. For our character, a *space bar* press initiates a jump. To code this, let's first configure the relevant Input Action, as with movement, by simply accessing the *Project* ➤ *Project Settings* menu and by adding a Jump action from the Input Map, as demonstrated earlier in this chapter. See Figure 6-16.

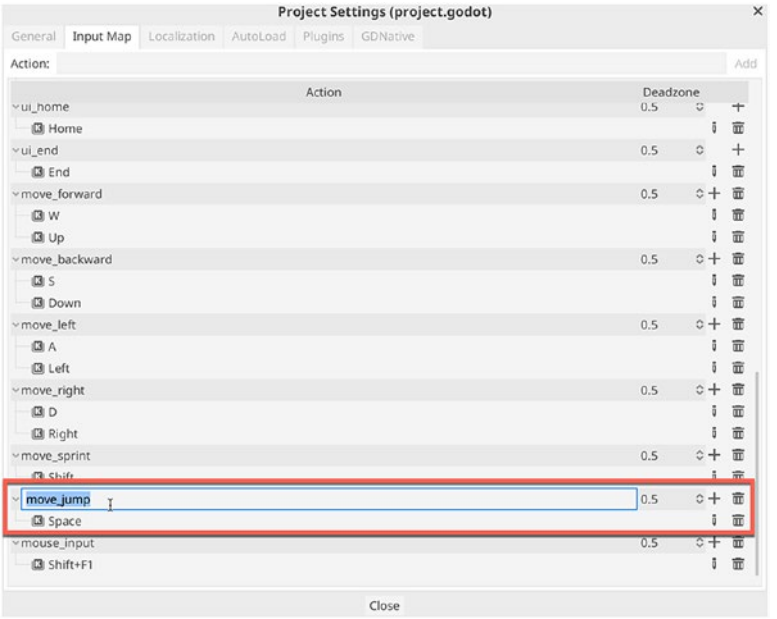

Figure 6-16. *Configuring the Jump Action…*

In addition, let's add three new variables: *JumpPower*, a float describing the strength of our jump; *Grounded*, to describe whether we are currently touching the ground or not; and *Snap*, which will be plugged into the *MoveAndSlideWithSnap* function, inside *_PhysicsProcess*. When we're jumping, Snap should be *Vector3.Zero* to ensure we can lift off without being connected to the floor, and when falling, it should be *Vector3.Down*, to ensure we fall back to ground.

```
private bool Grounded = false;

[Export]
private float JumpPower = 15f;
private Vector3 Snap = Vector3.Down;
```

Now we've added our variables, let's refine the *_Process* function to detect and handle a Jump situation. See Listing 6-7.

Listing 6-7. Detecting a Jump

```
if(Input.IsActionJustPressed("move_jump") && Grounded)
{
        Velocity.y = JumpPower;
        Snap = Vector3.Zero;
}
else
        Snap = Vector3.Down;
```

A jump works like inverted gravity. Gravity pulls you down with acceleration, and a jump pulls you up with deceleration. Listing 6-7 detects a jump press to initiate a jump, but it depends on the *Grounded* variable being accurate, representing our contact status with the floor. Godot provides a convenient function in the *KinematicBody* class to detect floor contact, namely, *IsOnFloor*. This function returns true if the *KinematicBody* is on the ground. The *_PhysicsProcess* function can therefore be written in full as in Listing 6-8.

Listing 6-8. Checking for Being Grounded

```
public override void _PhysicsProcess(float delta)
{
    Vector3 TargetVel = Velocity;
    Grounded = IsOnFloor();

    TargetVel.x = MoveDirection.x * SpeedMultiplier *
    MoveSpeed;
    TargetVel.z = MoveDirection.z * SpeedMultiplier *
    MoveSpeed;
```

```
    Velocity = Velocity.LinearInterpolate(TargetVel,
    Acceleration * delta);
    Velocity += Gravity * delta;
    Velocity.y = MoveAndSlideWithSnap(Velocity, Snap,
    FloorNormal, true, 4, Mathf.Deg2Rad(FloorMaxAngle)).y;
}
```

Excellent work! We now have a first-person controller that can move and jump, and its head can rotate from mouse movement. Let's move forward and create walk and sprint modes.

Note More information on the *IsOnFloor* function can be found at the Godot documentation online here: https://docs.godotengine.org/en/3.2/classes/class_kinematicbody.html#class-kinematicbody-method-is-on-floor.

Walking and Sprinting

Basically, sprinting can be coded as "fast walking." Previously, our first-person controller had a walk speed determining how quickly it moved around the scene when horizontal and vertical input was given. Here, we'll adjust the code to support two different speeds depending on whether the *Shift* key is being held down. A Shift key press should be configured as an Action from the Input Map – "move_sprint." To support running, we'll create a *SpeedMultiplier* float variable. A value of 1f is the "default speed." As a result, run functionality can be added to the *_Process* function with the Listing 6-9.

Listing 6-9. Add Sprint Functionality

```
if(Input.IsActionPressed("move_sprint"))
    SpeedMultiplier = RunMultiplier;
else
    SpeedMultiplier = 1f;
```

Head Bobs and Sine Waves

This section adds our final, polish feature to the first-person controller, specifically the *Head Bob*, that is, the smoothed, bobbing (*up and down*) motion of the head as it naturally adjusts to character locomotion, walking through the scene. A Head Bob is, simply put, a Y axis position adjustment, back and forth that applies only when the character is moving. To implement a Head Bob, we'll use a *Sine Wave*. This is a smooth, repeated curve that we can programmatically move through to determine the Y position of the character's head over time. This position can be generated from just two float variables, *Amplitude* and *Frequency*. Amplitude determines the maximum height of the curve and thereby the strength of the bob. Frequency determines the smoothness of the effect, the ratio of bobs to linear distance. Let's get started with Head Bobbing by creating three new class variables, which we'll use later.

```
[Export]
public float HeadAmplitude, HeadFrequency = 1f;
//Current Y-Pos of the head
private float BobHeight = 0f;
```

Next, we'll code a new and separate function to handle Head Bobbing independently, which will be called every frame, inside *_Process*. See Listing 6-10.

Listing 6-10. Coding a Head Bob

```
private void HeadBob(float delta)
{
    Transform T = CamNode.Transform;

    if(MoveDirection.y != 0 && Grounded)
    {
        BobHeight += delta;
        BobHeight = Mathf.Wrap(BobHeight,0f,360f);
        float LerpedHeight = Mathf.Sin(BobHeight *
        HeadFrequency) * HeadAmplitude;
        T.origin.y = LerpedHeight;
        CamNode.Transform = T;
        return;
    }

    T.origin.y = Mathf.Lerp(T.origin.y, 0f, delta);
    CamNode.Transform = T;
}
```

Excellent! You've now coded a great looking, and fully tweakable, head bob. Let's break down the code here. Firstly, the *HeadBob* function accepts a delta value, which can be passed from the *_Process* event. This is simply a *deltaTime* value. Inside the *_Process* event, the *HeadBob* function begins by checking our Grounded status to ensure we're on the ground. It then proceeds to wrap an incremental, numerical value between 0 and 360 into *Mathf.Sin* to retrieve a value along the since curve. This becomes the LerpedHeight and updates the camera origin on Y.

Completing the FPS Controller

So, we're done. The first-person controller is now fully coded. Great job! Godot doesn't ship with a C# first-person controller, so we've just made an important contribution. Check out the complete code for the controller in Listing 6-11.

Listing 6-11. The Complete First-Person Controller

```
using Godot;
using System;

public class FPSControl : KinematicBody
{
    [Export]
    private string LeftAxis, RightAxis, UpAxis, DownAxis =
    string.Empty;

    private Vector3 MoveDirection = Vector3.Zero;
    private Vector3 FloorNormal = Vector3.Up;

    [Export]
    public float MoveSpeed = 10f;

    [Export]
    public Vector3 Gravity = Vector3.Zero;

    private bool Grounded = false;

    [Export]
    private float Acceleration = 8f;

    [Export]
    private float JumpPower = 15f;
    public Vector3 Velocity;
```

```
[Export]
public float FloorMaxAngle = 45f;

private Vector3 Snap = Vector3.Down;

private Vector2 MouseMove = Vector2.Zero;

[Export]
public float MouseSensitvity = 10f;

[Export]
public NodePath HeadPath;
private Spatial HeadNode;

[Export]
public NodePath CamPath;
private Spatial CamNode;

[Export]
public float HeadAmplitude, HeadFrequency = 1f;
private float BobHeight = 0f;

[Export]
public float PitchLimit = 45f;

[Export]
public float RunMultiplier = 1f;
private float SpeedMultiplier = 1f;

// Called when the node enters the scene tree for the
   first time.
public override void _Ready()
{
    HeadNode = GetNode(HeadPath) as Spatial;
    CamNode = GetNode(CamPath) as Spatial;
    Input.SetMouseMode(Input.MouseMode.Captured);
```

```
    }

    public override void _PhysicsProcess(float delta)
    {
        Vector3 TargetVel = Velocity;
        Grounded = IsOnFloor();

        TargetVel.x = MoveDirection.x * SpeedMultiplier *
        MoveSpeed;
        TargetVel.z = MoveDirection.z * SpeedMultiplier *
        MoveSpeed;

        Velocity = Velocity.LinearInterpolate(TargetVel,
        Acceleration * delta);
        Velocity += Gravity * delta;
        Velocity.y = MoveAndSlideWithSnap(Velocity, Snap,
        FloorNormal, true, 4, Mathf.Deg2Rad(FloorMaxAngle)).y;
    }

//  // Called every frame. 'delta' is the elapsed time since
the previous frame.
    public override void _Process(float delta)
    {
        float Vertical = -Input.GetActionStrength(DownAxis) +
        Input.GetActionStrength(UpAxis);
        float Horizontal = Input.GetActionStrength(LeftAxis) +
        -Input.GetActionStrength(RightAxis);

        MoveDirection = Vector3.Zero;
        MoveDirection = HeadNode.Transform.basis.z * Vertical +
        HeadNode.Transform.basis.x * Horizontal;
        MoveDirection = MoveDirection.Normalized();
        HeadBob(delta);
```

```
    if(Input.IsActionJustPressed("move_jump") && Grounded)
    {
        Velocity.y = JumpPower;
        Snap = Vector3.Zero;
    }
    else
        Snap = Vector3.Down;

    if(Input.IsActionPressed("move_sprint"))
        SpeedMultiplier = RunMultiplier;
    else
        SpeedMultiplier = 1f;

    if(MouseMove.Length() <= 0) return;

    Vector2 MouseResult = MouseMove * MouseSensitvity * delta;
    MouseResult = new Vector2(MouseResult.x, Mathf.
    Clamp(MouseResult.y, -PitchLimit, PitchLimit));
    HeadNode.Transform = HeadNode.Transform.
    Rotated(Vector3.Up, -Mathf.Deg2Rad(MouseResult.x));
    HeadNode.Transform = HeadNode.Transform.
    Rotated(HeadNode.Transform.basis.x, Mathf.
    Deg2Rad(MouseResult.y));
    HeadNode.RotationDegrees = new Vector3(Mathf.
    Clamp(HeadNode.RotationDegrees.x,
    -PitchLimit,PitchLimit),
                HeadNode.RotationDegrees.y, HeadNode.
                RotationDegrees.z);
    MouseMove = Vector2.Zero;
}
```

```
    private void HeadBob(float delta)
    {
        Transform T = CamNode.Transform;

        if(MoveDirection.y != 0 && Grounded)
        {
            BobHeight += delta;
            BobHeight = Mathf.Wrap(BobHeight,0f,360f);
            float LerpedHeight = Mathf.Sin(BobHeight *
            HeadFrequency) * HeadAmplitude;
            T.origin.y = LerpedHeight;
            CamNode.Transform = T;
            return;
        }

        T.origin.y = Mathf.Lerp(T.origin.y, 0f, delta);
        CamNode.Transform = T;
    }

    public override void _Input(InputEvent motionUnknown)
    {
        InputEventMouseMotion motion = motionUnknown as
        InputEventMouseMotion;

        if (motion != null)
            MouseMove = motion.Relative;
    }
}
```

Testing the Controller

You've built a first-person controller. Great! Now let's test it. To do this, create a completely new empty scene. And then add a selection of mesh instances to it, boxes, cubes, cylinders, and others, to build some scenery. Ensure that you generate static bodies for each mesh to enable collisions. See Figure 6-17. Remember, the associated book companion files feature a basic collision scene for your testing.

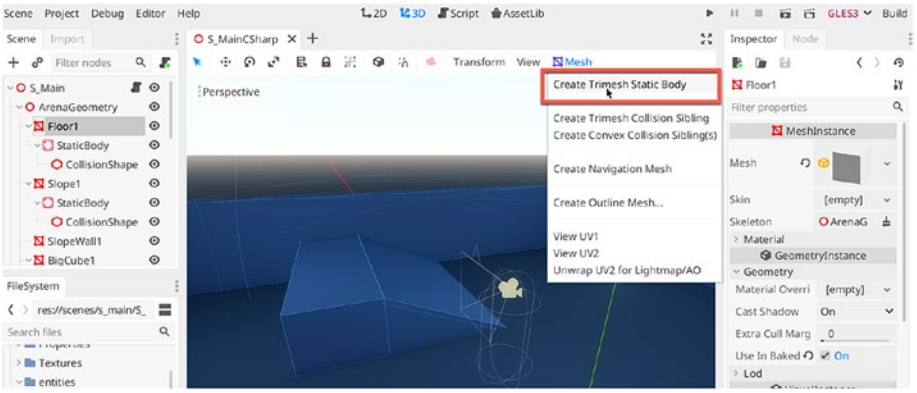

Figure 6-17. *Creating a World for Testing Our Controller…*

Next, let's add the player character. The first-person controller was created as a completely separate scene, which means it can be dragged and dropped into any other scene, just like a *Unity Prefab* or an *Additive Scene*. See Figure 6-18.

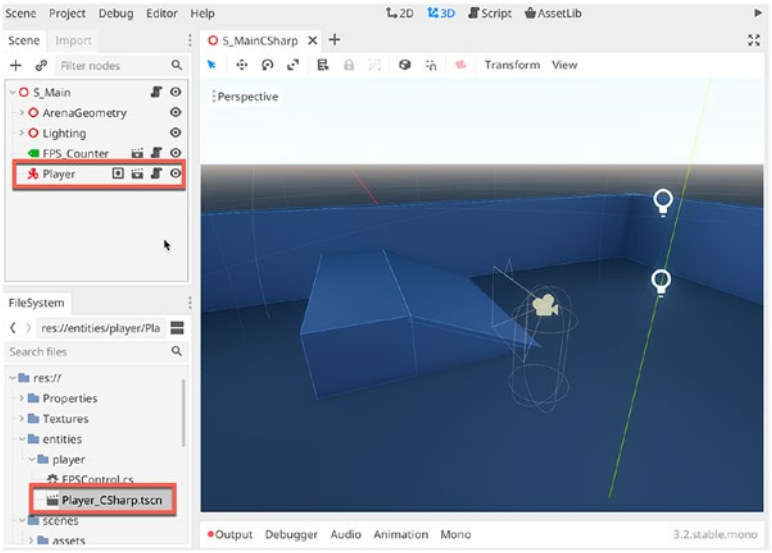

Figure 6-18. *Adding the Player Scene*

Once the player is added, you'll need to set your starting properties for the first-person controller. By selecting the player object, you can adjust its fields from the inspector, customizing them *for active scene*, as opposed to the original. See Figure 6-19.

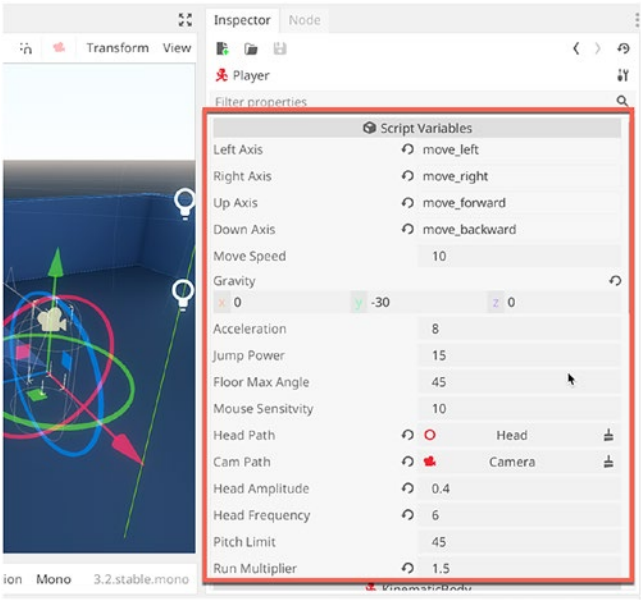

Figure 6-19. *Customizing the First-Person Controller*

Now you're ready to try out the new first-person controller. Simply save the scene, and then press the *Play Scene* button from the toolbar. See Figure 6-20. Great work!

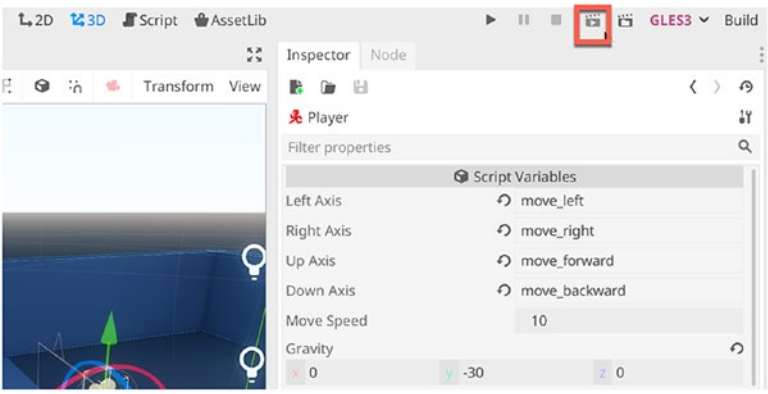

Figure 6-20. *Playing the First-Person Controller Scene…*

Summary

Congratulations. In this chapter, you coded a first-person controller in C# for Godot, complete with run, jump, and head-control mechanics. In reaching this far, you've seen varied examples for reading input, both event-based and polling-based; ways to move objects during *_PhysicsProcess* to account for physical interactions; how to use scenes as Prefabs, nested inside other scenes; and how to create input actions and more.

CHAPTER 7

Tips and Tricks

Previous chapters typically focused on specific groups of features within Godot, such as 2D and 3D worlds. This chapter, by contrast, focuses on how to achieve common tasks within Godot. The Unity documentation on such tasks is extensive. But the Godot documentation is limited. So let's now take a look at some common tasks.

How to Make Objects Look at the Cursor

You'll often want objects – like the player or enemy characters – to face the mouse cursor as it moves around the screen. Twin-stick shooter games are a common example. In these games, the player normally controls character movement – up, down, left, and right – using keyboard arrow keys or gamepad presses; and they control player orientation through mouse movement, specifically the player *look direction*. See Figure 7-1.

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5_7

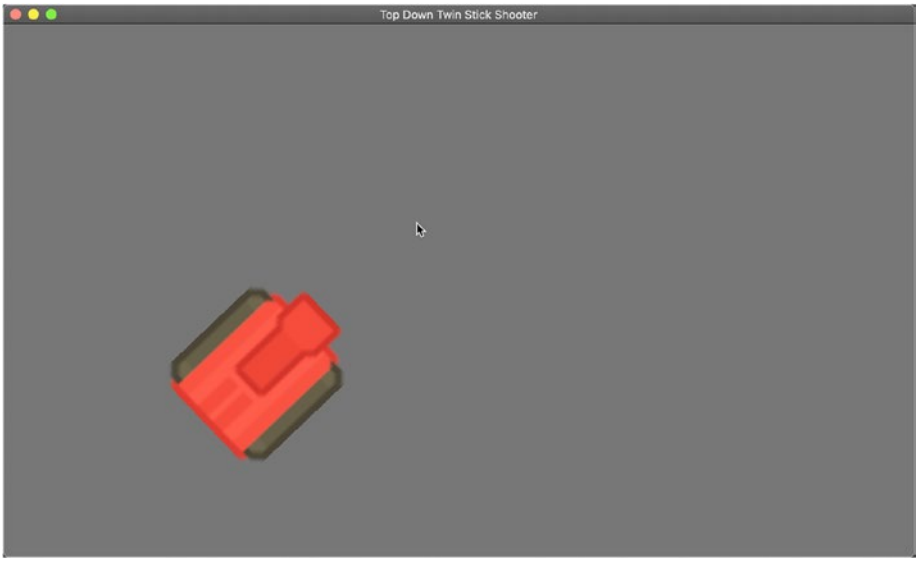

Figure 7-1. *Controlling an Object's Look Direction*

In short, we want to adjust an object's rotation to face the mouse cursor. Listing 7-1 demonstrates how a 2D Node can be rotated to face the cursor at any time.

Listing 7-1. Making an Object Look at the Cursor

```
using Godot;
using System;

public class LookMouse : Sprite
{
    private float StartRot = 0f;
    public override void _Ready()
    {
        StartRot = Rotation;
    }
```

```
  public override void _Process(float delta)
  {
        Vector2 CursorPos = GetLocalMousePosition();

        Rotation += CursorPos.Angle() + StartRot;
        Rotation = Mathf.Wrap(Rotation, Mathf.Deg2Rad(-360),
        Mathf.Deg2Rad(360));
  }
}
```

Listing 7-1 should be attached to any sprite node, and it'll make the node point toward the cursor. This code assumes that an object, by default, is rightward facing. That is, its resting pose will see the character looking toward the right, along the X axis. See Figure 7-2. If that's not the case for your object, then simply use the *RotationDegrees* field from the inspector to rotate the node so that it's rightward facing. The _Process event happens on each frame, and it will call *GetLocalMousePosition* to retrieve the mouse cursor location on screen, relative to the calling node.

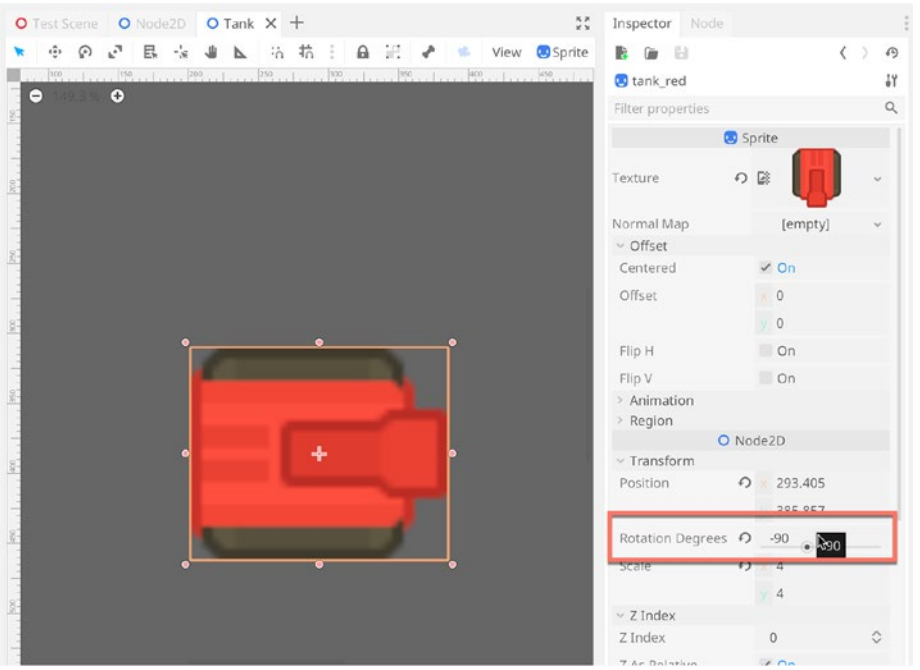

Figure 7-2. *Controlling an Object's Look Direction*

Singletons and Auto-Loading

Sometimes you'll need – and depend upon – behaviors that should exist in every scene. This may include player scores, player inventory, health statistics, and other kinds of global data or functionality that needs to be available almost everywhere. Now, when you're creating objects in different scenes – such as a player scene, or an NPC scene, or an environment scene – it can be tricky to properly test these scenes in isolation from each other without the needed global functionality being present. In our use case here, we'll write some code that simply prints the names of nodes as they're added to the scene for any scene anywhere. If the added node belongs to a specific group, such as "Lava Pit," then we'll connect up its *OnEntered* signal to a function so we can receive notifications about when the player enters the lava pit, presumably to take

damage or be destroyed. Now, this behavior needs to exist in *every scene* and to be present *before* any gameplay nodes are added to the scene, so we can be sure of catching every subsequently added node that could be a lava pit. We can achieve this behavior using *Singletons* configured to *Auto-Load*. Singletons are simply globally accessible classes, of which there can be only one instance simultaneously. The Auto-Load feature of Godot ensures your Singleton classes are added automatically to each and every scene at runtime, and further that'll be added before other scene nodes. Let's start by creating a new empty scene with a single *Node2D* object. See Figure 7-3.

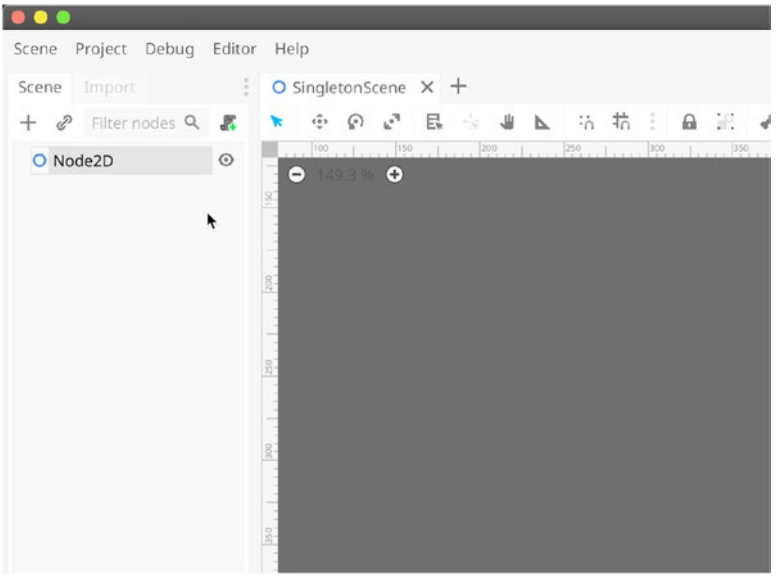

Figure 7-3. *Creating an Empty Scene…*

Next, let's add the following script, as shown in Listing 7-2, and then attach it to the root node of the scene. This code prints the name of each newly added node and also connects its signal to an event handler if the node belongs to the lava pit group.

Listing 7-2. Print the Name of Newly Added Nodes

```
using Godot;
using System;

public class ObjectChecker : Node2D
{
    public override void _EnterTree()
    {
        //Subscribe to the node added signal
        GetTree().Connect("node_added", this, "NodeAdded");
    }

    public void NodeAdded(Node N)
    {
        //Node has been added, print name
        GD.Print("Added node: " + N.Name);

        //Check group membership, and add to lava pit handler
        if needed
        if(N.IsInGroup("LavaPit"))
            N.Connect("OnLavaEntered", this, "OnLavaEnter");
    }

    private void OnLavaEnter()
    {
        //Do damage stuff here
    }
}
```

Now save the scene and access the Project Settings. Choose *Project* ➤ *Settings* from the Application Menu, and access the *AutoLoad* tab. From there, search for and find your SingletonScene, and click the *Add* button to add it to the *AutoLoad* list. See Figure 7-4.

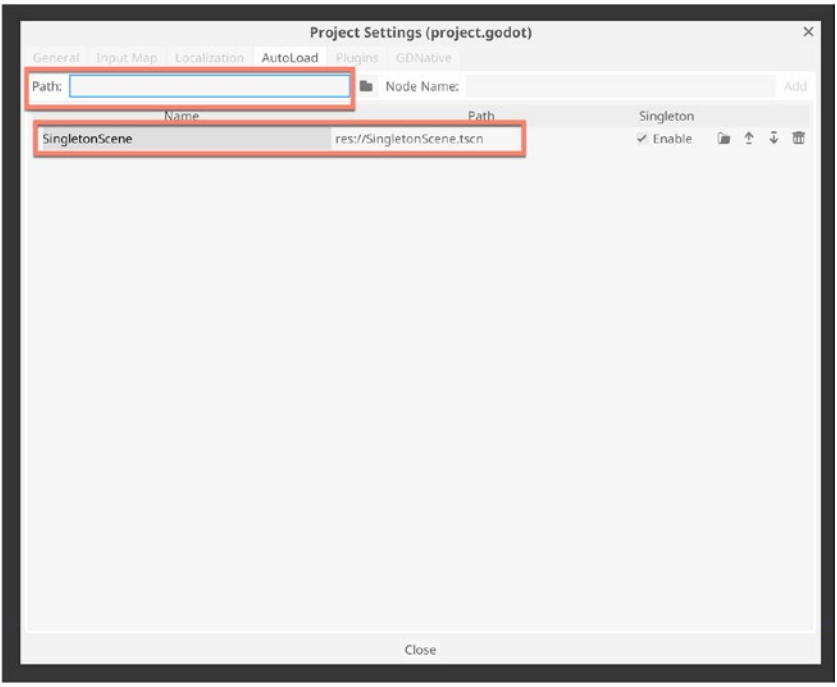

Figure 7-4. *Adding an Autoload Scene…*

Great! Now your Singleton scene will automatically be added to any and every scene at runtime *prior to* any other nodes. This means our code will successfully detect the addition of new nodes, *including* nodes created at scene startup, *because* they are added after the creation of our Singleton.

Batch Renaming

Often, you'll be working with scenes that contain many nodes. Some of these nodes will be manually created and named. But some will be imported from mesh files or auto-generated by add-ons, and these nodes feature default naming. In these instances, you could have a ton of nodes to rename if you want a nice, neat naming convention applied to all nodes in the scene tree. Additionally, you may need to refer to nodes by name using

the *GetNode* or *FindNode* functions, and you'll probably want memorable, meaningful, and structured names for your nodes. You can rename nodes individually by right-clicking a node and choosing Rename from the context menu. However, Godot also features a Batch Rename tool for renaming multiple nodes in one operation. To access this, select multiple nodes in the scene tree, and right-click. The context menu should display a *Batch Rename* option. If it doesn't, you can also access the Batch Rename tool by pressing *Ctrl+F2* on a PC or *Cmd+F2* on a Mac. See Figure 7-5.

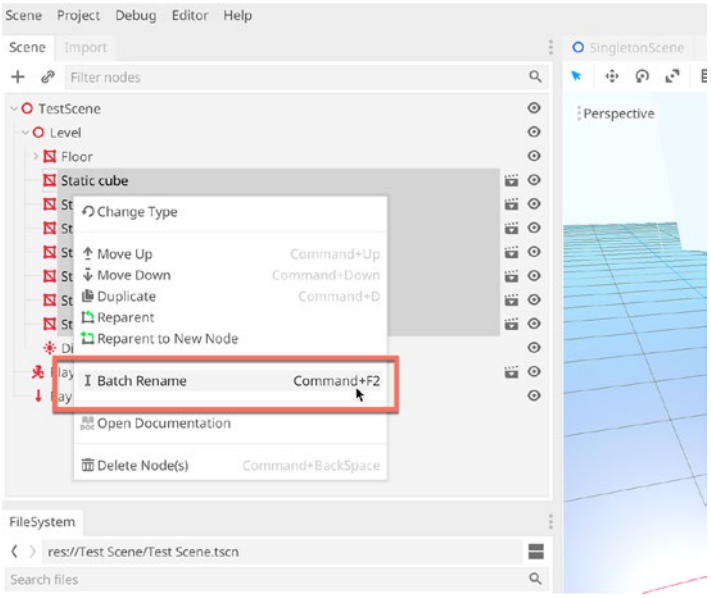

Figure 7-5. *Accessing the Batch Rename Tool*

After accessing the *Batch Rename* tool, you'll be presented with an options menu. This allows you to specify object names, and naming patterns, to search for in the selection and then rename based on specific criteria. See Figure 7-6. In this example, the aim is to rename all objects consistently and to append sequential numbering.

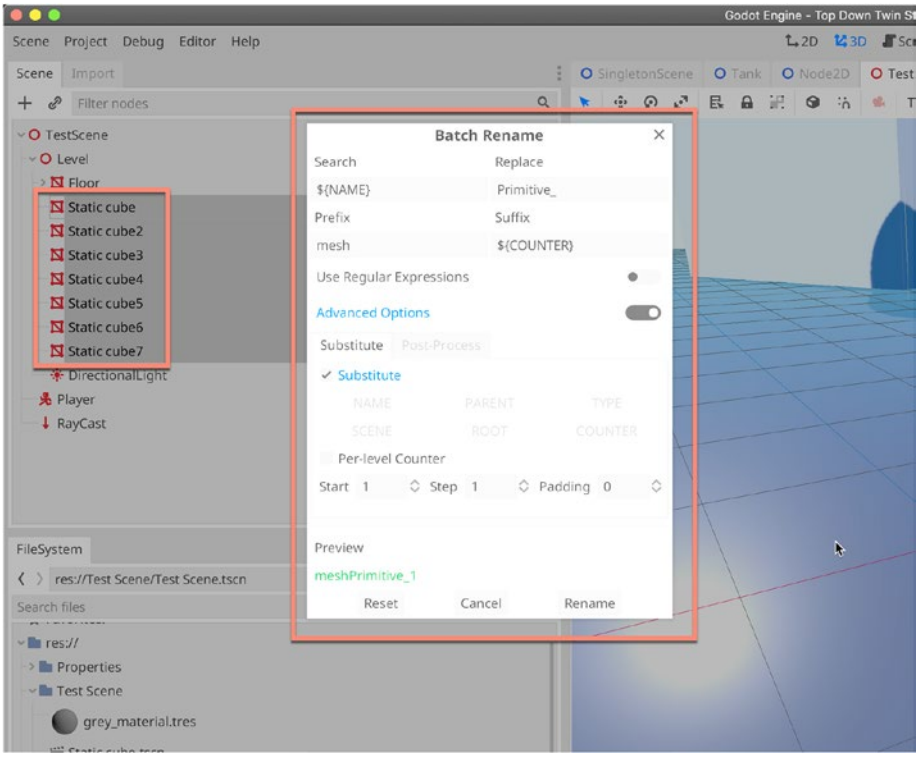

Figure 7-6. *Accessing the Batch Rename Tool*

When you're ready to apply *Batch Rename* to the selected objects, just click the Rename button. When completed in this example, the renamed objects are as shown in Figure 7-7.

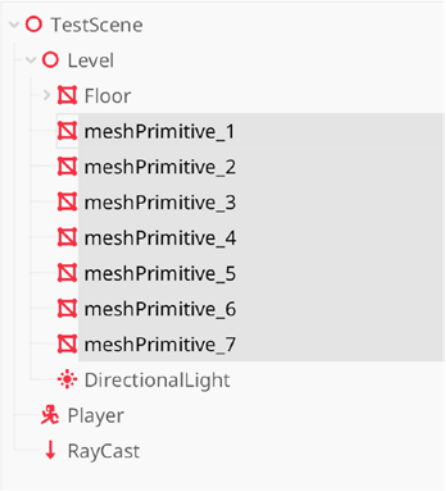

Figure 7-7. *Renamed Objects*

Textures As Masks

Consider Figure 7-8. This shows two separate image files. One is a brick texture, and the other is a simple black and white outline. Sometimes, you'll have two separate textures like this in Godot, and you'll want to use the shape image as a frame or space inside which the other image should display. The outline or shape image is known as the *Mask* – or sometimes the *Stencil*.

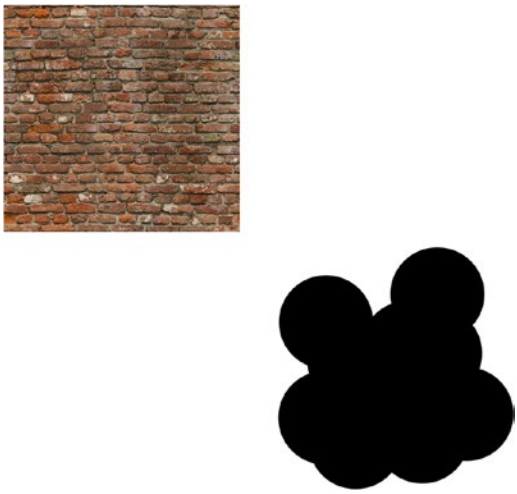

Figure 7-8. *Importing Image Masks. The Black Mask Should Import As White Instead. Black Is Used Here for Clarity*

Let's see how we can create a dynamic mask inside Godot – a mask that can be moved, changed, or updated over time, if we need it to. Start by importing two images into Godot; for our example here, I'll use the brick image and the black and white outline. See Figure 7-9. Then create a new scene and add the brick image as a Sprite texture.

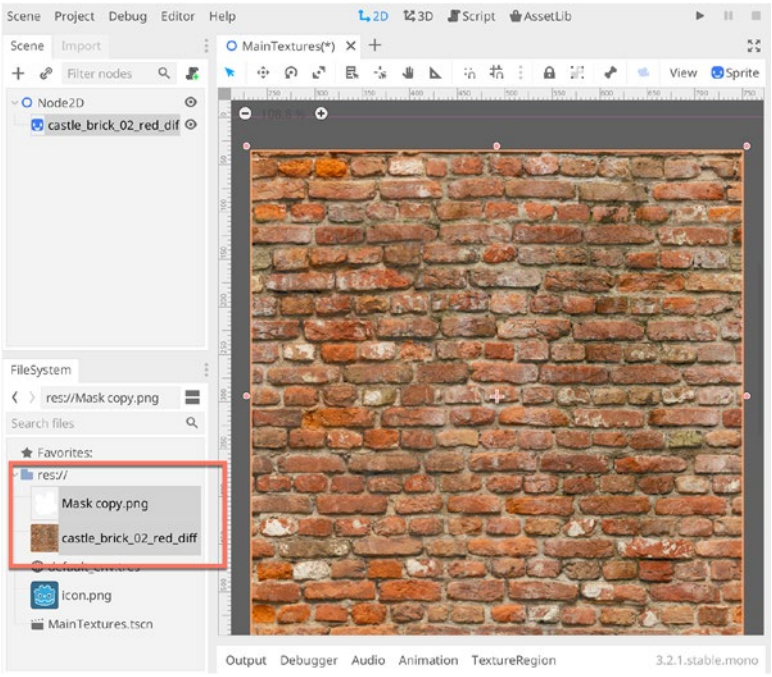

Figure 7-9. *Importing Images into Godot and Setting Up a 2D Scene. Bricks Are Shown Here As a Sprite*

Next, add a 2D light node to the scene. Right-click the root and choose Add Node, and then select Light2D. See Figure 7-10. The act will act like a projector for the mask.

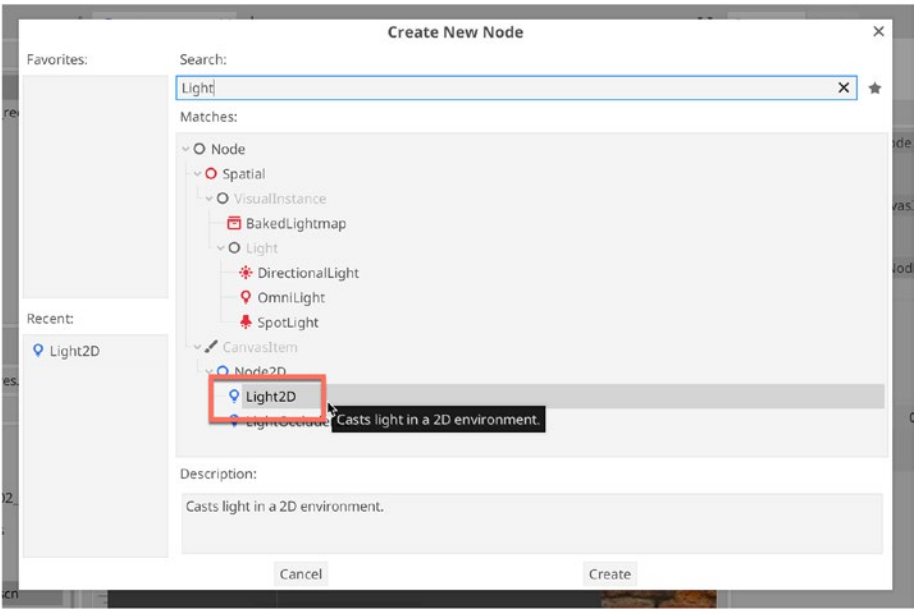

Figure 7-10. *Adding a Light2D to the Scene*

After adding the *Light2D* node, load your mask texture into the *Texture* slot from the Inspector, and then position the light over the brick texture. You'll notice that, by default, the light brightens the texture behind. See Figure 7-11.

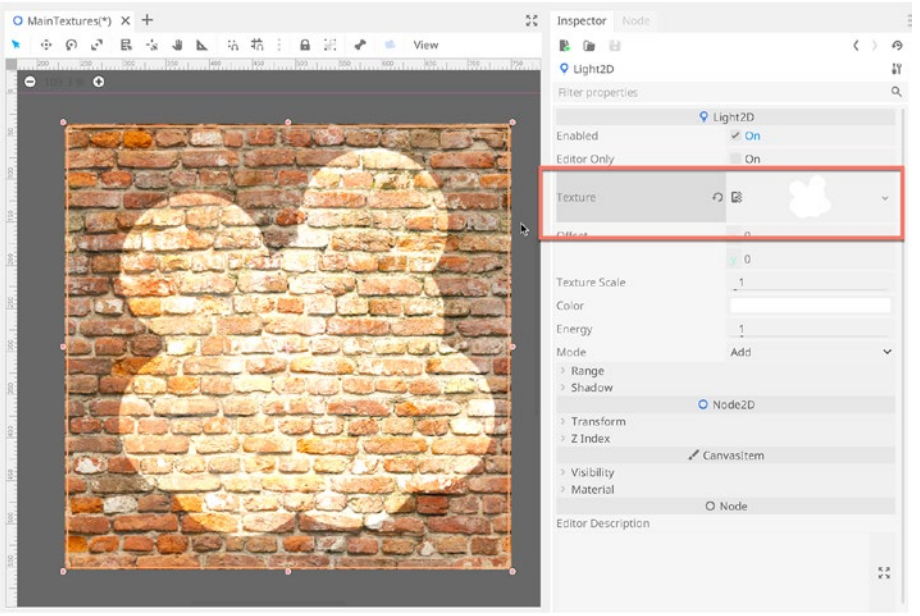

Figure 7-11. *Assign a Texture to a Light2D Node…*

Now select the brick texture in the background, and then add a new *CanvasItemMaterial* from the *Material* slot in the Inspector. This material will control how the texture should interact with intersecting light. See Figure 7-12.

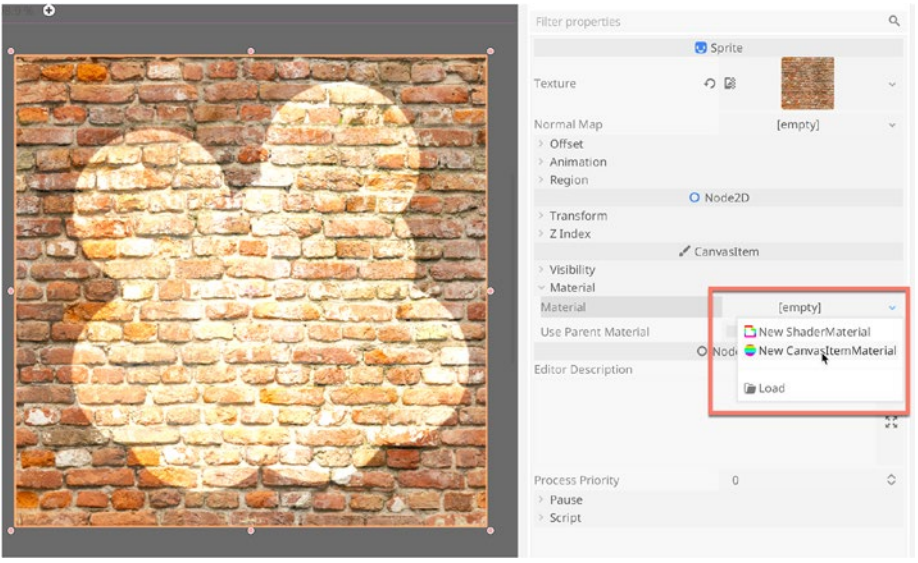

Figure 7-12. *Assigning a New CanvasItemMaterial to the Texture*

After creating the CanvasItemMaterial, change the *Light Mode* to *Light Only*. When you do this, the texture changes showing only the areas within the light texture and effectively creating a mask. However, the pixels still seem slightly washed out. We'll fix this in the next step. See Figure 7-13.

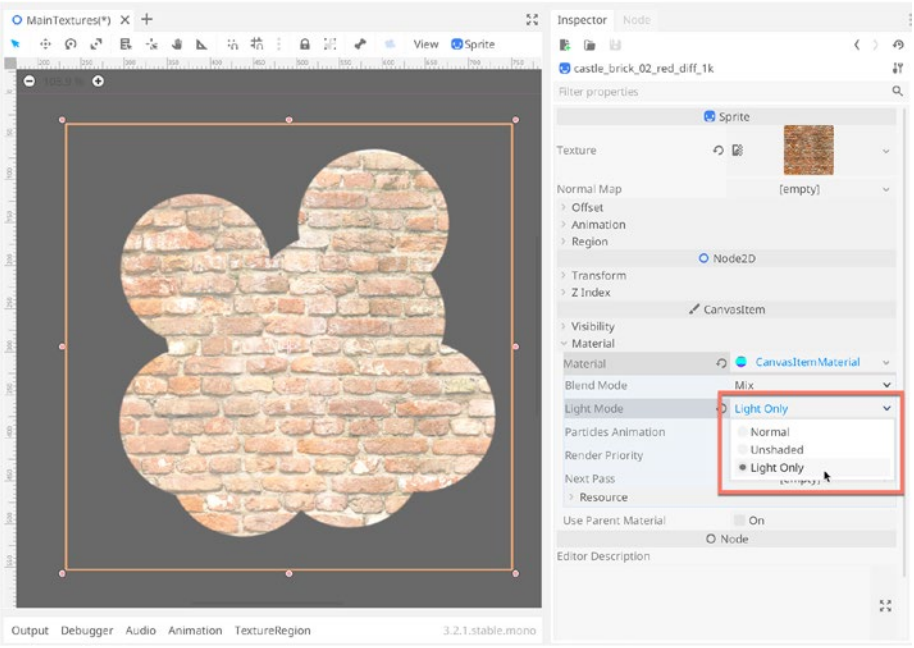

Figure 7-13. *Assigning a Light Only Mode to a Canvas Material Produces a Mask*

We can eliminate the washed out look by selecting the Light2D node and by changing its *Mode* to *Mix*. Now, the pixel saturation will be restored, and the texture behind will continue to be masked even after you move the Light around! Great. You've just created an effective 2D Mask. See Figure 7-14.

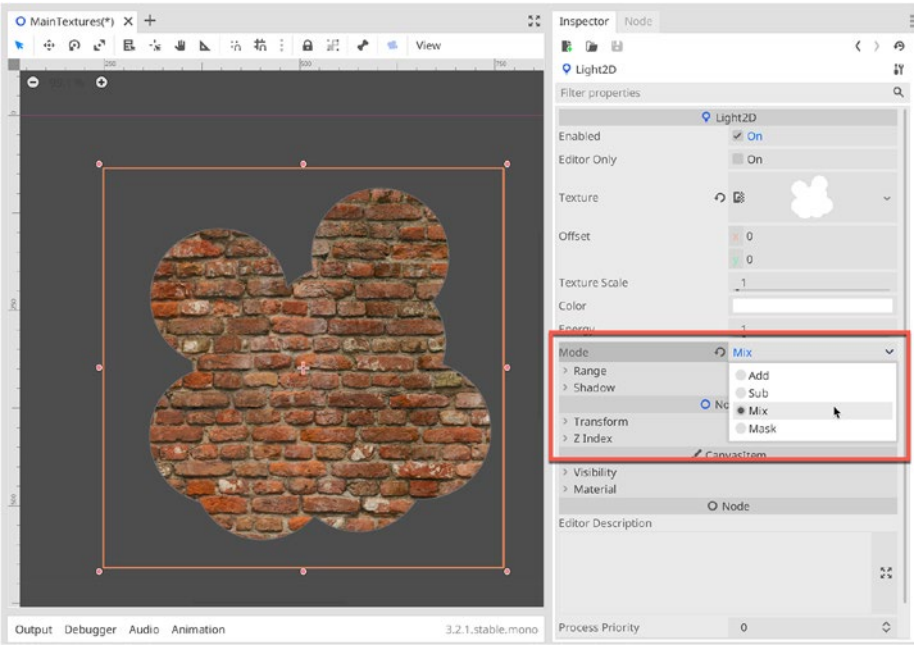

Figure 7-14. *Completing the 2D Mask*

Type-Independent Function Calling

As you develop real-world games with Godot, you'll make lots of classes and functions and properties. As your projects develop in complexity, you'll need to call different functions on different classes, but it can sometimes be difficult or messy to ascertain a Node's type in advance. That's where Godot's *HasMethod* and *Call* functions are useful. These functions are part of the ultimate ancestor *Object* class and so are supported by every object. This means you can try to call a function of a specified name on any object using the following code in Listing 7-3.

Listing 7-3. Calling a Named Function

```
if(MyNode.HasMethod("Explode"))
            MyNode.Call("Explode");
```

Progress Bars and Loading

Large scenes with many assets – such as an RPG town or a village – can take quite a while to load fully. Having your game suspend or hang for long periods is a frustrating experience. Consequently, it makes sense to load larger scenes in a separate thread, or process, to avoid stalling the game entirely. And you may show a progress bar or loading screen to express the loading progress. You can achieve that in Godot easily using the *TextureProgress* node and a simple script. Simply create a new 2D scene and create a *TextureProgress* node. See Figure 7-15.

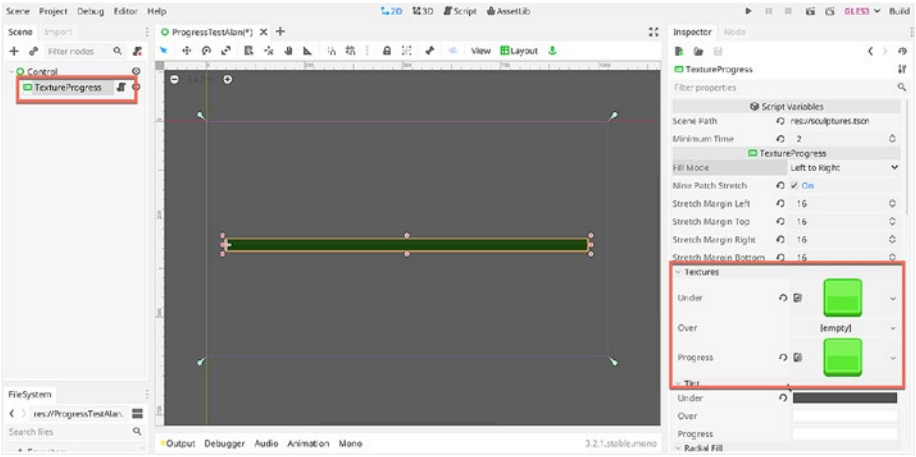

Figure 7-15. *Setting Up a Texture Progress Node…*

Next, attach the following script, as shown in Listing 7-4, to the TextureProgress node. This script will load the specified scene and update its progress in the bar.

Listing 7-4. Level Loading

```
using Godot;
using System;

public class TextureProgressMain : TextureProgress
{
    [Export]
    public string ScenePath;

    [Export]
    public uint MinimumTime = 2000;
    private uint TimeStart = 0;

    private Thread ThisThread = null;

    private void ThreadLoad(string ResPath)
    {
        ResourceInteractiveLoader RIL = ResourceLoader.
        LoadInteractive(ResPath);
        if(RIL==null)return;

        int StageCount = 0;

        while(true)
        {
            uint TimeElapsed = OS.GetTicksMsec() - TimeStart;
            float TimeProgress = (float)TimeElapsed/(float)
            (MinimumTime*1000f)*(float)MaxValue;

            if(StageCount < RIL.GetStageCount())
                StageCount = RIL.GetStage();

            float LoadProgress = (float)StageCount/(float)RIL.
            GetStageCount()*(float)MaxValue;
            CallDeferred("set_value", (TimeProgress<LoadProgress)
            ?TimeProgress:LoadProgress);
```

```
            //Take a break
            OS.DelayMsec(100);

            if(Value>=MaxValue)
            {
                CallDeferred("ThreadDone", RIL.GetResource() as
                PackedScene);
                return;
            }

            if(StageCount >= RIL.GetStageCount())
                continue;

            //Poll current stats
            Error PollData = RIL.Poll();

            if(PollData == Error.FileEof)
            {
                StageCount = RIL.GetStageCount();
                continue;
            }

            if(PollData != Error.Ok)
            {
                GD.Print("Error Loading");
                return;
            }
        }
    }

    private void ThreadDone(PackedScene R)
    {
        ThisThread.WaitToFinish();
        SceneTree ST = GetTree();
```

```
        Node Root = R.Instance();

        ST.CurrentScene.Free();
        ST.CurrentScene = null;
        ST.Root.AddChild(Root);
        ST.CurrentScene = Root;
    }

    public override void _Ready()
    {
        TimeStart = OS.GetTicksMsec();
        ThisThread = new Thread();
        ThisThread.Start(this, "ThreadLoad", ScenePath);
    }
}
```

When this script is attached to the TextureProgress node, several parameters are exposed from the Inspector, as shown in Figure 7-16.

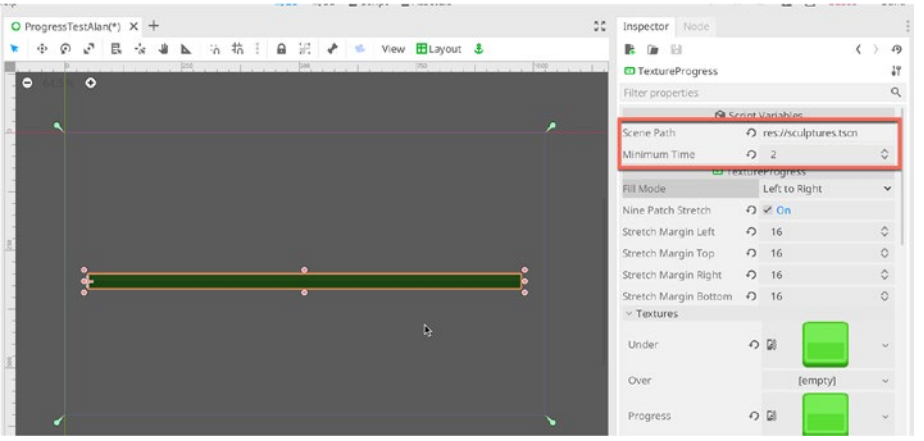

Figure 7-16. *Controlling Scene Loading*

This script accepts two parameters. The first is *Scene Path*, which is the resource path to the scene, which should be loaded. The second is *Minimum Time*. This specifies a minimum amount of time, in seconds, for which the loading bar will show. In cases where a scene loads incredibly fast, this will prevent the loading screen from simply flickering into view briefly and then disappearing. The progress bar will represent either the loading progress of the scene or the progress of the minimum time, whichever is the slowest.

How to Save Game States

Save states let gamers retain their progress, resuming from earlier sessions. With save states, your game remembers how much progress you've made. There are many different ways to code save states, and this section looks at JSON – a text-based language for saving game data easily and quickly. Let's start by considering the following sample scene, shown in Figure 7-17 and included in the book companion files.

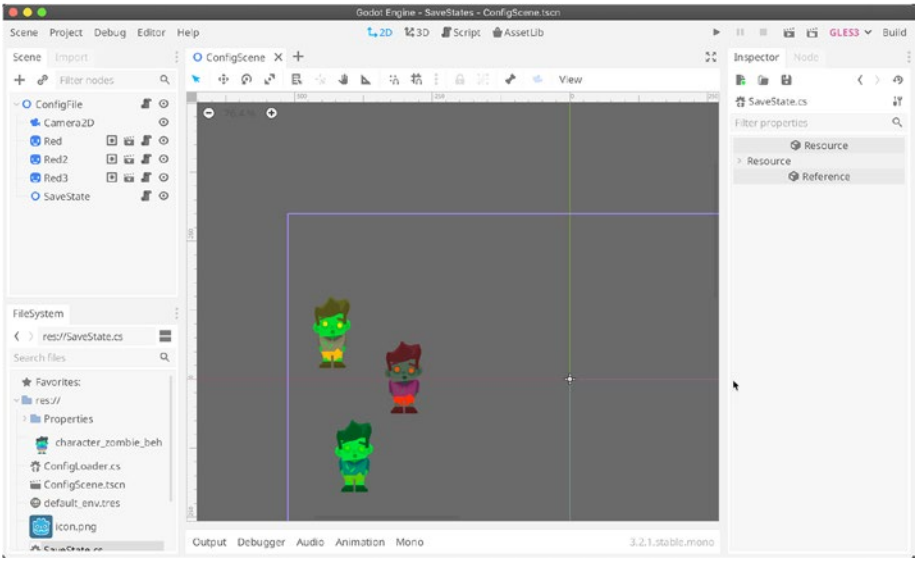

Figure 7-17. *A Sample Scene with Three Zombies*

The sample scene features three zombie characters. We'll save their position and color, allowing it to be restored in later play sessions. You can, of course, save any data you want. To get started, we'll need to create two script files: one to be attached to each object that can be saved and the other which is a global script that manages the save and load process. First, select all objects to be saved – the zombies in our example – and add them to the same group. Here, I've created a group called *Persistent*. See Figure 7-18.

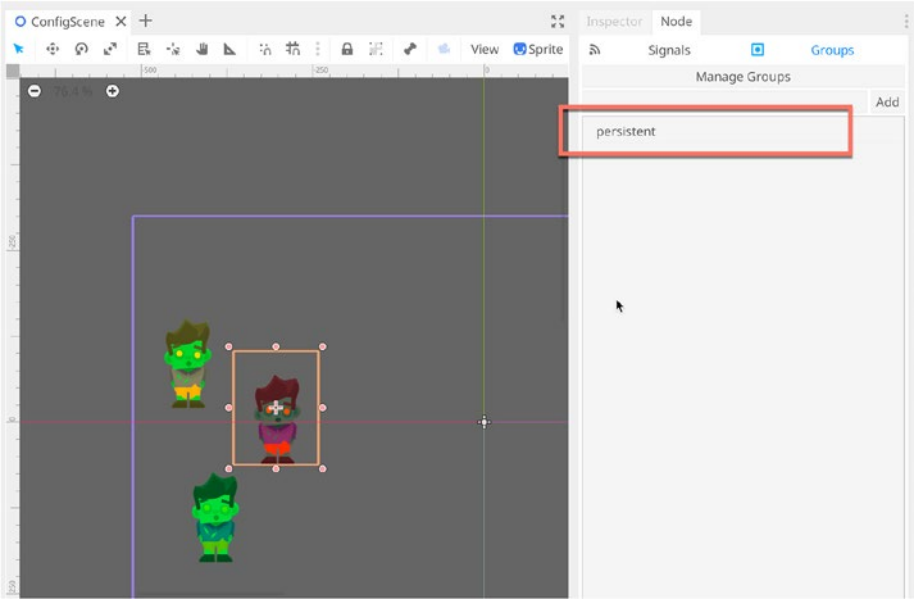

Figure 7-18. *Adding Zombies to a Persistent Group*

Next, the following script should be attached to each sprite to be saved. This script file is intended to be called, or invoked, by the Save Game Manager. It serves only two purposes. First, it converts its properties – like position and color – into a JSON string for Save operations. And second, it converts a JSON string back into properties for Load operations. These two critical processes effectively convert an object to and from JSON. See Listing 7-5.

Listing 7-5. Serializing a Sprite Object

```
using Godot;
using System;

public class SerializeNode : Sprite
{
    public Godot.Collections.Dictionary<string, object> Save()
    {
        return new Godot.Collections.Dictionary<string,
        object>()
        {
            {"Filename", Filename},
            {"Parent", GetParent().GetPath()},
            {"PositionX", Position.x},
            {"PositionY", Position.y},
            {"ModColor", Modulate.ToHtml(true)}
        };
    }

    public void Load(Godot.Collections.Dictionary<string,
    object> SavedData)
    {
        Position = new Vector2((float)
        SavedData["PositionX"],(float)SavedData["PositionY"]);
        Modulate = new Color((string)SavedData["ModColor"]);
    }
}
```

Next, create a *Node2D* object, and attach a new script (*SaveState*), which can be used to invoke saving and loading behavior for each object. See Listing 7-6.

Listing 7-6. Save States

```
using Godot;
using System;

public class SaveState : Node
{
    //Name of file for saving and loading
    public string SGDName = "GameData.sav";
    public void Save()
    {
        //Open file for writing
        File SaveFile = new File();
        SaveFile.Open("user://"+SGDName, File.ModeFlags.Write);

        //Find all objects in scene to be saved
        Godot.Collections.Array Nodes = GetTree().
        GetNodesInGroup("persistent");

        //For each object, get its JSON representation
        foreach(Node N in Nodes)
        {
            if(N.Filename.Empty())
                continue;

            if(!N.HasMethod("Save"))
                continue;

            //Save JSON to file
            var SaveData = N.Call("Save");
            SaveFile.StoreLine(JSON.Print(SaveData));
        }

        //Close file
        SaveFile.Close();
    }
```

```
    public void Load()
    {
        File SaveFile = new File();

        if(!SaveFile.FileExists("user://"+SGDName))
            return;

        //Open file for reading data
        SaveFile.Open("user://"+SGDName, File.ModeFlags.Read);

        //Remove any duplicate objects already in scene
        Godot.Collections.Array Nodes = GetTree().
        GetNodesInGroup("persistent");
        foreach(Node N in Nodes)
            N.QueueFree();

        //Loop through file
        while(SaveFile.GetPosition() < SaveFile.GetLen())
        {
            string Line = SaveFile.GetLine();

            if(Line.Empty())break;

            Godot.Collections.Dictionary NodeData = (Godot.
            Collections.Dictionary)JSON.Parse(Line).Result;

            //Load objects back into scene
            PackedScene PS = (PackedScene)ResourceLoader.
            Load(NodeData["Filename"].ToString());
            Node NewScene = PS.Instance();
            Node ParentNode = GetNode(NodeData["Parent"].
            ToString())as Node;
            ParentNode.AddChild(NewScene);
```

```
            if(NewScene.HasMethod("Load"))
                NewScene.Call("Load", NodeData);
        }

        //Close file
        SaveFile.Close();
    }

    //Test functionality. Can be removed
    public override void _UnhandledInput(InputEvent @event)
    {
        if (@event is InputEventKey eventKey)
        {
            //Pressing S will save
            if (eventKey.Pressed && eventKey.Scancode == (int)
            KeyList.S)
            {
                Save();
                return;
            }

            //Pressing L will load
            if (eventKey.Pressed && eventKey.Scancode == (int)
            KeyList.L)
            {
                Load();
                return;
            }
        }
    }
}
```

Excellent. Together, these two scripts support load and saving behavior. Be sure to check out the project included in the book companion files.

Summary

Congratulations! You've now completed the final chapter and the book. This chapter presented a selection of super-handy tips and tricks for achieving common gameplay behaviors in Godot using C#.

Index

A. Thorn, *Moving from Unity to Godot*, https://doi.org/10.1007/978-1-4842-5908-5

G

H

I, J

K

L

M

N

O

P, Q, R

S

T

U, V

W, X, Y, Z