FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Turn based role playing game in Godot engine
Student:	Minh Hieu Ta
Supervisor:	Ing. Jan Matoušek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

There are not many finished open-sourced quality games nor tutorials in Godot 4 engine using C# language for their scripting. The project should demonstrate correct use of programming and design patterns for this choice of engine and programming language on a sample of turn based role playing game in Godot 4 engine.

Instructions:
1) Analyze at least 5 game engines and programming/scripting languages used with them, focus on their capabilities, documentation, tutorials and ease of use.
2) Present a project of turn based role playing game to be used as an example of using C# language in Godot 4 engine.
3) Analyze game design patterns and techniques that could be used in a Godot 4/C# game.
4) For the project, design game architecture with focus on proper use of programming and design patterns. Try to fully exploit features of the engine.
5) Implement the game using designed architecture and perform thorough tests.
6) Write a tutorial or tutorial series covering practices and techniques used in the project.

Electronically approved by Ing. Michal Valenta, Ph.D. on 14 February 2024 in Prague.

Bachelor's thesis

TURN BASED ROLE PLAYING GAME IN GODOT ENGINE

Minh Hieu Ta

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jan Matoušek
May 16, 2024

Contents

List of Figures

List of code listings

Declaration

In Prague on May 16, 2024

Abstract

This bachelor's thesis deals with the implementation of a turn-based game in the Godot Engine 4.2 with the use of C# as the main programming language. The thesis follows the development process from conception to design and implementation. The result of the work is an example project to be used as a learning resource for people with basic-to-intermediate knowledge of programming, but no experience with game development.

Keywords game development, video game, turn-based game, role-playing game, example project, tutorial, Godot Engine, C#

Abstrakt

Tato bakalářská práce se zabývá implementací tahové hry v Godot Engine 4.2 s užitím C# jako hlavní programovací jazyk. Práce pokrývá proces vývoje od konceptu, k návrhu, až po implementaci. Výsledkem je ukázkový projekt pro užití jako výukový materiál pro jednotlivce se základními až středními znalostmi v programování, ale žádnými zkušenostmi s vývojem her.

Klíčová slova herní vývoj, počítačová hra, tahová hra, role-playing hra, ukázkový projekt, tutorial, Godot Engine, C#

Terminology

CSV	Comma seperated value
Scene	A Node tree forming one cohesive logical unit
SceneTree	The main Node tree from which an entire Godot game stems
Signal	Godot's implementation of the Observer pattern
Node	The basic building block in Godot
Resource	A data container in Godot, commonly refers to user-defined custom Resources

Introduction

The video game industry has grown immensely since its inception, creating a massive market of games being released every year, from big AAA productions, to many smaller indie games and hobbyist projects.

Video games have also become increasingly more complex, being put up to higher quality standards, requiring many different technologies and tools, while still expected to take a similar amount of time to develop.

One consequence of this has been the rise in the popularity of game engines, which are tools designed for game development that typically feature finished implementations for graphics rendering, sound and physics, amongst other things. Originally more commonly created as internal tools, there are now many pre-made game engines available on the market, which are particularly appealing as they have already finished a significant amount of the low-level work, letting the game developers focus on creating the games themselves, rather than focusing on implementation details.

There are many popular engines on the market, the industry standard Unity and Unreal Engine, to engines popular with hobbyists and smaller studios such as Godot, GameMaker and Construct. Many of these engines boast similar feature sets, however, where they can significantly differ is in the workflow, ease of use, and availability of learning materials and tutorials.

Godot is an open-source game engine that has been relatively obscure until the release of Godot 4, which saw improved 3D capabilities after only being considered viable for 2D development before. While Godot features its own scripting language known as GDScript, it also provides support for custom bindings of other languages such as C++ and Rust through a technology called GDExtension. It also has official support for C#, which saw many improvements with the release of Godot 4.

However, despite this, Godot remains somewhat niche and its availability of tutorials and learning material is far smaller compared to some of the more widely used game engines such as Unity. On top of that, most tutorials for Godot are focused on GDScript, with C# mostly considered as an alternate second option.

The main goal of this thesis is to create a project in the Godot 4 game engine using C# as the main programming language, with the intent of letting it be an example project meant to showcase features, common techniques and architecture design commonly used for projects made in the engine. In the end, the goal is to familiarize anyone interested in game development with Godot, its constructs and concepts.

This thesis does not aim to create a fully-featured or complete game, but rather a playable prototype with finished implementations of more essential aspects of the game. There is not much focus on game design, the creative process of designing a game, user experience aspects such as UI design, ”quality of life” features, or intuitiveness and teaching the player about the game’s systems, user testing and feedback in regards to improving and tweaking game systems and balance.

The “Analysis” chapter contains an analysis of 5 game engines, with particular focus on Godot. Each engine analysis will consist of a short summary, the features, structure and workflow employed by the engine, documentation and learning material availability, and finally a few example games.

The “Turn-based Game Project” chapter talks about the game concept itself. From the artistic aspects such as inspirations and choice of genre, to the design such as its systems and gameplay.

The “Godot Basics and Techniques” chapter will discuss proper techniques and patterns commonly used in Godot projects, and the features of the engine that facilitate and encourage the use of these techniques.

The “Design Architecture of the Game” chapter describes how the game’s concepts and ideas introduced in the “Turn-based Game Project” chapter are translated into structures inside the actual game engine, and which specific engine features were used to implement each aspect.

The “Implementing the Game” chapter, will go over the game development process in more specific detail, in a form similar to a tutorial. This chapter includes many code examples and functions used in the project, as well as the decision-making process behind each implementation choice.

Chapter 1

Analysis of Game Engines

In recent years, the amount of publicly available game engines has increased. There are now many game engines with differing workflow, design philosophies, advantages and disadvantages to choose from. To better understand what makes Godot unique, and how its features differ from or correspond to other game engines available on the market, this chapter will contain an analysis 4 other game engines: Unity, Construct, GameMaker and PICO-8.

This selection was made based on the statistics of the most popular game engines on the video game hosting and selling website itch.io, as well as a few other factors.

Table 1.1 Game Engine use statistics on itch.io (10/03/2024)

Engine	Games on itch.io	% of Total
Unity	114,374	45.5%
Construct	36,256	14.4%
Godot	24,205	9.6%
GameMaker	17,241	6.9%
Twine	14,294	5.7%
bitsy	8,268	3.3%
Unreal Engine	7,242	2.9%
PICO-8	6,568	2.6%
RPG Maker	5,656	2.2%
Ren'Py	3,493	1.4%
Other	13,891	5.5%
Total	251,488	

These statistics were obtained from itch.io's "Most used Engines" page. [1] There are two things to note regarding these statistics:

- It is not mandatory to mark which game engine your game uses, and as such, not every game on itch.io is included in these statistics. At the time the data was obtained (10/03/2024), itch.io currently has 914,042 projects under the Games category, meaning only about 27.5% of games on itch.io are tagged with which engine they use.
- itch.io is not indicative of trends in the larger video game industry, as it is mostly used by hobbyists and indie developers.

As you can see from the table 1.1, Unity is by far the most popular at 45.5%, with the second most used being Construct at only 14.4%. Behind that is Godot and lastly GameMaker. Among these is the unusual self-proclaimed ”fantasy console” PICO-8, which offers a very unique and niche environment for game development, making an interesting option for analysis among these other more conventional game engines.

In the video game industry, Unreal Engine is widely used. However note that it places 7th on the itch.io popularity. This is due to the design and workflow being largely optimized for AAA production, meaning it doesn't see much use in hobbyist spaces and small productions. It is also only capable of creating 3D games, not 2D.

As such, Unreal Engine will not be among the analysed game engines, as Godot is more suited for smaller projects with smaller teams. The analysed engines are all used in hobbyist spaces, as seen by their usage on itch.io, and are therefore expected to employ more similar workflows and design philosophies to Godot.

In the following sections, these game engines will be analysed in the following order: Unity, Construct, GameMaker PICO-8 and finally Godot.

1.1 Unity

Unity is a proprietary engine that popular among hobbyists as well as in professional settings as the industry standard. The engine supports both 2D and 3D games,with support for AR and VR as well, and has been used in both games and films.

It uses C# as its scripting language, using which one can script both in-game logic, as well as the editor itself, which allows it to be extended for use with custom classes and data types.

It can export to Linux, Windows, macOS, iOS, Android, HTML, as well as various consoles like XBox One, Playstation 4/5 and Nintendo Switch with the proper platform modules installed.

1.1.1 Structure

Unity projects begin with the Scene, which where all the various game objects are stored. In that sense, a Scene is a level that contains various things like environments, characters and UI elements. Scenes are then saved as assets that can be loaded and swapped during runtime.

GameObjects are the basic building block in Unity. They are the objects placed inside Scenes, ranging from characters, items, to cameras and lights.

GameObjects consist of several “Component”s, which represent various properties and functions, such as a component for 2D space coordinates, 2D sprite displaying, 3D mesh rendering, etc. By attaching various Components each handling a specific function, a GameObject's behavior can be defined and extended. This design favors composition of inheritance.

GameObjects can then be saved as an asset in the project files, called a Prefab. A Prefab stores the data of the GameObject, such as its components, the property values, etc. When creating Prefabs, several Prefabs can also be nested together, i.e. adding Prefabs as children of other prefabs. Prefabs can then be placed inside scenes and their properties can be changed, making them act as sorrt of templates. Prefabs can also be used to instantiate new GameObjects during runtime.

Scripting is done by attaching a script to a GameObject as a Component. This script inherits from Unity's MonoBehaviour class, which facilitates various Unity features. Scripts can then implement behavior by accessing the GameObject's various Components and calling their methods or changing their properties.

1.1.2 Documentation and Learning Materials

Unity has online documentation, which contains a manual that goes over various aspects and features of the engine, as well as a Scripting API documentation that contains information on the API provided by the engine, that is: the various classes and structures, their properties and methods, alongside example code.

Being one of the most used game engines, there are also many tutorials available from both official and unofficial sources. Through the dedicated “Unity Learn” website, Unity offers many standalone tutorials for various features of the engine, as well as full courses aimed at essentials, programmers, and areas such as VR and AR development.

Unity also has a notably large Asset Store, with has both paid and free 2D and 3D assets, audio, various tools and plugins, even full implementations such as the official “FirstPerson” controller asset, which implements first person movement. Through the use of assets like these, many beginners can create games with very little coding required.

1.1.3 Games made in Unity

- Beat Saber
- Hollow Knight
- Pokémon GO

1.2 Construct

Construct is a HTML5-based game engine for 2D games, with support for some basic 3D elements. While it was originally released under the GPL license, it later switched to a proprietary license.

It is aimed primarily at non-programmers, allowing quick and intuitive creation of games through visual programming. Aside from that, the engine also supports scripting using JavaScript.

The editor itself is browser-based, making it very accessible on most modern hardware. However, there is no offline alternative, meaning everything needs to run from the web.

Internally, the engine is only capable of exporting to HTML5, meaning exports for iOS, Android, Windows, Linux, macOS and Xbox are facilitated through web wrappers.

1.2.1 Structure

As mentioned, the engine workflow is designed primarily for the use of its visual programming system. As such, all of the following elements are manipulated in the editor through visual programming blocks and menus.

Layouts are scenes, which contain various pre-arranged Objects laid out by the user. They serve the purpose of storing levels, menus and title screens.

An Object Type is the equivalent of a class in conventional systems. When creating a new Object Type, the user selects from a list of plugins which define the base function and behavior of the object. These plugins span many options ranging from common game objects such as sprites, images, text, to data containers like Array, JSON, XML, as well has input reading objects like Gamepad, Mouse and Keyboard.

Objects can then be further extended by assigning them Behaviors. These range from things like movement such as 8-directional movement, standard platformer movement, tile movement, to others such as tweens, camera focus and wrap.

Events are how the engine facilitates interactions between objects and other various game logic. You can define Events inside an Event sheet which is then set to a specific Layout. An Event comprises of a trigger, which are conditions under which the event is called, and then the actions that will be taken.

1.2.2 Documentation and Learning Materials

Since it is geared towards non-programmers, there is a lot of effort to make the engine as beginner-friendly as possible.

The engine itself features ”Guided Tours” which run directly in the editor and walk the user through the steps of creating an example game, or showcasing a particular feature of the engine. These are particularly user-friendly as they follow every step of the process directly in the editor.

There engine has online documentation containing articles ranging from “Getting Started”, tips and guides, to refence on the various Behavior and plugins.

There is also a dedicated “Resources” tab on the official website, which contains written and video tutorials from both official and non-official sources.

Construct also has an official asset store, which contains things such as image and sprite assets, as well as plugins and templates that feature pre-made implementations of systems such as inventory or multiplayer systems.

1.2.3 Games made in Construct

- Small Saga
- There Is No Game (game jam version)

1.3 GameMaker

GameMaker is a game engine primarily used for 2D games, although 3D support is available.

Similar to Construct, GameMaker is marketed as a beginner-friendly game engine with features like visual programming. However, GameMaker also has its own scripting language known as Game Maker Language (GML), which is an imperative, dynamically typed language.

The engine features a built-in image editor for creating sprites, as well as frame by frame animation. As the engine is primarily used for 2D development, it also features inherent support for tilesets, used to build levels.

GameMaker can export to Linux, WIndows, macOS, iOS, Android HTML, as well as various consoles from Xbox One, Playstation 4/5 and Nintendo Switch.

1.3.1 Structure

GameMaker projects consist of various types of data, ranging from sprites, objects, rooms, to sounds and more. These assets are all categorized and contained in the Resources window.

“Rooms” contain the layout of various objects on laid out on a grid space and serve the purpose of defining levels.

Objects work similarly to other game engines: they define a set of properties and behavior, that can then be instanced in levels in the editor during runtime. Objects can be assigned a sprite and displayed. Game logic and object behavior is implemented through Events.

Events are logic attached to Objects that dictate its behavior whenever its trigger is activated. Triggers can be things such as button presses, collisions, timers, or the “step” trigger, which calls the event on each frame. Events can be implemented using visual programming, or through scripting using GML.

1.3.2 Documentation and Learning Materials

GameMaker has an online manual that contains articles on the various features of the engine, as well as introductions and quick start guides which go over basic operations in the editor. The manual also contains documentation on both the engine's GML scripting language, as well as the visual scripting method.

The official website for the engine also features a "Tutorials" tab, which contains official tutorials that go through the process of implementing simple example projects, with tags identifying which tutorials make use of the coding or visual scripting methods.

The start page when launching the engine also contains a link to the Tutorials page, as well a section displaying various resources such as recommended beginner tutorial projects or video tutorials.

Despite having support for visual scripting, many tutorials for GameMaker tend to recommend the use of the scripting language over the visual method.

1.3.3 Games made in GameMaker

- UNDERTALE
- Wandersong

1.4 PICO-8

Out of the 5 game engines analysed in this chapter, PICO-8 is perhaps the most unconventional one. Self-described as a "fantasy console", PICO-8 is a virtual machine meant to mimic the limited hardware of old 8-bit systems. In doing this, the graphical, audio and performance limitations are meant to encourage creativity and uniqueness.

The virtual machine, and therefore any game developed for it, has a 128x128 display resolution with 16 colors. PICO-8 programs are written using Lua syntax, however it does not include the Lua standard library or the ability to import any other modules, instead offering an API with PICO-8 specific functions.

Games made in the PICO-8 can be exported as cartridges. These cartridges contain all of the data of the game, meaning distributed cartridges can be opened in the virtual machine to see all of its project files. Other than cartridge exports which are meant to be loaded into and run in the virtual machine itself, the engine also allows stand-alone exports to Linux, Windows, macOS and HTML.

1.4.1 Structure

The entire process of the development takes place in the PICO-8 virtual machine, meaning the machine contains all the tools necessary for game devlopment, ranging from a script editors, to sprite image and audio editors.

The virtual machine contains a commandline that functions much like a standard command line, it can be used to navigate the filesystem, create directories, save, load and run games.

All code in PICO-8 games is contained in a single script edit in this editor. Unlike other game engines, there is no concept of objects, as all game logic, sprite rendering and input happens in this single script file. Another intentional limitation of the PICO-8 is that it allows 8192 tokens at most in its code. A token is a literal value, string, variable or operator, bracket or a keyword.

The image editor contains all of the sprites to be used in the game, placed in one single 128x128 spritesheet. It has basic drawing tools such as a Draw, Stamp, Select, Fill and Shape, which consists of oval, rectangle and line. You can also import .png files.

The map editor is then used to define the layout of levels in the game. It is a 128x64 grid of 8x8 tiles. Similar to the spritesheet, all of the maps are contained in a single block, meaning all of the levels are laid out together here, with only the appropriate chunk being loaded during a game.

Finally the virtual machine contains an SFX and Music editor, where the user can create up to 64 sound effects by defining the sound by setting the frequency and volume of several sequential notes. Using the created SFX, Music can then be arranged.

1.4.2 Documentation and Learning Materials

Because of the nature of the PICO-8, it is intended for hobbyists and small projects, rather than full-scale releases. Working in the PICO-8 is very different from most modern software development approaches due to the intentional limitations imposed by the virtual machine, which can make it unintuitive and difficult to use for beginners.

The engine has online documentation, which is a single page manual that contains a getting started section which describes basic navigation and use of the virtual machine such as commandline commands and keyboard shortcuts. It also contains documentation on the various tools described in the earlier Structure sections, amd finally a brief overview of Lua syntax and the various functions that are part of the API of the virtual machine.

The website also has a “Resources” section, which contains links to the online documentation, as well as several other tutorials, videos and articles made by others.

A large source of learning material is to be found in the cartridges for the games themselves, as they can be simply loaded in the virtual machine to see all of the project files, including code, sprites, map layouts and sounds. The engine’s website has a dedicated section for sharing game cartridges.

1.4.3 Games made in PICO-8

- Celeste Classic
- POOM

1.5 Godot

Godot is an open-source game engine that supports 2D and 3D. The editor is available on desktop platforms such as Linux, Windows and macOS, as well as on mobile Android devices, and through the web using HTML.

It supports scripting using its own custom language: GDScript, which uses python-like syntax. It also has C# support. A project can even mix and match GDScript and C# code in a project, and even instantiate and access data between the languages. This is however not used anywhere in the example project. You can read more about cross-language scripting through its page in the Godot documentation. [2]

Aside from GDScript and C#, support for other languages can be achieved using “GDExtension”, which can be used to create bindings to any language. There is an official binding for C++, as well as third-party bindings for other languages such as Rust, Go and Swift. [3]

Godot can export to Windows, Linux, macOS, iOS, Android and HTML, although at the time of writing (Godot 4.2.1), when using the C# builds of the engine, HTML exports are unavailable and iOS and Android exports are marked as experimental with limitations.

1.5.1 Structure

Godot uses a very inheritance-heavy design. Its basic building blocks: Nodes use inheritance to accumulate properties and methods from other Nodes. These Nodes are then used to construct tree-based structures using child Nodes and parent Nodes, which resembles a more composition-oriented design.

In Godot, objects and levels are not differentiated. That is: both objects in a level, and the level layout itself are simply a collection of Nodes assembled in a Node tree. This design takes advantage of the flexible and multi-faceted nature of Nodes, which there are many of, each handling a specific function such as displaying sprites, playing audio, playing animations, UI elements such as buttons and scrollbars, etc.

Scripting is done by attaching scripts to Nodes. Scripts attached to Nodes inherit from its type, which gives the script access to the Node's properties and methods. By making use of these methods and properties, game logic can be implemented. From the script, it is also possible to traverse the Node tree and access other Nodes, change the structure of the Node tree and perform operations with them.

Node trees can then be saved as assets: Scenes, to be instantiated later. Scenes can also be inherited to create child Scenes that update accordingly with any changes to the parent Scene. Once again, because Node trees are used for both objects and levels, there is no differentiation between the two.

1.5.2 Documentation and Learning Materials

Godot has an online documentation that is also built in directly into the engine from the same source, allowing for viewing from the editor. The documentation source is also available on GitHub.

The online documentation contains sections such as getting started, which serves as a first introduction to working with the engine, a manual, which describes the many features available in the engine and how to work with them, including step by step tutorials, code snippets and in some cases downloadable example projects. Finally the documentation contains a class reference that contains GDScript global functions and variables, as well as individual pages on classes used in the engine, ranging from Nodes, Godot's basic building blocks, to data types such as vectors, arrays, etc.

Many articles in the documentation assume GDScript by default, but contain both GDScript and C# examples for most code snippets. There are also as several sections specifically about C#, its differences and equivalent techniques to those used in GDScript.

There are also many learning sources available from unofficial sources, such as "GDQuest", which contains written tutorials, video tutorials, as well as demo projects with downloadable project files.

Godot also has an official Asset Library, which contains plugins, templates, and even example projects.

1.5.3 Games made in Godot

- Brotato
- Cassette Beasts
- Dome Keeper
- The Case of the Golden Idol
- Windowkill

Chapter 2
Turn-based Game Project

This chapter will introduce the creative aspect of the development: the game design, the mechanics, gameplay loop, inspirations and the overall goal while designing the game to be implemented as the example project.

2.1 The Choice of Genre

The game is a turn-based RPG, inspired by traditional turn-based JRPGs such as Bravely Default, Dragon Quest, Persona and older Final Fantasy titles. As a genre, turn-based can sometimes be seen as an outdated system, characterized by repetitive menu navigation rather than deft real-time inputs.

Successful modern games with turn-based combat typically implement some unique mechanic that keeps things interesting and introduces complexity on top of the base combat system.

- Bravely Default with its Brave and Default system, allowing the player to act more times per turn at the risk of not being able to later, creating a risk and reward situation.

- Persona 5 with its One More system, which also encourages targeting weaknesses, the protagonists ability to create monsters with various attributes, as well as the mixing of the social link system, which creates more narrative focused down-time between combat.

- Yakuza: Like a Dragon adds timed prompts to attacks as well as while defending, giving the game qualities of an action game and therefore requiring the player to pay attention at all times.

This project is an attempt at introducing one such unique mechanic, in part by addressing another aspect that could be considered to be a shortcoming of many traditional turn-based games: fairly homogeneous attack options.

Many skills in traditional turn-based games tend to serve a similar purpose, that is to deal damage, with the only main difference being the elemental attribute and attack power of each skill. The game attempts to mitigate this by offering more varied skills with side effects that are meant to promote skill composition diversity and encourage exploring the synergy and combination potential between various skills. This was partly inspired by collectible card games such as Magic: The Gathering, which features thousands of cards with many different effects.

2.2 Traversal - The Dungeon Screen

The game begins inside the dungeon traversal screen, which is presented in the form of a deck of cards. A “Dungeon” is the term used for levels in the game, where each dungeon defines all of the cards contained in its deck.

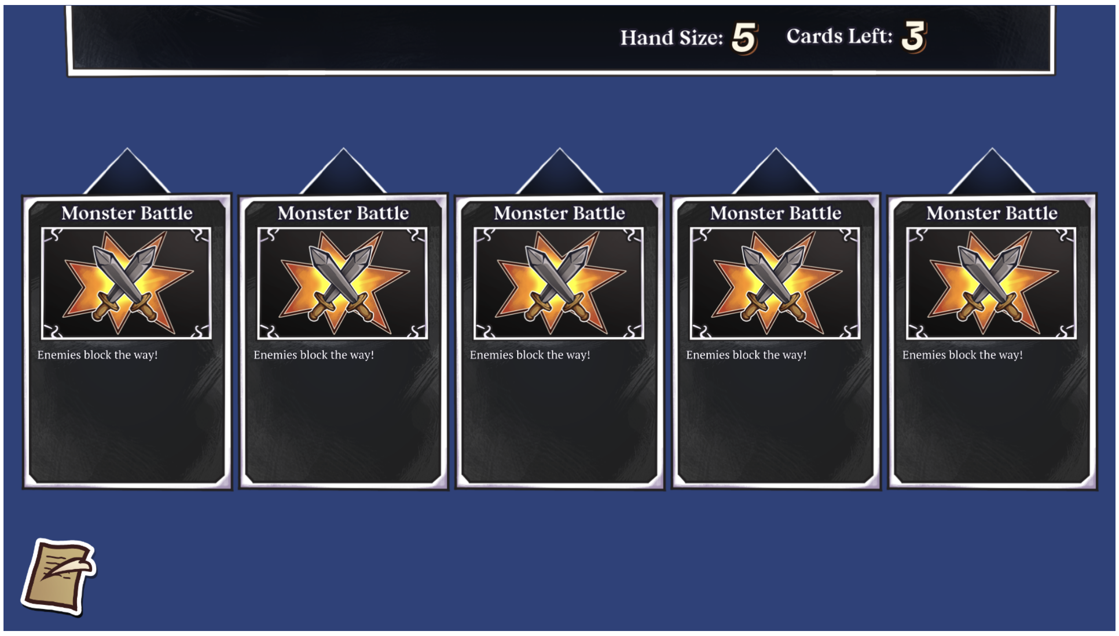


Figure 2.1 Screenshot from traversing a Dungeon in the game

A card can have many effects, from shuffling the hand, removing another card from play, triggering a battle, bestowing party members with boons or penalties, etc. However, at the time of writing, only the “Monster Battle” card is implemented in the example project, which is used to trigger a battle.

The goal of the game is to activate and empty all the cards out of the deck, one by one. There is a variable hand size which limits how many options the player has at every move, and efficient use of cards to avoid having to activate undesirable ones is encouraged.

When the final card is drawn, it will always be the “Boss Battle” card. Activating this card and subsequently defeating the Boss enemy means successfully finishing the dungeon. One does not need to activate the rest of the cards to clear the dungeon.

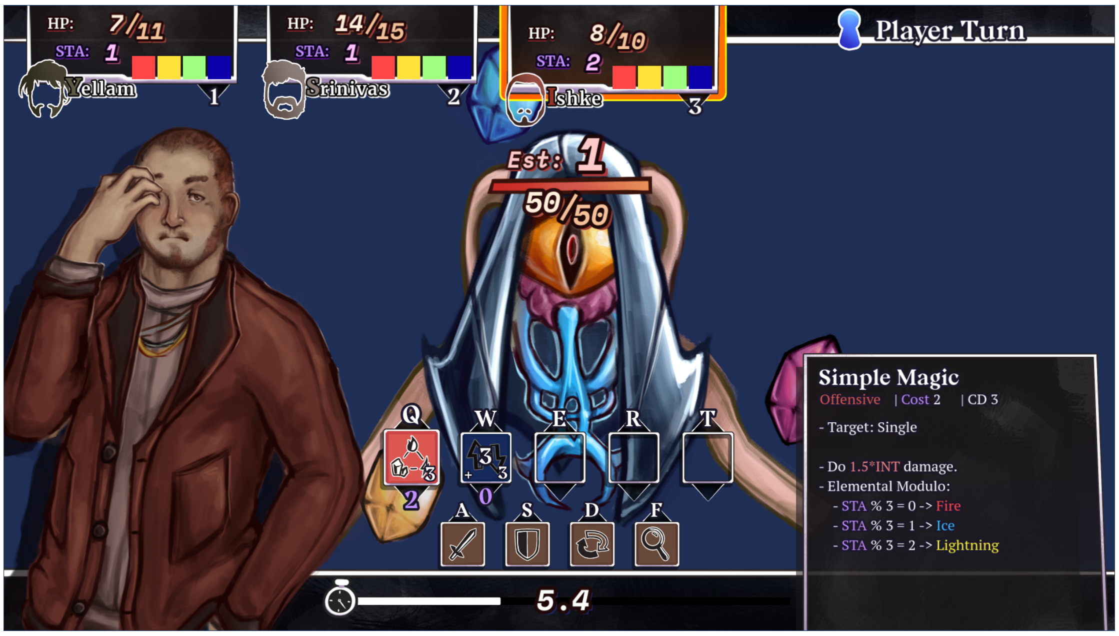
2.3 Combat

Combat is the main gameplay aspect of the game. Combat occurs when a “Monster Battle” card is activated during dungeon traversal.

During a battle, two opposing teams: the player and the enemy team, perform actions with the goal of defeating the other team.

2.3.1 Battle Design

Combat occurs in turns, switching between the player and the enemy.



Simple Magic
Offensive | Cost 2 | CD 3

- Target: Single

- Do 1.5*INT damage.
- Elemental Modulo:
  - STA % 3 = 0 -> Fire
  - STA % 3 = 1 -> Ice
  - STA % 3 = 2 -> Lightning

Figure 2.2 Screenshot from an in-game battle

The player's team consists of 3 active characters that partake in the battle, called the "Party", alongside 1 extra character not currently in the battle that can be swapped in and out, called the "Bench". The enemy team can contain any amount of enemies that are all active.

First, the player takes control during the Player Turn, directing all of their characters and selecting which act to perform. During the player turn, a timer counts down to encourage fast on-the-spot thinking. When all of the party members have expended their turn or the timer runs out, the Enemy Turn begins where the computer performs an act with each of the opposing units.

Every participant in a battle has two main stats: Health and Stacks. Much like in other games, Health describes how much more damage a participant can take before being defeated. Stacks are a resource serving a similar purpose to Mana, or Skill Points in other games. They are gained by performing the various acts during battles and do not carry over between fights. This design decision was made so that there would be less of an emphasis on resource management and Stack conservation typically prominent in other games.

The acts that the Player Characters are able to perform are: up to 5 unique skills assigned to each character, and basic actions available to everyone.

Using these actions, the goal the player is to defeat all enemy units by reducing their Health to 0. For the enemy units, the goal is to win by defeating any one of the player's units by reducing their Health to 0.

2.3.2 Battle Actors

Battle Actor is the term used for participants in a battle, that is both player characters and enemy units. Battle Actors contain several stats that each serve a different purpose:

Health/MaxHealth: a standard Health system: if Health drops to 0, a Battle Actor is defeated.

Stack: a Resource used to activate Skills. At the start of battle, each Battle Actor starts with a few Stacks, and further Stacks can be gained by performing various actions throughout the battle.

Strength: an attack stat used in calculations for Physical attacks.

Intelligence: an attack stat used in calculations for Magical attacks.

Defense: reduces damage taken.

Speed: this stat contributes to increasing the timer in case of player characters, and reduces the timer in case of enemies.

Element: the inherent Element of the Battle Actor. Attacks that default to the user's Element, such as Basic Attack, pull from this value.

Elemental Affinity: the Battle Actor's resistance to each element, that is, a percentile increase of reduction in damage taken from each respective element.

Skills: various abilities that the character is able to perform. A character is able to have up to five various skills.

2.3.3 Element System

Attacks in the game are split between two main types: Physical and Magical. Both Physical and Magical then have 3 specific elements associated with them, making it a total of 6 elements.

Physical Elements:

- Blunt
- Slash
- Pierce

Magical Elements:

- Fire
- Ice
- Lightning

There is also a unique 7th element: "None", which represents attacks that don't belong to any particular element.

Each offensive skill is assigned one of these elements. Some skills can also change their element dynamically based on some custom conditions, such as the "Basic Attack" skill, which gains the inherent Element of its user.

The Element of a Skill is factored into calculations when dealing damage, as each Battle Actor has different resistances and weaknesses to each Element.

2.3.4 Skills

Skills are the primary way interactions are performed in the game. Every action that both player characters or monster units perform is considered a skill. After a skill is used and its effects are resolved, the user's turn is considered finished and they will not be able to act again until the next turn.

Skills can have the following parameters:

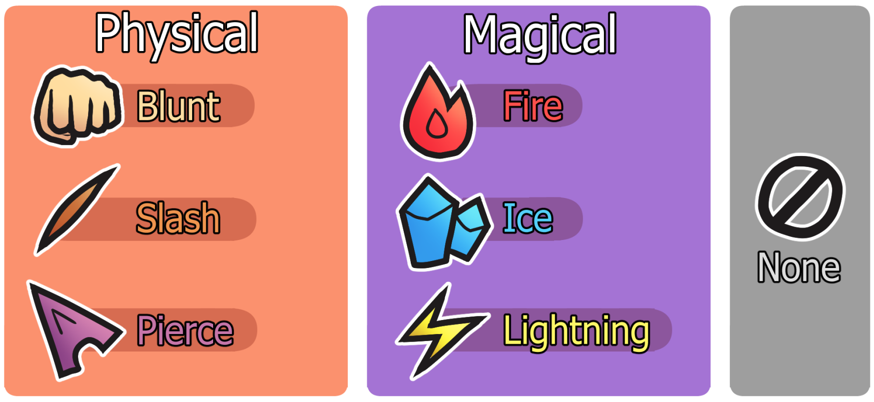


Figure 2.3 Element chart

Skill Type: describes the main function of the skill. These types are as follows:

Basic: the basic actions available to everyone.

Offensive: skills who primary function is to deal damage, i.e. reduce the Health of its target.

Utility: skills that perform miscellaneous effects, such as changing the Stack count.

Element: for offensive skills, determines its Element.

Cost: how many Stacks a battle actor has to pay to use the skill.

Cooldown: how many turns will have to pass for the skill to be usable again after activation.

Snap: a skill can have the “Snap” characteristic.

Snap Skills can be used without expending a turn meaning they can be combined with other actions to be performed on the same turn.

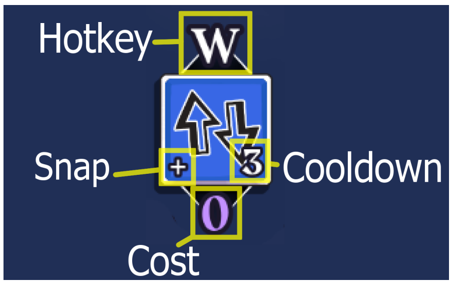


Figure 2.4 A skill as it appears in the in-game UI, alongside explanation of its various fields

The basic actions that are available to everyone are as follows:

Basic Attack: A standard attack that doesn’t require any Stacks, and generates 1 Stack.

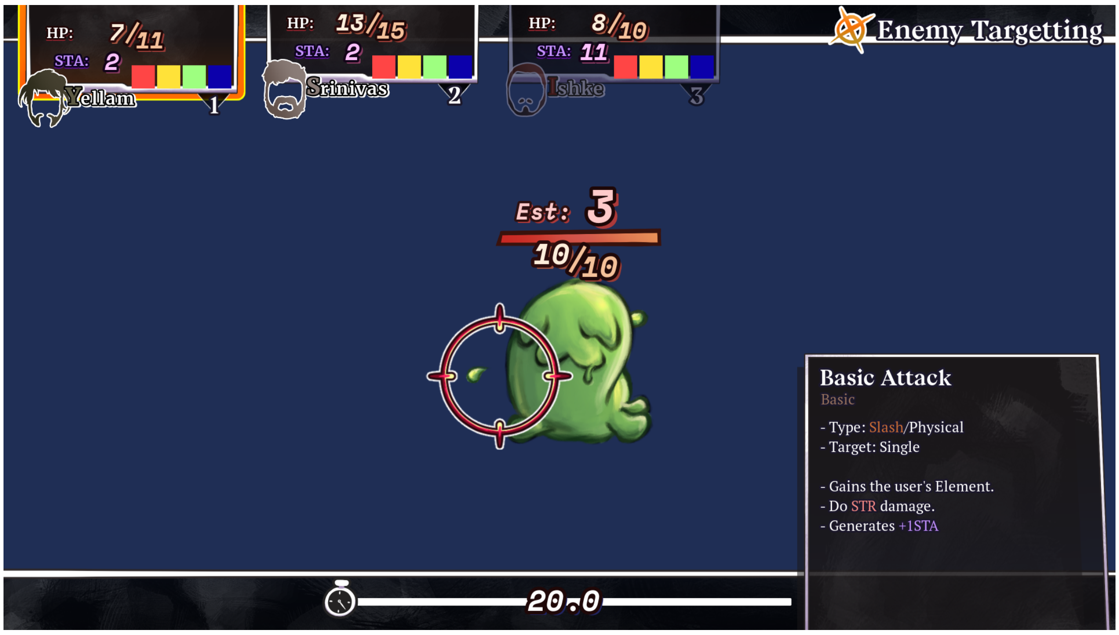
Analyze: Displays information regarding an enemy such as its attack patterns, skills and weaknesses. Generates 2 Stacks.

Swap: Swap between Characters in the Party and on the Bench, or even to swap positions of two characters in the main Party. Generates 2 or 1 Stacks depending on use.

2.3.5 Monsters

The term “monster” is used for all enemies encountered by the player during combat. Monsters are represented by an animated sprite model on the screen that is interacted with during battle.

When the Player prepares to attack an enemy, the cursor changes to a target shape. The player then selects which enemy to attack by directly clicking on an area on the screen. A monster can have several areas on its body that react differently depending on which one is targetted. These areas will be called “appendage”s.



Basic Attack
Basic
- Type: Slash/Physical
- Target: Single

- Gains the user's Element.
- Do STR damage.
- Generates +1STA

Figure 2.5 Game screenshot of player attempting to attack an enemy.

Depending on the appendage targetted, enemies can take more or less damage, or the player can even miss entirely. This allows enemies to be unique and distinguishable from each other by exhibiting different movement patterns and target areas to take advantage of.

The layout and most effective area to strike depends on the enemy. Each enemy has a an “appendage efficiency” value assigned to spots in its model, these areas can even change, appear, or disappear through various means. For example: in case of the simplest enemy, the “Slime” monster, it receives full damage when attacked near its face, and half damage when attacked anywhere else.

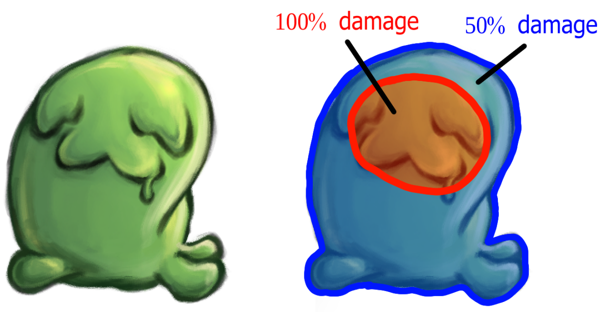


Figure 2.6 The “Slime” enemy, and its appendage layout based on effectiveness

There are also enemies that take advantage of disappearing appendages to become completely untargetable, such as the “Stack Sprite” enemy, which in its default form has no valid appendages, resulting in all attacks missing. This enemy reveals its one targetable appendage at random intervals, at which point the player is supposed to take advantage of the opportunity and attack it as soon as possible.

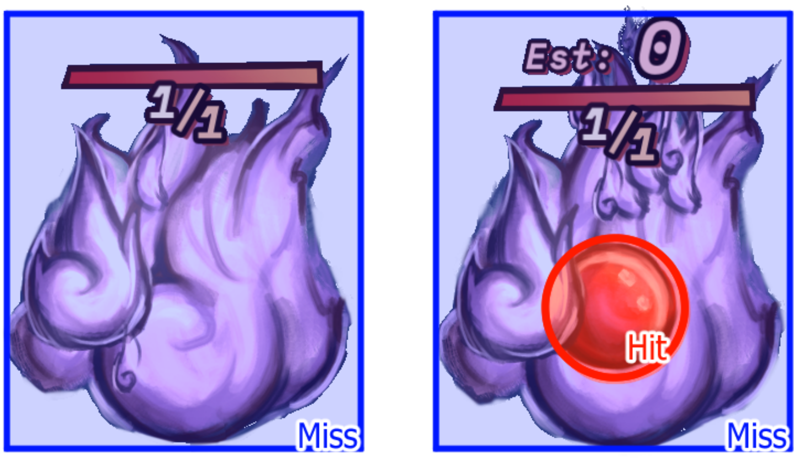


Figure 2.7 The “Stack Sprite” enemy with its appendage hidden on the left, and revealed on the right

The third enemy type, the “Rainbow Sentry” has a main body with very low appendage coefficients, making attacking the body ineffective. Instead its weak points are located in gems that move around it in a circular manner. These gems each correspond to a single element and are created randomly by the Rainbow Sentry through the course of battle.

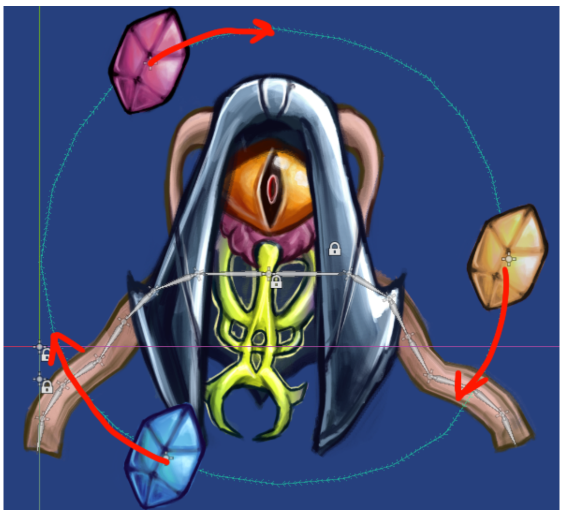


Figure 2.8 The “Rainbow Sentry” enemy, whose only effective appendages are located in the rotating gems

Chapter 3

Godot Basics and Techniques

In this chapter, we will go over setting up Godot, its basic concepts and features, basic editor navigation, and techniques and practices that work well with the design of the engine. All of the sections in this chapter will be written using tutorial-like wording, featuring step by step instructions.

3.1 Prerequisites

The first requirement will be downloading the Godot engine. Note that there are two separate downloadables for the Godot Engine, a regular and a .NET version, which features C# support and is therefore the build we will use.

Additionally, we will also need to download the .NET SDK, which contains tools necessary for for C# development.

Once everything is ready, open the example project by launching the Godot executable, clicking “Import”, and selecting the root folder of the project, i.e. the folder where “project.godot” is stored. Once it has been added to the Project Manager, open it and wait for the project to load.

3.2 Setting up Godot 4 for C# development

This section roughly follows the “C# Basics” page in the Godot documentation. [4] It will briefly explain how to prepare Godot for development with C#, as it requires a few extra steps compared to only using the natively supported GDScript.

Godot is designed for use with GDScript and its own in-engine code editor. However, this editor is only meant to be used for GDScript, and is quite limited compared to other, fully fledged code editors. As such, some additional steps will need to be taken to prepare the environment for C# development.

The code editor we will be using is Visual Studio Code (referred to as VS Code from now on). Alongside it, there are three extensions that we will use:

C#: Basic C# support for VS Code.

godot-tools: General support for Godot file types and syntax highlighting.

C# Tools for Godot: Additional support for Godot specifically for C# functionality, such as code completion.

To install these extensions, press Ctrl+Shift+X in VS Code, and search for C# and godot respectively and install them.

Next we will need to configure the Godot editor to use an external code editor. In the Godot editor, go to Editor > Editor Settings, and search for the “dotnet/editor” field. Under “External Editor”, select Visual Studio Code.

This will configure Godot to use VS Code when opening C# scripts, however it will still use the built-in editor for GDScript files. If you’d like to configure Godot to use VS Code for all scripts, those settings are under “text_editor/external”, in which case you’ll also want to set the “Exec Flags” field according to the documentation. [5] Now accessing a script in Godot should open a new VS Code window with the project folder as its root.

3.3 Basic Godot Editor Navigation

The following sections will frequently refer to the Scene, FileSystem and Inspector windows, which are highlighted in figure 3.1.

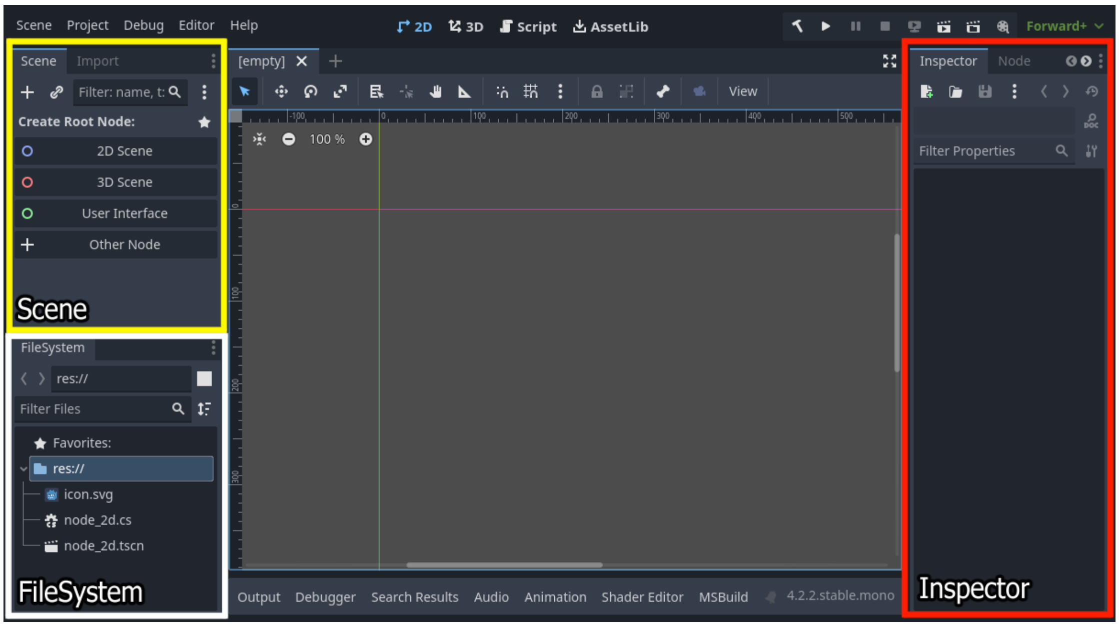


Figure 3.1 The Godot Editor layout

The FileSystem, Scene, and Inspector windows all have a search function that should be taken advantage of, as they allow one to quickly reach deeply nested parts in the respective windows.

One can also customize the layout of any of the windows by dragging the windows around the edges to resize them, and by pressing the three dot icon at the top right, and then picking a position, or make it floating, i.e. detach it into its own, separate window.

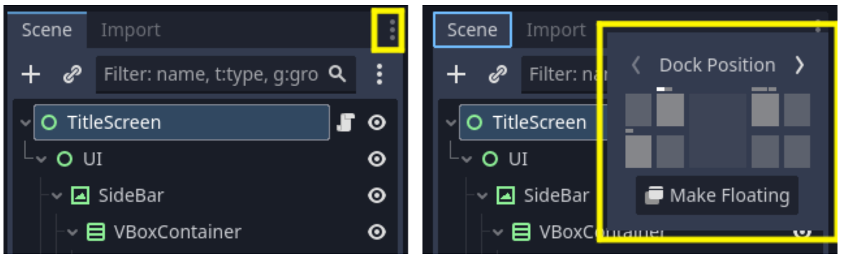


Figure 3.2 Customizing a window's layout in the editor

3.3.1 Scene Window

This window contains the layout of a Scene, which is a tree of Nodes, Godot's most basic building block. Nodes and Scenes are discussed in more detail in section 3.5 and 3.6 respectively.

In the example project, the currently selected scene should be the title screen of the game, contained in the TitleScreen Node. To create a new Scene, click + to the right of the TitleScreen tab.

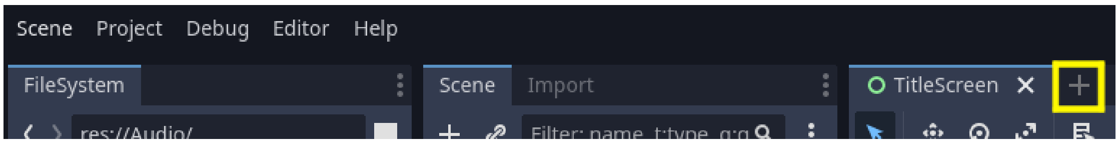


Figure 3.3 Creating a new Scene

The Scene window will prompt you to select a Node type to create as the root Node, i.e. the main Node of the scene that will be the parent Node to the rest of the children. For example select User Interface, which creates a Control Node, the most basic Node for creating UI. This will create a new Node tree with a Control Node as its root.

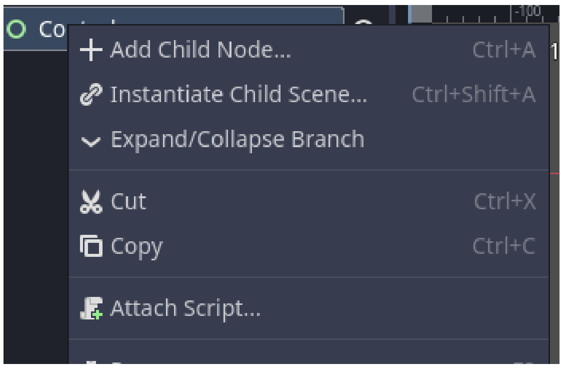


Figure 3.4 Right-clicking a Node in the Scene tree dialog

From here on out, you can build the structure of the Node tree by right-clicking the Node in the Scene dialog and adding more child Nodes. You can drag the nodes around and nest them in a tree-based manner.

You can save the Scene as a Resource by pressing Ctrl+S, which will prompt you to specify the scene's file path in the project files. Scenes are stored .tscn files, and are formatted as human-readable text files, making them naturally work well with version control systems such as git. You can then add these Nodes to a new Scene by dragging them into the Scene from the FileSystem window.

3.3.2 Inspector Window

The Inspector window lists all the properties of the currently selected Resource, whether it is a file from the FileSystem or a Node. You can view and edit the various properties through this window.

As Nodes can contain an incredibly high amount of properties nested within categories, navigation can be efficiently done by searching for the property by name, using the "Filter Properties" field. In the Nodes section at 3.5, many Node properties are referenced by their name and can easily be found using this feature.

3.3.2.1 Properties and passing by reference

You can copy and paste nearly any value in the Inspector window using the dialog that appears upon right-clicking it. Bear in mind that there are separate dialogs for when you right-click the property name, and when you right-click the value itself.

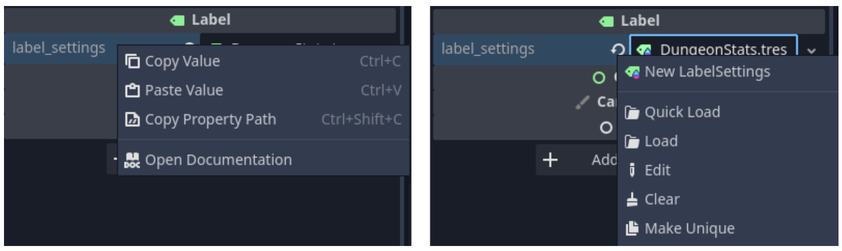


Figure 3.5 Right-clicking a property in the Inspector window. On the left: right-clicking the property name, on the right: right-clicking the value

It is important to note that: in case of fields that are Resources, i.e. any field that isn't just a primitive data type, they are copied by reference. As such, if one were to copy the label_settings property from one Label to another. Any changes done to this LabelSetting would also appear in every other Label that was set this way. To separate Resources, right-click the Resource and select "Make Unique" to create a separate copy of the Resource. Or in cases where a Resource contains more Resources nested within itself, "Make Unique (Recursive)" will create a separate copy for every nested Resource as well.

3.3.2.2 GDScript expressions in the Inspector Window

One very useful feature of the editor is that one can type GDScript expressions into Inspector fields. For example, you can use math operations such as addition, subtraction, multiplication, division, modulo, and even sin and cos, or constants such as PI.

In figure 3.6, one such expression is being used to edit the rotation property of a Node2D. Because rotation is stored in the unit of radians rather than degrees, a mathematical expression is used to convert 45 degrees into radians, directly in the Inspector.

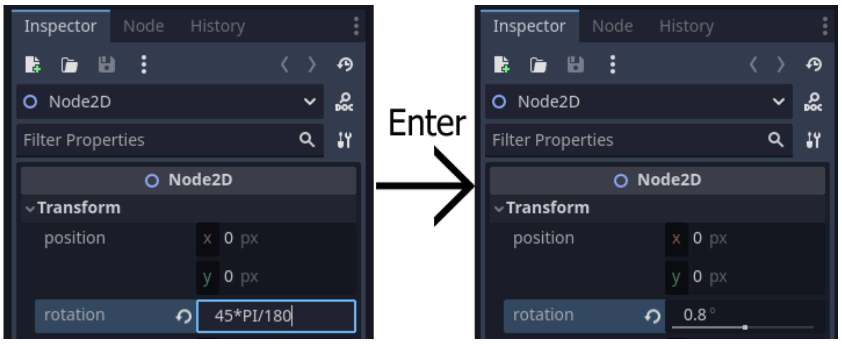


Figure 3.6 Entering a math expression into a field in the Inspector

3.4 Playtesting

At the top right corner of the editor are several buttons related to playtesting the game.

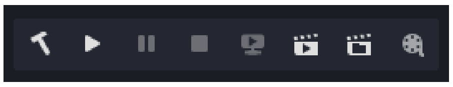


Figure 3.7 Top right buttons in the Godot editor. From left to right: Build, Run Project, Pause, Stop, Remote Debug, Run Current Scene, Run Specific Scene, Movie Maker Mode

The leftmost button, Build, is specific to the C# version of the engine, and will rebuild the C# assemblies. This is necessary when an exported variable or signal is added to a script, as the editor will otherwise not automatically update and display the fields in the Inspector window. Exported variables and signals are discussed in more detail in 3.9 and 3.7 respectively.

Run Project will launch the game from the Main Scene, alternatively you can Run the Project from the currently edited Scene, or by selecting a specific Scene from the FileSystem.

During playtesting, the Scene window will gain a “Remote” tab, which when clicked will show the live Scene tree of the game as it is played. You can access any of the Nodes to see their current properties in the Inspector window, and even change them.

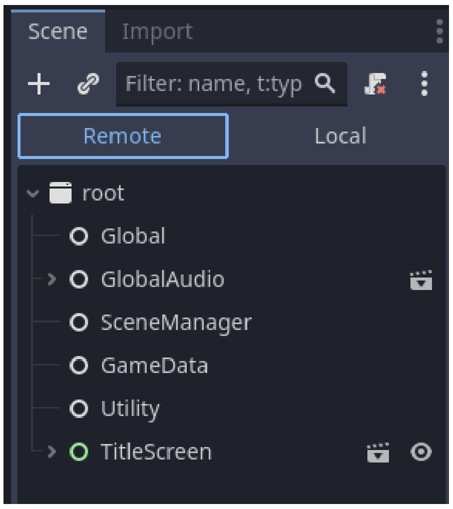


Figure 3.8 The Scene window during playtesting

3.5 Nodes

Nodes are one of the most important concepts in Godot. They are the smallest building block that are used to build nearly every element used in a Godot game. [6]

There are many types of Nodes, and every Node is also a class. Many of them are built on top of each other through the use of inheritance. For example, the Control Node, which is the base Node used for all UI elements, is inherited by other UI Nodes implementing a specific UI function such as clickable buttons, a scrollable window, text labels, etc.

Through this inheritance-based design, Nodes are split into different categories of sorts. There is Control, used for UI elements, as well as Node2D and Node3D, which are used for 2D and 3D features of the engine respectively. There are also Nodes that do not belong to either of these categories, such as Timer or AudioStreamPlayer.

Nodes are then used to build tree structures, where a Node can have any number of child Nodes, which can in turn also have their own child Nodes, creating a complex, nested and branching structure. In a Node tree, Nodes at the top are drawn first, and then the lower Nodes are drawn on top of them. Child Nodes are also drawn in front of their parents by default.

A Godot game consists of a “root” Node, which is the topmost Node that then contains all other Nodes as its children. Every Node inside this tree is processed as part of the program. The class that handles and manages this hierarchy of Nodes and therefore the game loop is called the “SceneTree”.

A Node tree which implements one coherent logical whole is called a Scene, which is discussed in more detail in 3.6.

Here is an overview of Nodes most prominently used in the project:

- Node2D
  - AnimatedSprite2D
  - CanvasGroup
  - Sprite2D
- Control
  - Button
  - ColorRect
  - HBoxContainer/VBoxContainer
  - Label
  - NinePatchRect
  - RichTextLabel
  - TextureRect
- AnimationPlayer
- AudioStreamPlayer

3.5.1 Node2D

Node2D is the base Node from which all Nodes that use the 2D part of the engine inherit. Here are some of its most notable properties:

position: stored as a Vector2, which is a structure of two floating-point values, here used to represent the x (horizontal) and y (vertical) axis.

Note that in Godot, the y axis points downward, this means that the higher the y position value is, the lower on the screen it is.

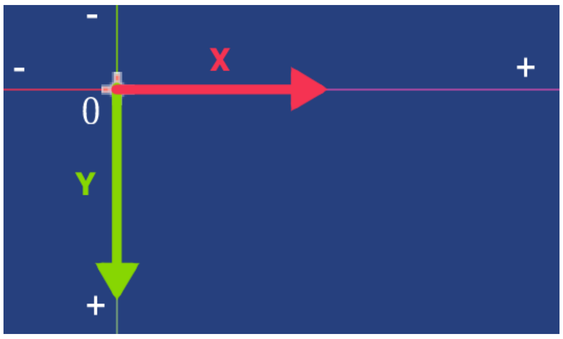


Figure 3.9 The 2D coordinate system in Godot

rotation: rotation stored in radians.

scale: stored as a Vector2, defines the scaling multiplication of the Node.

Node2D also inherits from the abstract CanvasItem class, This means that Node2D also contains CanvasItem attributes, such as:

visible: whether to draw the Node or not.

modulate: tints the Node by multiplying it with a specific color. Modulates every child Node as well.

The color of the modulation is set to pure white by default, which simply displays the sprite without any changes.

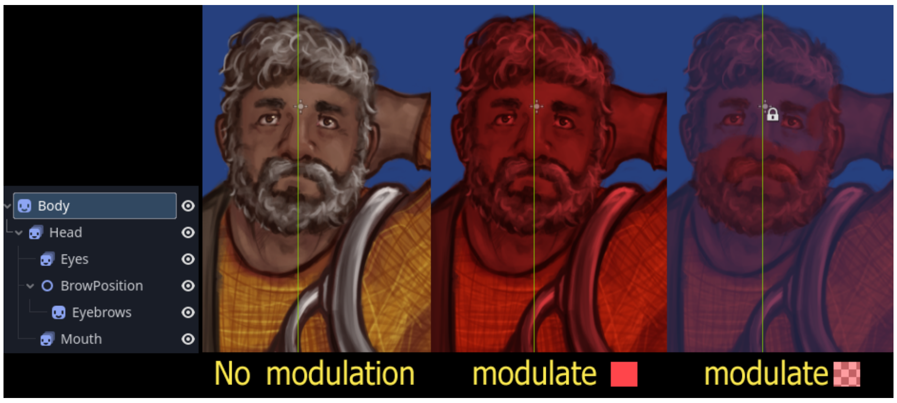


Figure 3.10 From left to right: a Sprite2D with no modulation, red modulation, and red transparent modulation

self_modulate: tints the Node by multiplying it with a specific color. Modulates only self, leaving child Nodes the same.

z_index: the drawing order of the Node. Using this property, you can change the layer order without having to change the Node's position in the Node tree.

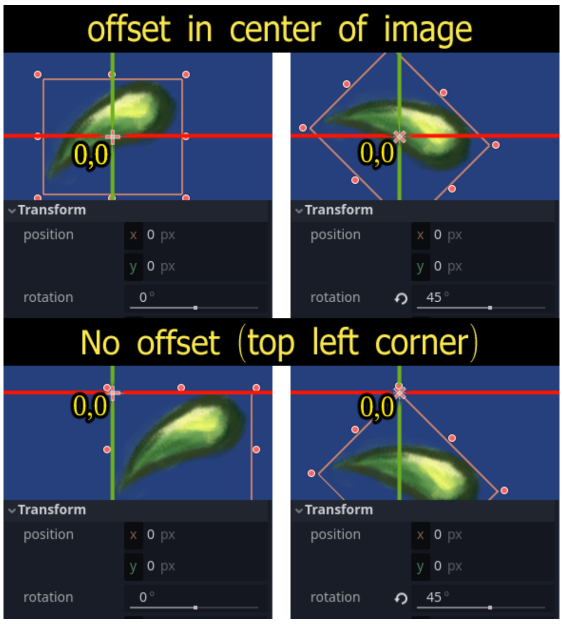
material: a material is used to alter a node's appearance by changing its blend mode, or by using a shader[1].

All Node2D Nodes follow the transformations, i.e. position, rotation and scale, as well visibility and modulation from their parent Node2D Nodes. This means that moving a parent Node2D will collectively move all of its children as well, and hiding a parent Node2d would hide all of its children.

3.5.1.1 Sprite2D

This is one of the most basic Nodes used to display images. The image it displays is assigned to the "texture" property.

It contains the "offset;; property, which defines the origin point of the sprite, i.e. where on the sprite is considered the 0,0 position. It is also the pivot point around which the rotation property rotates, and from which the scale property scales.



Transform
position x 0 px
y 0 px
rotation 0°

Transform
position x 0 px
y 0 px
rotation 45°

Transform
position x 0 px
y 0 px
rotation 0°

Transform
position x 0 px
y 0 px
rotation 45°

Figure 3.11 Sprite2D's offset property, note the position of the Node relative to the 0,0 coordinate

3.5.1.2 AnimatedSprite2D

A Node used to display a sprite using frame by frame animation. It can contain multiple different animations to switch between.

When you select an AnimatedSprite2D Node in the Scene window, an Animation Editor window will appear at the bottom of the editor, where you will be able to add the respective animation frames, edit their durations, manage all the animations contained by the AnimatedSprite2D, etc.

[1] A shader is a program that alters visuals. Discussed more in 5.1.3

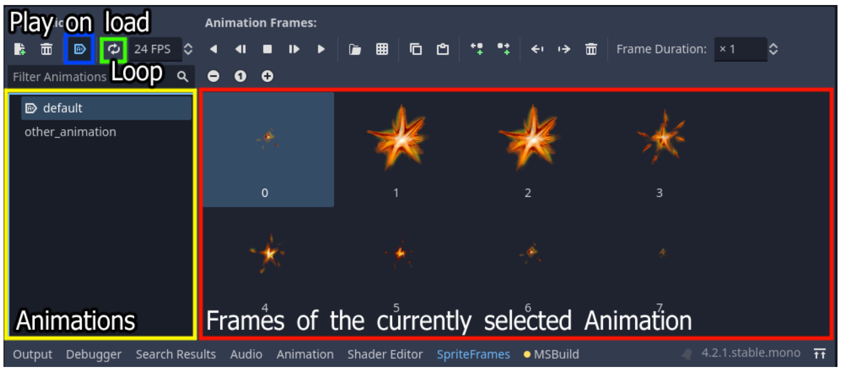


Figure 3.12 Editing an AnimatedSprite2D in the editor

An AnimatedSprite2D will not start playing the animation by default. To do this, toggle Play on load in the editor, or start the animation using code.

Code listing 3.1 Starting an AnimatedSprite2D from code

```
animatedSpriteNode.Play(nameOfAnimation);
```

3.5.1.3 CanvasGroup

A CanvasGroup Node will group its child Sprite2D Nodes together to be rendered in one draw call, i.e. it draws multiple Sprite2D Nodes as if they were one singular image. This is useful for cases where a visual that consists of several child sprites needs to modulate its transparency. If one were to use a standard Node2D and modulate it alongside its children, it would lower the transparency individually for each Node, revealing the overlap between each part.

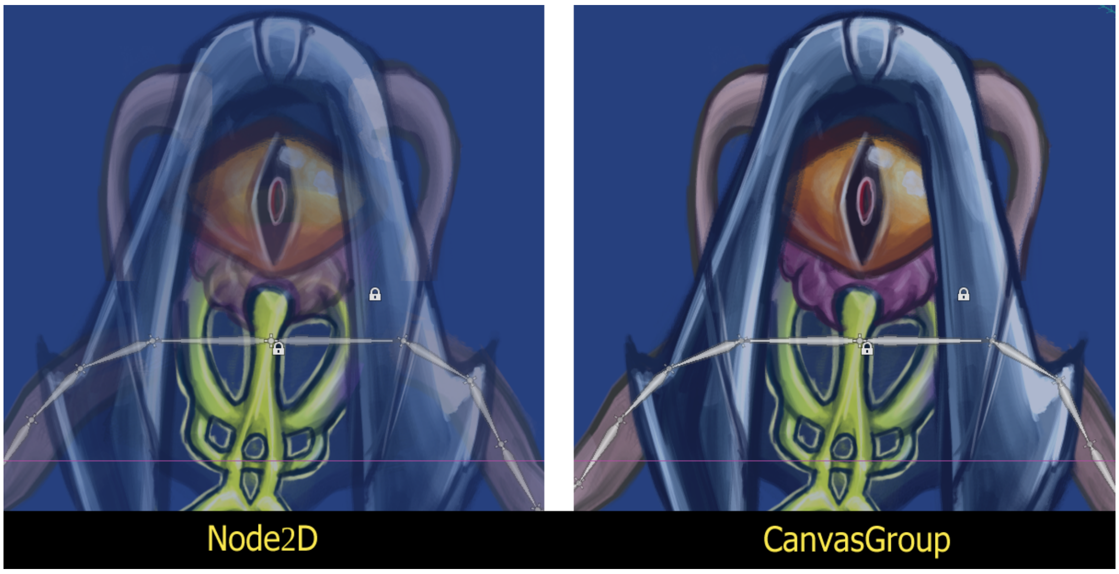


Figure 3.13 Modulating sprite transparency on a regular Node2D vs. CanvasGroup

It is also useful for use with shaders for the same reason, i.e. reading the visual as one single image, rather than having to set the shader to each child Sprite2D individually.

3.5.2 Control

Control is the base Node from which all other UI element Nodes inherit. The Control Node and all of its children should be used for UI elements, as opposed to gampeplay elements, such as player or enemy sprites, where Node2D should be used instead.

Like Node2D, Control also inherits from CanvasItem, which gives it the same properties such as visible, modulate, self_modulate, z_index and material.

Control Nodes also have many of their own properties that are used to specify their behaviour and various UI-related functions, which will be described in the following sections.

3.5.2.1 Theme

A Theme defines the appearance of Control Nodes. It is a property that can be set from the Inspector window. Child Control Nodes will inherit the Theme of their parent Nodes.

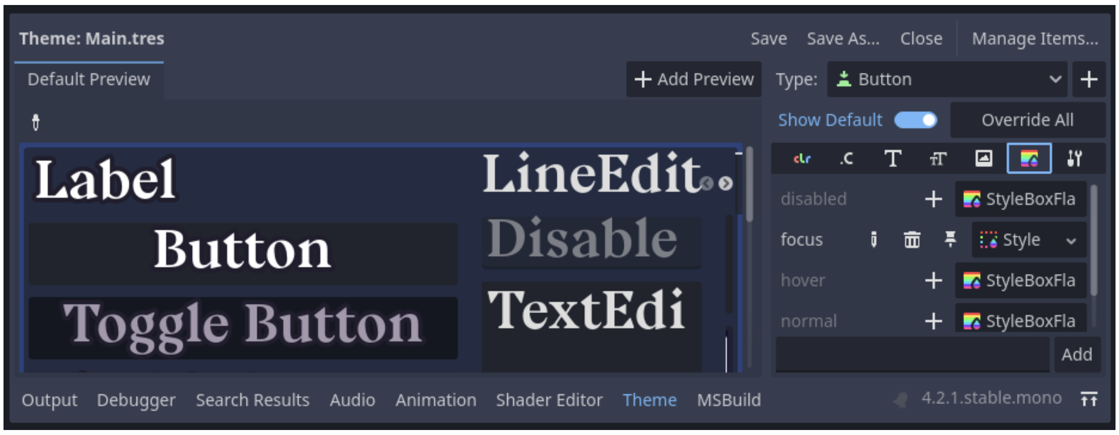


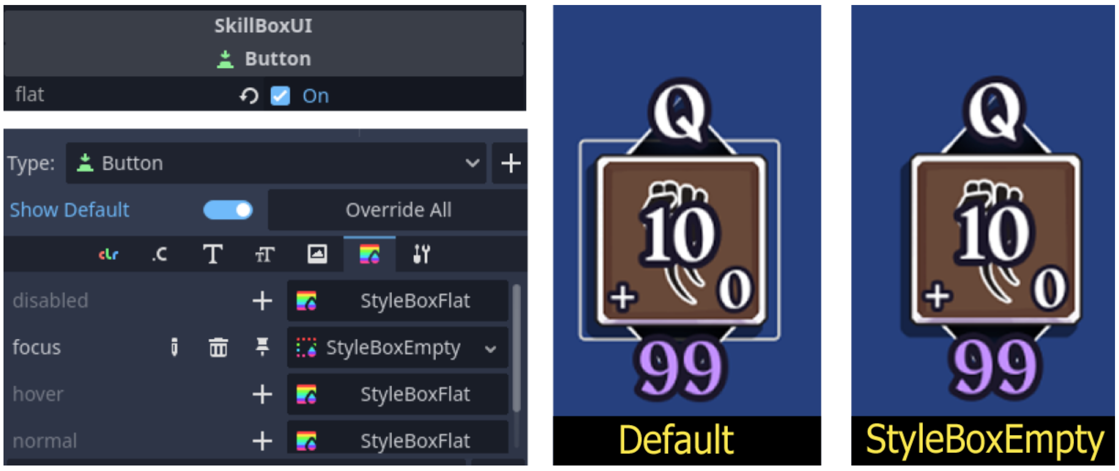
Figure 3.14 The Theme Editor

After creating a new Theme or selecting a Control Node and then clicking the Theme assigned to it in the Inspector window, the bottom window will switch to the Theme Editor. A theme contains information about the “default_font” to be used, as well as settings for individual Control Nodes.

To add a Control Node to the list of Nodes edited by the theme, press the + button next to “Type:”. Its various properties can then be edited in the various tabs below. To override a property, click on the + to the left of the value.

For example, settings on a button’s “StyleBox” will determine its appearance when disabled, currently focused, hovered over by a mouse, and while the mouse button is held down. This appearance is determined by a StyleBox Resource, which can be flat-colored, empty, or use an image texture.

In the example project, a combination of Theme and Button Node settings is used to achieve a truly flat button: a button without any decorations, to be used as a gameplay UI element.


SkillBoxUI
Button
flat On

Type: Button
Show Default
Override All

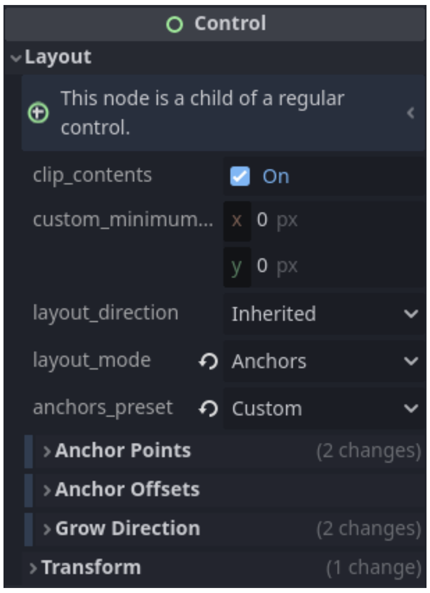
disabled	StyleBoxFlat
focus	StyleBoxEmpty
hover	StyleBoxFlat
normal	StyleBoxFlat

Figure 3.15 Theme settings on the flat button

The flat property of the Button Node removes the Button's background. However, the button will still display a border when it is focused. To remove the border, the "focus" appearance of the Button is edited in the Theme Editor to display a StyleBoxEmpty.

3.5.2.2 Control Nodes and Responsive Design

Godot's Control Nodes are designed to be usable outside of video game contexts. In this sense, they behave similarly to UI components in a standard desktop application, and allow for responsive design, i.e. automatically resizing and repositioning when the window resolution and aspect ratio changes. To specify and configure this behaviour the properties in the Layout section of a Control Node are used.


Control
Layout
This node is a child of a regular control.

clip_contents	On
custom_minimum...	x 0 px
	y 0 px
layout_direction	Inherited
layout_mode	Anchors
anchors_preset	Custom
Anchor Points	(2 changes)
Anchor Offsets	
Grow Direction	(2 changes)
Transform	(1 change)

Figure 3.16 Inspector Window: the Layout properties of a Control Node

Some notable Layout properties are:

clip_contents: whether to cut off drawing of child Nodes that are extending outside of this Control Node's range. This will clip all CanvasItem Nodes, meaning both Control and Sprite2D Nodes.

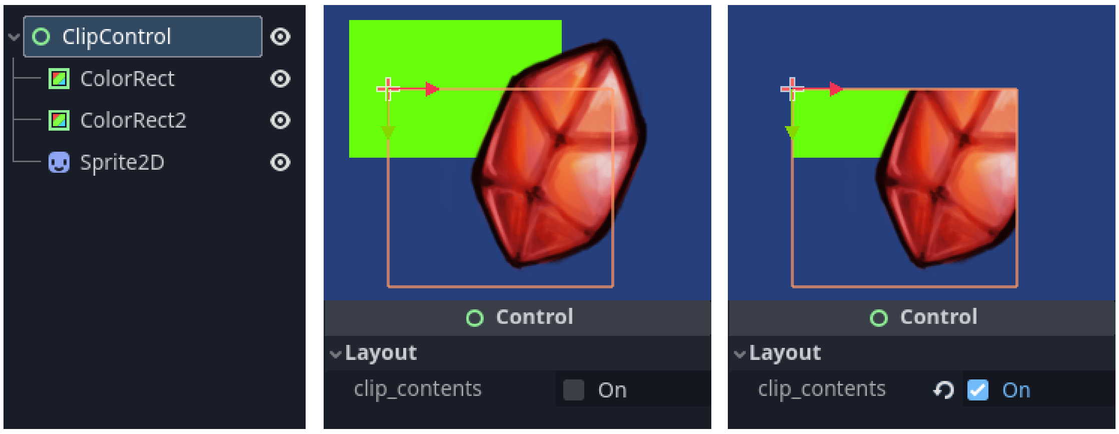


Figure 3.17 Control Node's clip_contents property

custom_minimum_size: Control Nodes resize and scale themselves to fit the parent. This can often be an issue when it causes the Node to get squashed, cut off, or become too small. Using this property, one can force the Node to always be at least a certain size.

layout_mode: whether to layout the Control Node using x,y position similar to a Node2D, or use Anchors to facilitate responsive design.

anchors: when the layout_mode is set to Anchors, the Control Node will expand, shrink and resize itself accordingly to the window resolution.

There are presets which automatically set the anchors to one of several predefined behaviors, such as full screen, anchor to the top left top row, etc. Or, using Custom anchors, this behavior can be defined in more detail.

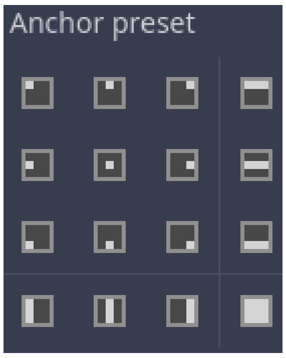
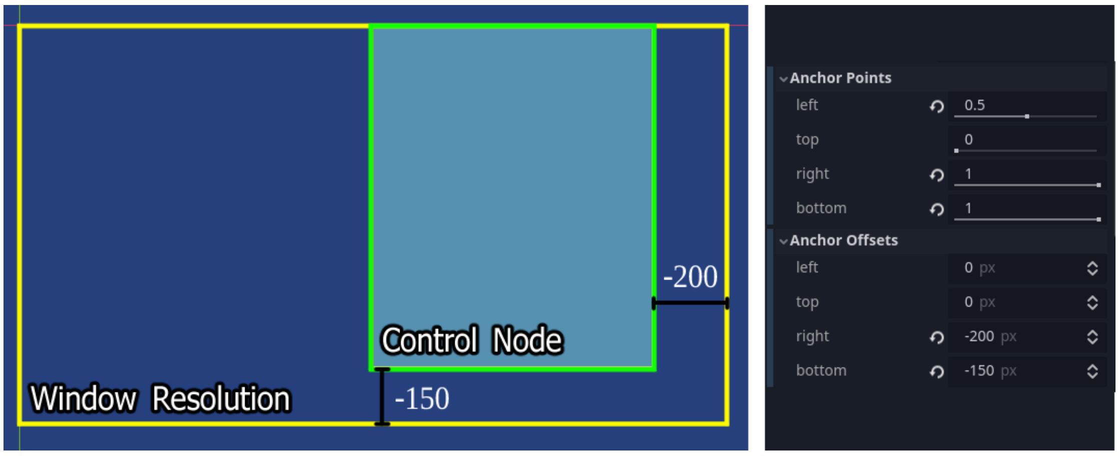


Figure 3.18 The various Anchor presets of a Control Node

When using Custom anchors, there are Anchor Points and Anchor Offset properties for the top, left, right and bottom respectively.

Anchor Points are float values that typically range from 0 to 1. They are a percentage-based representation of the window size, where 0 is the beginning, and 1 is the end. For example, the left Anchor point at 0 would cause the left edge of the Control to stick to the left side of the screen, and at 1 the left edge of the Control would stick to the right side of the screen. Anchor Points can also exceed 1, in which case they expand beyond the visible window.

Anchor Offsets are a pixel integer value that define a fixed amount of pixels the edge will be shifted from the Anchor Point.

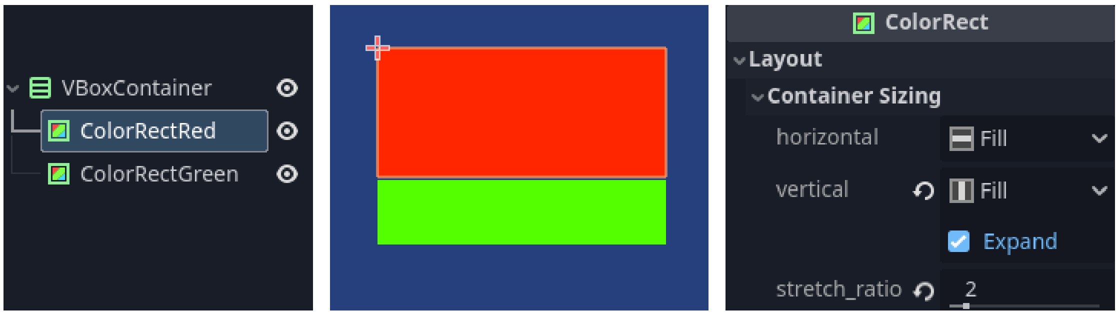


Anchor Points
left 0.5
top 0
right 1
bottom 1
Anchor Offsets
left 0 px
top 0 px
right -200 px
bottom -150 px

Figure 3.19 An example of a Control Node with various Anchor values set

Aside from using Layout properties, there is a special type of Control Node: "Container". Container Nodes are are used to hold other Control Nodes as their children. Container Nodes automatically handle the arrangement of their children, their scaling and positioning.

When a Control Node is a child of a Container Node, most of its Layout properties are not editable, as those are handled by the parent Container Node. This can make Container Nodes unintuitive to work with, as they remove a lot of control over the placement and size of child Nodes. However, alongside "custom_minimum_size", once can use properties in the Container Sizing section to customize the way a Control Node behaves when it is the child of a Container Node.



ColorRect
Layout
Container Sizing
horizontal Fill
vertical Fill
✓ Expand
stretch_ratio 2

Figure 3.20 Child Node of a Container with Expand and stretch_ratio

3.5.2.3 User Input and Focus, Mouse properties

Under the Focus and Mouse sections are properties that dictate how the Control Node behaves during user input.

Focus: allows for UI to be navigated using solely the keyboard or a controller without requiring mouse input.

This style of navigation works by always having a Control Node that is currently selected, i.e. it has Focus.

Navigation can then be done by pressing up, down, left or right on the controller or keyboard. This behaviour can be customized by manually assigning which neighbour each direction would jump to, although in most cases, Godot's built-in logic should suffice and manually fine-tuning neighbours should not be necessary.

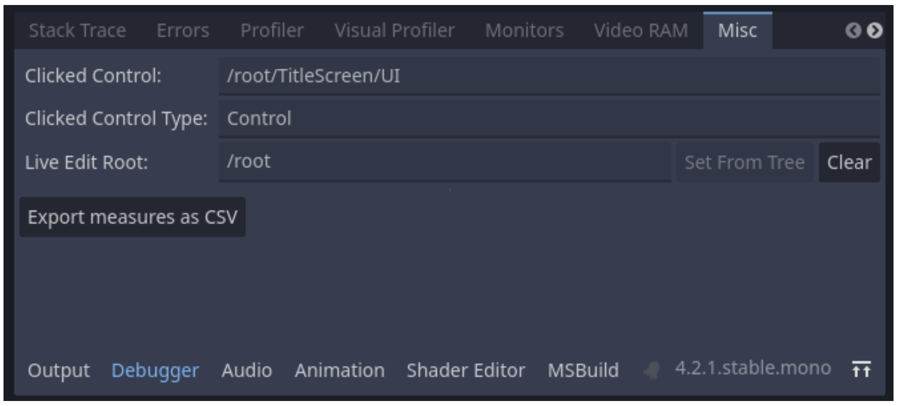
By setting the "focus_mode property", Focus can also be entirely disabled, set to only activate through mouse inputs, or set to activate through both mouse and key input which is the default setting.

Mouse properties in a Control Node dictate how mouse input is handled.

The "mouse_filter" property determines how the Node handles mouse events such as hovering over the Node, or mouse button inputs. When set to Stop, the Node will stop and handle the input. When set to Pass, the Node will receive the input, but if it is not handled, it will be passed to its child Nodes. When set to Ignore, it will not capture any inputs whatsoever.

When a Control Node with mouse_filter set to Stop or Pass receives mouse input, it handles it, and the input will not propagate any further to other Control Nodes, or in case of Pass it only propagates it to its child Nodes if it does not handle it itself.

In cases where mouse inputs are not reaching a Control Node as desired, it is likely because a different Control Node is consuming the input instead. This can be debugged fairly easily during playtesting using the Debugger in the bottom window, in the Misc tab, which details the last Control that was clicked.



Clicked Control: /root/TitleScreen/UI
Clicked Control Type: Control
Live Edit Root: /root

Figure 3.21 The Misc tab in the Debugger showing the last clicked Control

3.5.2.4 Button

A Node for a standard flat-colored button. For a button that uses a custom image texture, the TextureButton Node should be used instead. Both of them inherit from the abstract BaseButton class, which has the following properties:

disabled : if this property is true, the Node will not register any presses.

toggle_mode : makes the button act like a toggleable button. The state of the button is then stored in the “button_pressed” property.

button_mask : by default, buttons will only register left mouse clicks. Using this property, one can customize the button to also read right or middle clicks.

To implement behaviour on being pressed, connect the “pressed” signal of the Node to a method. Alternatively you can also use the more granular “button_down” and “button_up” signals, which are emitted when the button is pressed down and released respectively.

3.5.2.5 ColorRect

ColorRect is a rectangular Control Node displaying a flat Color. It can for example be used for fading-to-black transitions, or to darken visuals behind it by setting it to a translucent color.

3.5.2.6 HBoxContainer/VBoxContainer

VBoxContainer and HBoxContainer are Container Nodes that arrange their children to be lined up vertically and horizontally respectively.

With the “alignment” property, these children can be aligned to Beginning, Center, or End. Using the “separation” property under Theme Overrides, one can also define a gap between each Node.

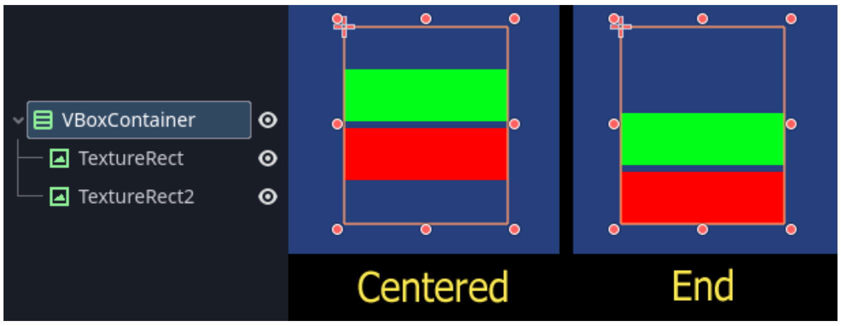


Figure 3.22 A VBoxContainer containing two child ColorRect Nodes, showcasing the “alignment” property

Being a Container Node, the movement and scaling behaviour of its child Nodes is defined individually for each of those Nodes using their properties in the Container Sizing section in the Inspector window.

3.5.2.7 Label

Label is used to display basic text. Some of its properties include:

label_settings: font settings that can be shared among Label Nodes. Overrides the default font of the Theme.

horizontal_alignment: whether to anchor the text to the Left, Center, Right, or to Fill the Label Node.

vertical_alignment: whether to anchor the text to the Top, Center, Bottom, or to Fill the Label Node.

autowrap_mode: by default, Labels will expand horizontally to fit in text outside of its region. With autowrap, it will instead wrap text around. Text wrapping can be Arbitrary (at any point in the text) or by Word

Additionally, Labels also contain the “visible_characters” and “visible_ratio” properties. Which can be used to display only a portion of the text contained within the Label. Whereas visible_characters displays a specific amount of characters, which requires information about how long the current text is, visible_ratio displays a percentual amount of the characters using a range from 0.0 to 1.0. These properties can be animated to create a text scrolling effect commonly used in games.

3.5.2.8 RichTextLabel

RichTextLabel is used for displaying formatted text, such as bold or italicized letters, ordered and unordered lists, or even images.

These effects are facilitated using BBCode. By using “tags”, which are enclosed around text, various effects can be achieved. For a more detailed rundown of all BBCode tags available, refer to the BBCode in RichTextLabel page in the Godot Documentation. [7]

First, BBCode must be enabled in the Node’s “bbcode_enabled” property. Afterwards, the string in the text property will automatically be parsed for BBCode tags and displayed accordingly.

Code listing 3.2 Example of BBCode use in RichTextLabel

```
Regular.
[i]Italicized[/i].
[b]Bold[/b].
[b][i]Bold and italicized[/i][/b].
[color=#EFAA00]Colored[/color].
```

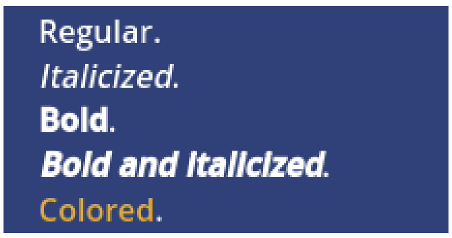

Regular.
Italicized.
Bold.
Bold and Italicized.
Colored.

Figure 3.23 The resulting text displayed using the text at 3.2

Note that it is necessary to provide separate font files for regular, bold and italicized text each.

One can also create custom tags and effects by creating a class inheriting from RichTextEffect. In the example project, this is used in a very surface level manner to create a tag that simply changes the color of the text to a predefined value, so as to not have to use the very long “[color=#...]” tag every time it is needed.

Code listing 3.3 Example of BBCode use in RichTextLabel

```
[Tool]
[GlobalClass]
public partial class RichTextAtkColor : RichTextEffect
{
    // The name of the tag
    // i.e this effect would be used as: "[atk]text[/atk]".
    public readonly string bbcode = "atk";

    // Implementation of the tag.
    public override bool    _ProcessCustomFX(CharFXTransform charFX)
    {
        charFX.Color = new Color("#F38984");
        return true;
    }
}
```

Custom BBCode effects have to be enabled for each RichTextLabel Node by adding it to the “custom_effects” property. Note that for the RichTextEffect to appear in the list of offered RichTextEffects, a build of C# assemblies, followed by a restart of the engine may be needed.

3.5.2.9 NinePatchRect

A Control Node that displays an image texture using the 9-slice scaling technique.

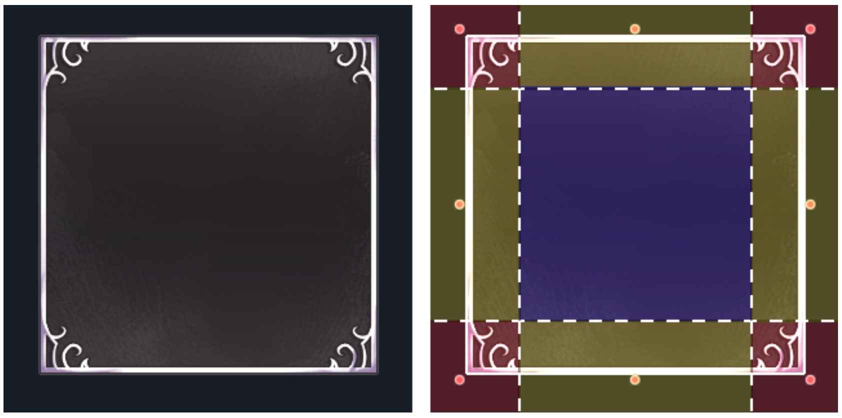


Figure 3.24 On the left: an image texture, on the right: the image texture separated into 9 areas using the 9-slice technique

By separating the image texture into 9 areas: the outer edges and a center, the image can be resized without significantly distorting the image texture, as only the center is stretched, whereas the edges and corners are preserved. This makes NinePatchRect ideal for windows that need to be displayed at various sizes.

These areas can be defined visually in the Inspector Window using the “Edit Region” button, or manually by entering pixel values for each edge in the Patch Margin properties section.

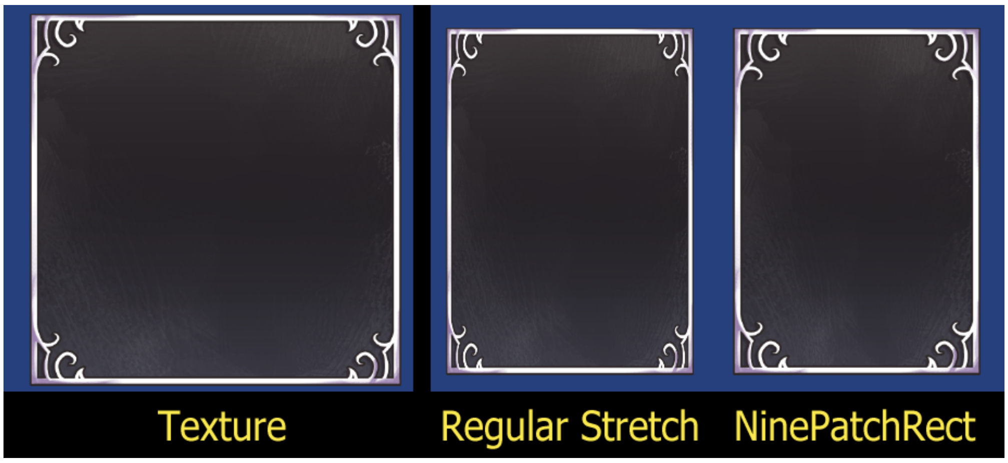


Figure 3.25 A comparison between regular image stretching and NinePatchRect

3.5.2.10 TextureRect

A Control Node that displays an image texture. While it may seem to serve the same function as Sprite2D, a Control Node is meant to be used for UI elements because of its properties that allow for responsive design, whereas Sprite2D is generally meant to be used for gameplay elements.

TextureRect contains the following properties:

expand_size: defines how the minimum size is determined.

By default, it is set to "Keep Size", which makes the TextureRect Node unable to be scaled smaller than the texture resolution.

stretch_mode: how the image texture is aligned with the overall area of the Node when it is stretched into a different size than the one of the texture.

3.5.3 AnimationPlayer and Tween

The AnimationPlayer Node can be used to animate nearly every property of a Node. Aside from AnimationPlayer, there is also another method to animating values in Godot: Tween.

3.5.3.1 AnimationPlayer

An AnimationPlayer can contain multiple animations, which are identified by a string name. To play an animation, the Play() method of the AnimationPlayer is used, with the animation's name passed as an argument.

Code listing 3.4 Playing an animation on an AnimationPlayer Node

```
someAnimationPlayer.Play("animation_name");
```

Upon selecting an AnimationPlayer Node in the Scene window, the bottom window will automatically switch to the Animation tab, where editing and management of its various animations takes place.

An animation consists of one or more properties being changed over time. These changes are defined using several "keyframes" placed over a timeline, which dictate the value the property will have at a certain time point in the animation. The animation is then achieved by blending the values between keyframes. An AnimationPlayer can animate any number of properties from any number of Nodes.

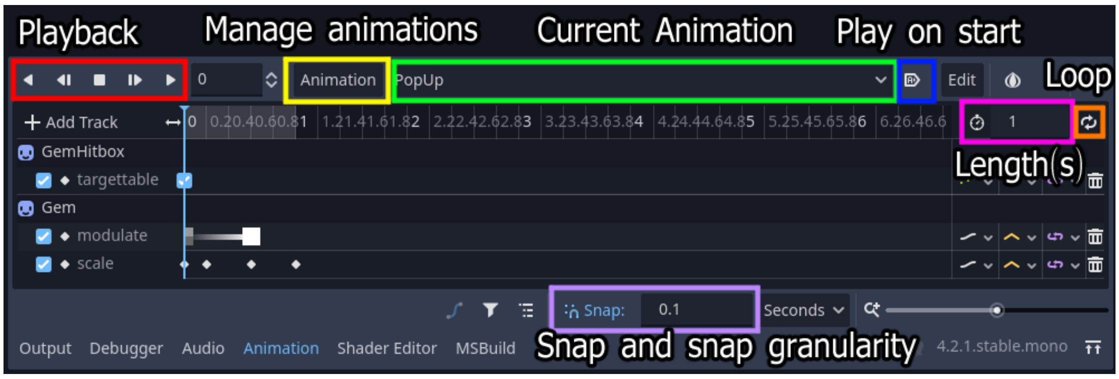


Figure 3.26 The Animation window

To animate a property, it first needs to be added to the currently edited animation as a Track. To do that, press Add Track > Property Track, and select the corresponding Node and property you want to add.

There is also another way to add a new property Track: when the Animation window is currently active in the bottom window, a key button is added to the right of all properties in the Inspector window, which will add a keyframe of the property with the current value to the animation timeline. If the property doesn't have a Track in the animation yet, it will be created and added automatically.

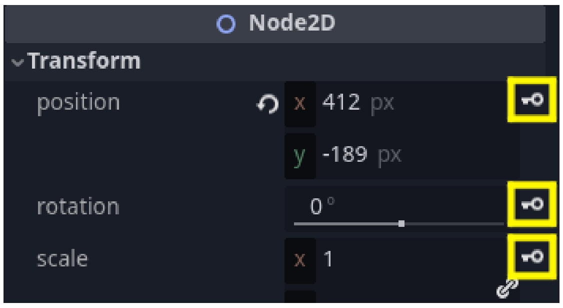


Figure 3.27 Keyframe buttons added next to properties in the Inspector window when the Animation window is active

When working with keyframes in the timeline, you can add keyframes by right-clicking the corresponding track on the timeline and selecting "Insert Key". To edit a value of a keyframe, click the keyframe in the timeline, which will display its value in the Inspector window. you can also change the "easing" of the keyframe, which defines the curve, using which the value will be blended to the following keyframe. By right-clicking the curve, you can also pick from various curve types.

A final way to add keyframes is using the toolbar. When the Animation window is active at the bottom, the toolbar adds several options related to animations.

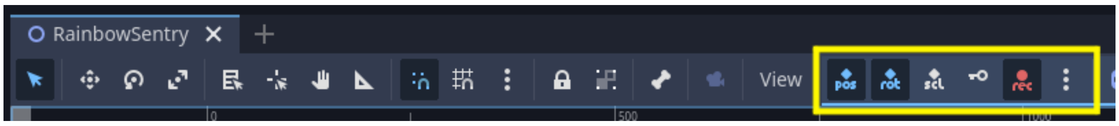


Figure 3.28 Animation-related buttons on the toolbar when the Animation window is active

By toggling the "auto insert" (rec) option on, keyframes will be added to the animation automatically every time you adjust a Node's position, rotation or scale directly in the editor. You can toggle which of these three properties to record this way in the toolbar. Note that the Tracks for position, rotation and scale each need to be added manually first.

In the example project, this feature was extensively used during the animation of monster models, as it allowed for repositioning and fine-tuning the transforms directly on the model.

Aside from properties, there are also several other things an animation can animate. These can be seen when adding a new track using the "Add Track" button, One notable among these, is the "Call Method" Track, which allows you to call any Node's method during any point in the animation, including methods with arguments as you can define them as parameters in the keyframe.

This ability to animate any property, as well as call methods makes AnimationPlayers incredibly flexible, as they could even be used to implement game logic. However, this is not advisable, as it could make debugging difficult by intertwining logic between code and animations.

3.5.3.2 Tween

In cases where an animation needs to animate through values that are not known beforehand, but only at runtime, an AnimationPlayer Node is not enough, as the values of the keyframes need to be specified in the animation.

In these situations, a Tween should be used instead. A Tween isn't a Node, but rather an object, that can be used to animate properties by interpolating from the current value to another, this is called "tweening". These values are passed to the Tween at runtime through code.

To create a Tween, the CreateTween() method can be used, which is accessible from any Node class, or the SceneTree instance.

Code listing 3.5 Creating a Tween instance

```
Tween tween = CreateTween();
```

The Tween's actions can then be defined by appending instructions to them using the following methods:

TweenProperty(): tweens a Node's property to some new value, over a specified duration. The property has to be specified using its string name, which you can see by hovering over the property in the Inspector window. In case of variables consisting of multiple values, such as "position" which is a Vector2 value, you can specify the x or y coordinate like "position:x" or "position:y" respectively.

Code listing 3.6 Tweening the x position of a Node2D to move 100px to the right over 0.5 seconds

```
tween.TweenProperty(someNode2d, "position:x", this.Position.X + 100, 0.5);
```

TweenInterval(): pauses the Tween Node for the specified amount of seconds. Can be used for example to tween a property, wait a few seconds, then continue tweening another property.

TweenCallback(): calls a method. The method needs to be passed as an argument of Godot's Callable type.

Code listing 3.7 Using TweenCallback for a method without any arguments, and a method that requires arguments

```
...
public void SomeMethod() {...}
public void MethodWithArg(int someValue) {...}
public void StartTween()
{
    Tween tween = CreateTween();
    tween.TweenCallback(Callable.From(SomeMethod));
    // Create lambda function that calls method with a specific argument.
    tween.TweenCallback(Callable.From(() => MethodWithArg(5)));
}
```

TweenMethod(): calls a method multiplpe times over a duration. Unlike TweenCallback(), which simply calls the method once, TweenMethod() could be said to interpolate the method call from one argument value to another.

Code listing 3.8 Using TweenMethod to call a method with arguments ranging from 0 to 10 over a duration of 0.5 seconds

```
...
public void MethodWithArg(int someValue) {...}
public void StartTween()
{
    Tween tween = CreateTween();
    tween.TweenMethod(Callable.From( (int arg) => MethodWithArg(arg)),
                                     0, 10, 0.5);
}
```

By default, all of the tweens are performed sequentially. To perform them simultaneously, the SetParallel() method can be used with a bool argument to enable or disable parallel running. Enabling it makes any tween appended afterwards perform simultaneously, although tweens that were appended before will still activate sequentially.

A Tween will start running as soon as it is created. If you want to create a Tween and define its actions without immediately performing them, use the Stop() and Play() methods.

3.5.4 AudioStreamPlayer

AudioStreamPlayer is the most standard Node used to play sound files. AudioStreamPlayer2D and AudioStreamPlayer3D are also available, which play spatial audio, i.e. audio that pans from left to right and gets quieter the further it is from the listener.

3.5.5 AudioStreamPlayer properties

Some of AudioStreamPlayer's properties include:

stream: the audio file to be played.

volume_db: the volume of the audio.

pitch_scale: changes the pitch of the audio. This will change the tempo as well, i.e. faster tempo and higher pitch, or slower tempo and lower pitch.

autoplay: play the audio immediately upon instantation of the AudioStreamPlayer Node.

bus: the Audio Bus that the audio is routed through.

Note that the AudioStreamPlayer has no property for looping the audio. This is because this set on the audio resource itself: the AudioStream. On the AudioStream file, you can also define where the loop begins and ends.

3.5.6 Audio Bus

The Audio Bus window is located in the bottom window, under the Audio tab. All audio is played through one of the buses, as specified by its "bus" property.

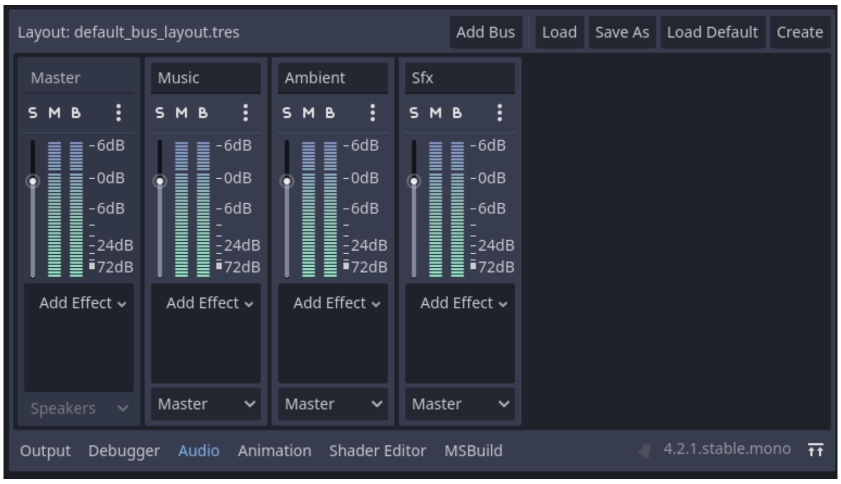


Figure 3.29 The Audio Bus window

A bus modifies all audio routed through it. It can control volume or add effects such as distortion, reverb, delay, etc. A bus can then be routed to any of the buses to the left of it to chain their effects. Finally all audio is eventually routed to the Master bus, which can be used to control overall volume, or apply effects globally.

In the example project, the Audio Bus is used to separate the audio into three buses: Music, Ambient and Sfx. With this design, it is then possible to control the volume of various sounds separately, such as for audio volume settings.

3.5.7 Nodes Summary

This ends the section on Nodes. As you can see, there many Nodes designed for different use cases. This section has only gone over Nodes that are most prominently used in the example project, with only brief explanations of some of their more common functions.

To summarize: Nodes are the basic building block in Godot, and their behaviour is implemented and extended by attaching scripts to them that inherit from their class type, which allows you to access, manipulate and use their properties and methods.

3.6 Scenes

When assembled together, a Node tree which forms one coherent logical whole is called a Scene. For example, a collection of Nodes that implements the player character could be considered a "Player Character" Scene.

Scenes are somewhat abstract when it comes to their use: they can be used as both objects and levels. [6] Godot does not make a distinction between these two, as they are ultimately just a collection of Nodes.

A Scene is saved to the project FileSystem as a .tscn file. It can then be placed inside other Scenes, or instantiated at runtime.

Scenes themselves can also be inherited, similar to the way a class would be. An inherited scene can overwrite its parents' fields, be extended by adding new Nodes to the tree, etc. [8]

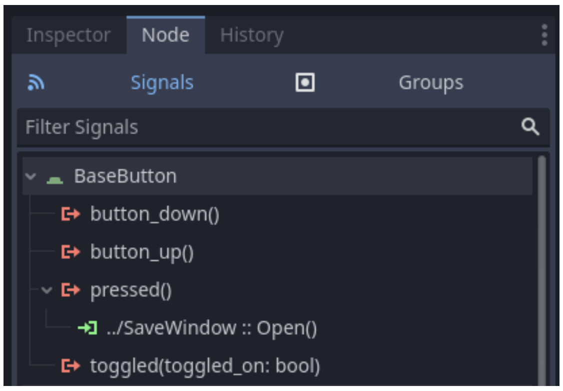
3.7 Signals

Another key concept in Godot are Signals. Signals implement the Observer Pattern, which you can read about in more detail in Game Programming Patterns by Robert Nystrom. [9]

Simply put, a "subject" can emit a Signal, which "observers" can subscribe to, so that they are notified whenever a Signal is emitted and perform some operation. This is useful because it keeps the code decoupled: the subject doesn't know about the observer. [9] A subject simply emits signals, and lets any interested observers handle the rest. This allows Nodes to communicate and interact with each other without being hard-wired through code.

In Godot, each Node has many built-in Signals that they emit at various times. By connecting these Signals to methods, you can further extend a Node's behavior in response to certain events. Additionally, you can connect, disconnect or define new signals through code, which is further discussed in section 3.8.3.2.

To connect a Signal to a method from the editor, switch to the "Node" tab in the Inspector window which contains all of the Node's signals. Signals can also contain values. They will be assigned to the method's arguments in the order they are declared in. A Signal should only be connected to a method when it provides all of its necessary arguments.



BaseButton
- button_down()
- button_up()
- pressed()
  - ../SaveWindow :: Open()
- toggled(toggled_on: bool)

Figure 3.30 The Signals window

Double-click a Signal to connect it to a method. First select the Node, and then type in the method name, or use the "Pick" button which lists the Node's methods. By toggling on the "Advanced" option, you can also bind additional arguments to when the method is called by the Signal. This can be used to bind a Signal with no values to a method that requires arguments.

A good general practice in Godot is for a parent Node to call a child Node's method directly by getting a reference to it using GetNode() and travel downwards inside of its own Node tree. This is because the parent Node will generally be aware of its internal composition and the child Nodes it is composed of.

When a child Node wants to access its parent Node, it should instead emit a Signal, which is then connected to the parent. This allows the child Node to be reused in multiple places, as it will be able to be placed inside any Node, with the parent Node deciding itself how it uses the

Signal. If one were to hard-wire the method call instead, the child Node would not be reusable, as it would have to depend on its parent being of a specific class, and for it to contain a specific method.

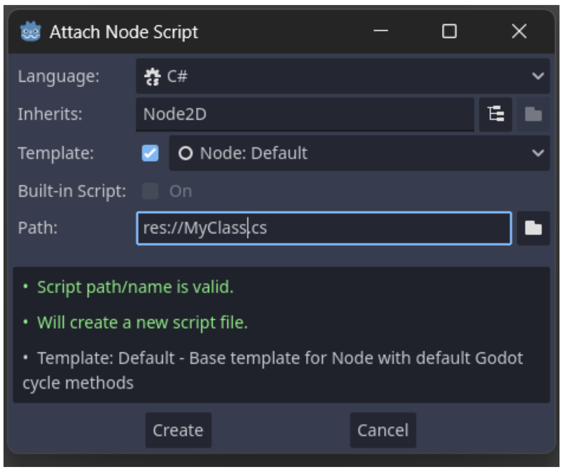
3.8 Scripting

In Godot, scripts are attached to Nodes to implement game behavior. Scripts attached to nodes should inherit from the Node's class, which provides methods and properties that allow you to make use of the Node's functions and behavior.

3.8.1 Attaching scripts and manipulating them in the editor

To attach a script to a Node, right-click it in the Scene window and select Attach Script...

As a project can mix several programming languages, the dialog that appears offers a choice in which language to use for this particular script, and several other fields, including the directory path where the script will be stored in the FileSystem.



Attach Node Script

Language: C#
Inherits: Node2D
Template: ✓ Node: Default
Built-in Script: On
Path: res://MyClass.cs

- Script path/name is valid.
- Will create a new script file.
- Template: Default - Base template for Node with default Godot cycle methods

Figure 3.31 Window dialog for attaching a script to a Node

It should be noted that all classes inheriting from any one of Godot's classes have to be marked as partial. This is because Godot uses Source Generators to facilitate C# interaction with the engine.[10]

You can open a script file by double-clicking it in the FileSystem, or through the Scene window by clicking the Script icon to the right of the corresponding Node.

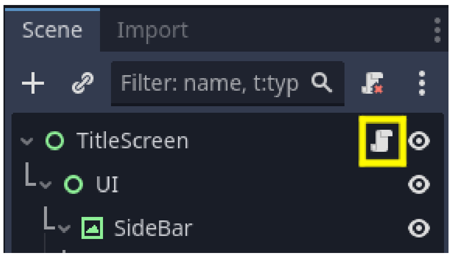


Figure 3.32 Opening a script through a Node in the Scene window

3.8.2 Scripting Basics

The various Nodes available in Godot have many fields, methods and signals that you can use to implement or extend behaviour. Because of the inheritance-based design of Nodes, some Node types can contain many methods accumulated through inheritance.

When using a Node, refer to its page in the Godot documentation, which lists all of its fields, methods and signals, as well as the Nodes it inherits from, meaning they contain further methods fields and signals you have access to. In the Godot Documentation, all attributes and methods are written in snake case i.e. “snake_case_example”. However in C#, all of them these fields were rewritten to the C#-standard pascal case, i.e. “PascalCaseExample”.

The following sections will go over some more commonly used methods that belong to the Node class, which is the base class every other Node inherits from, meaning all of these methods are accessible from any and all types of Nodes.

3.8.2.1 Override Methods

The methods in this section are marked as virtual, and are meant to be overridden to implement behavior at various times.

_Ready(): this method is called whenever a Node and its children enter the scene tree.

It is called from bottom up, i.e. it is first called on all of the Node’s children before it is called on the Node itself. Because of this, when a Node’s _Ready() method is called, you can expect all of its children to have already called their _Ready() methods and initialized accordingly.

Additionally _Ready() is called only once, this means that if the Node is removed from the tree, and later re-added, the _Ready() method will not be called again.

This method is typically used to initialize variables containing references to a Node’s children, so they can later be accessed through the variable instead of having to search through the Node tree every time it needs to be accessed.

Code listing 3.9 Using the _Ready() method to initialize variables referencing child Nodes

```
public override void _Ready()
{
    icon = GetNode<TextureRect>("Icon");
    label = GetNode<RichTextLabel>("Label");
    animationPlayer = GetNode<AnimationPlayer>("AnimationPlayer");
}
```

_EnterTree(): similar to _Ready(), this method is called on a Node and its children when it enters the scene tree.

_EnterTree() is called before _Ready(), and it is called from top down, i.e. it is first called on the Node itself, and then on its children.

Unlike _Ready(), _EnterTree() can also be called multiple times. This means that when a Node is removed and later re-added to the tree, _EnterTree() will be called again.

_Process(): this method is called on every frame.

One should not rely on the consistency of _Process() calls, as it is called on every frame, which is not constant and changes with the framerate. It is important to avoid framerate-dependent code, which would cause a game to behave differently on different devices with different framerates. This can manifest in ways such as faster movement speed at higher framerates.

To account for variable framerate, the "delta" value is received as an argument, which contains the time elapsed in seconds since the previous _Process() was called.

■ **Code listing 3.10** Using _Process to move a Node2D to the right every frame

```
public override void _Process(double delta)
{
    this.Position += new Vector2(100 * (float)delta, 0);
}
```

In code 3.10, the Node is moved by $100 * delta$ pixels to the right every frame, i.e it will have travelled to the right by 100 pixels after 1 second has passed. If one were to move the Node without accounting for variable framerates, i.e. at a constant 100 pixels every frame without delta: the Node will have moved 3000 pixels after 1 second on a 30 fps device, or 6000 pixels after 1 second on a 60 fps device.

_PhysicsProcess(): is called on every physics processing frame.

With a similar use as _Process(), the difference in _PhysicsProcess() is that it ensures that it is called consistently, on every "physics frame". In other words, the "delta" value should be consistent.

As the name suggests _PhysicsProcess() is intended to be used to implement logic related to physics systems, as those require consistent operations.

_Input(): this method is received when the Node receives user input. The input is received as an argument of the InputEvent type. You can then parse the InputEvent and determine which action to perform. Input handling is discussed in more detail in 3.12.

3.8.2.2 Node Methods

The methods in this section can be called on any Node.

GetNode(): returns the Node specified by a Node path. Node paths are similar to a file system structure, where they contain a sequence of Node names, separated by a "/".

In C# the type of the returned Node also needs to be specified, this can be achieved using typecasting, or by using GetNode() as a generic method as is seen below. Because of the inheritance-based design of Nodes, the accessed Node can be cast into any of its supertypes, i.e. when accessing a Sprite2D Node, one could cast it as Node2D instead, if only Node2D properties are needed.

For the following code examples, assume the Node tree at 3.33, with the GetNode() method being called from the Node named "UI".

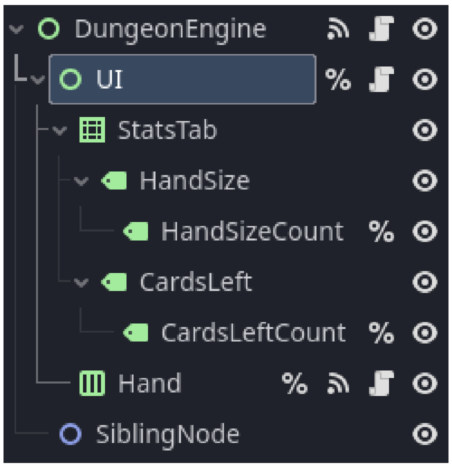


Figure 3.33 Node Tree of the code used in GetNode() examples

From the script attached to the "UI" Node, you could then access the HandSizeCount using the following code:

Code listing 3.11 Accessing a Node using GetNode, example 1

```
GetNode<Label>("StatsTab/HandSize/HandSizeCount");
```

Or access the "Hand" Node using:

Code listing 3.12 Accessing a Node using GetNode, example 2

```
GetNode<DungeonHandUI>("Hand");
```

In the above code, DungeonHandUI is the class of the script attached to the "Hand".

You can can also access the parent Node using the ".." string. However, this is inadvisable in most cases, as it hard-wires the child Node to its parent. As discussed in section 3.7, prefer using Signals instead in such cases.

Code listing 3.13 Accessing a Node using GetNode, example 2

```
GetNode<Node2D>("../SiblingNode");
```

All of the above calls use relative paths, i.e. paths that begin at the Node the method is called in. Using the "/root/" prefix, one can specify a Node using an absolute path, which begins at the root of the entire SceneTree that the game runs on. This can be used to access Autoload Nodes which are discussed in more detail in section 3.13.

When designing a Scene, it's possible that some Nodes will be moved around frequently, thus breaking scripts as they change their Node paths. In such cases, the "Scene Unique Node" feature can be used, which allows the Node to be specified using simply its name without having to provide the entire path to it. To mark a Node as Scene Unique, right-click it in the Scene window and select "Access as Unique Name", which will mark it with a "%" sign. The Node can then be accessed from any script using its name prefixed by a "%":

Code listing 3.14 Accessing a Scene Unique Node using GetNode

```
GetNode<Label>("%HandSizeCount");
```

For this feature to function, the name of the Node needs to be unique to avoid ambiguity. Note that the scope of a Scene Unique Node is only limited to the Scene that they are in. For example, if a Node with a Unique Node is saved as a Scene, i.e. a .tscn file, and then later placed inside a different Scene, it will not be accessible using “%” from this new Scene.

GetNodeOrNull(): functions similarly to GetNode(), however, GetNode() will throw an error if the Node is not found at the Node path. GetNodeOrNull() will instead return null, so it can be used in cases where a Node’s existence is unclear.

GetChild(): returns a child Node using its numeric index.

GetChildren(): returns all of this Node’s children as an array. Order is guaranteed to be consistent, as it will contain all of the Nodes children in order from top to bottom. Uses Godot’s array class,
Godot.Collections.Array, rather than a C# array.

QueueFree() and Free(): Free() essentially functions as a destructor for Godot’s Object type, which Node inherits from. It will remove the Node from the Scene Tree and free it from memory, as well as its children.

QueueFree() functions the same as Free, however, it waits until the end of the current frame before performing the freeing. This ensures that all Nodes have finished processing their operations, making it a safer option than Free() which does so immediately. Unless necessary, prefer using QueueFree() over Free() in most cases.

AddChild(): adds a Node as its child. Will fail if the Node is already a child of another Node.

RemoveChild(): removes a child Node from this Node.

It important to note that this will remove the Node from the SceneTree. This will cause the Node to become what is called an “Orphan Node”. An Orphan Node will not be automatically freed from memory, which can cause memory leaks. This means that it will need to be freed manually in the script, unless it is added back to the SceneTree.

If you want to simply delete a Node, use QueueFree() or Free(), which will both remove it from the SceneTree and free it from memory.

A typical usage scenario of RemoveChild() would be to remove a Node using RemoveChild(), and subsequently add the Node back to a different parent using AddChild().

MoveChild(): moves a child Node to the specified index, where the topmost child is at index 0 and the bottom-most child is ChildCount()-1.

ChildCount(): returns the amount of children the Node has.

3.8.3 Scripting Signals

Signals are one of the aspects of Godot that work differently in C# compared to GDScript. While Signals in C# can still be used and manipulated using similar methods as in GDScript, they are also available as C# events, which provide more type-safety and more C#-like syntax and conventions. All of the differences between GDScript and C# handling of signals are covered in the “C# signals” page in the Godot Documentation. [11]

3.8.3.1 Connecting and Disconnecting Signals

As Signals are events, connecting and disconnecting a method to a Signal means to add it or remove it from the event’s invocation list. This can be done using “+=” and “-=”.

Code listing 3.15 Connecting a method to a Signal

```
private void SomeMethod() {}
private void ConnectSignal(Button buttonNode)
{
    // Connecting SomeMethod to Button's "Pressed" Signal.
    buttonNode.Pressed += SomeMethod;
    // Disconnecting.
    buttonNode.Pressed -= SomeMethod;
}
```

3.8.3.2 Custom Signals

Aside from using a Node's built-in Signals, you can define custom Signals using delegates marked with the "[Signal]" attribute.

Code listing 3.16 Defining a custom Signal

```
[Signal]
public delegate void CustomSignalEventHandler(int someValue);
```

Note that all custom Signal names have to end with "EventHandler".

A custom Signal can contain values that are then passed as arguments to connected methods in the order they are defined. These values can be of primitive data types, or Godot classes. This means user defined classes cannot be emitted by Signals through usual means, however, one can emit classes that inherit from Godot's Resource class, which are discussed in section 3.11.

3.8.3.3 Emitting Signals

Using the EmitSignal method, you can emit both built-in and custom Signals by providing their name as an argument, alongside the values that the Signal may contain.

Code listing 3.17 Emitting a signal alongside a an argument value

```
EmitSignal(SignalName.CustomSignal, 10);
```

When accessing a signal by name, rather than hard-coding in strings which are error-prone, use SignalName instead. In the above code example which emits the custom Signal defined in the code at 3.16, note that the Signal is referred to without the "EventHandler" prefix.

3.9 [Export] attribute

The [Export] attribute allows you to make a variable editable in the Inspector window, and then saved alongside the scene where it's edited, similar to other Node properties. This can be used to define and store data in the editor, instead of hard-coded in scripts.

Code listing 3.18 attribute] Marking a variable with the [Export] attribute

```
public partial class SkillBoxUI : Control
{
    [Export]
    private int index = 0;
    ...
```

Note that when adding a new [Export] variable,, the C# assemblies need to be rebuilt afterwards using the “Build” button at the top right, for the Inspector window to update and display the variable.

As the editor needs to be able to recognize the type of the exported variable to display it in the Inspector window, only primitive data types, and Godot types can be exported. This means that when exporting for example an array, the Godot.Collections.Array has to be used instead of C# collections. However, custom classes can be exported if they inherit from Godot’s “Resource” class, this is discussed in more detail in 3.11.

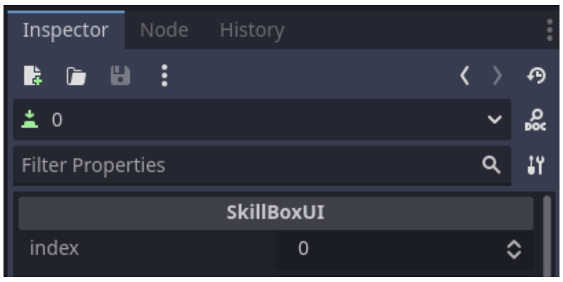


Figure 3.34 An [Export] variable appearing in the Inspector window

3.10 Resources

In Godot, Resource is a data container that provides data to Nodes. This means that nearly all assets in a Godot project inherit from Resource. Some examples include:

- Texture
- Script
- Font
- AudioStream

All Resources are passed by reference. You can create a shallow or deep copy using the Duplicate() method.

3.11 Custom Resources

Making extensive use of the [Export] feature, custom Resources are a feature that allow you to define and store classes as data. This data can be edited from the Inspector window and subsequently be saved as a .tres file, which can later be loaded during runtime.

To create a custom resource, simply create a class that inherits from Godot’s “Resource” class.

Code listing 3.19 Defining a custom Resource

```
[GlobalClass]
public partial class BattleSetup: Resource
{
    [Export]
    public Array<PackedScene> Monsters { get; set; }
    ...
```

Working with custom Resources somewhat similar to working with Nodes. You can edit a custom Resource's properties in the Inspector window, as well as make use of features such as Signals without having to create and instantiate an entire Node. Classes that inherit from Resource can also be passed to Signals as arguments when emitting them.

By marking a Resource class with the "[GlobalClass]" attribute, it will be added to the list of Resources seen in the editor. For example when assigning a Resource to a property in the Inspector window, it will now list the custom Resource alongside other built-in types.

3.12 Input

This section briefly explains reading of input whether it is keyboard, mouse, controller, gameplay or UI input.

3.12.1 Receiving input in Nodes

Player input: mouse, keyboard, or controller is received by Nodes in order from the bottom of the Node tree to the top. Nodes can consume these inputs by overriding methods such as _Input(), _UnhandledKeyInput() or _UnhandledInput().

_Input(): receives all input. This method should be used for application-wide inputs.

_UnhandledKeyInput(): receives keyboard input, but only if it wasn't consumed by some other Node. This method should be used for gameplay input.

_UnhandledInput(): receives input, but only if it wasn't consumed by some other Node. This method should be used for gameplay input.

Additionally, Control Nodes contain the _GuiInput() method, which should be used in place of the above ones, as it is intended for UI input and is handled before the _Unhandled methods.

An input can be consumed by Control Nodes such as Buttons to prevent them from propagating further through the Node tree. This can be used to for example prevent the player character from reading input and moving while menu navigation is occurring.

Code listing 3.20 Receiving input on a Control Node through _GuiInput()

```
public override void _GuiInput(InputEvent @event)
{
    // Determine what kind of input was received
    if(@event.IsActionPressed("ui_cancel"))
    {
        // Perform some logic.
        ...
        // Mark input as handled to prevent it from propagating.
        AcceptEvent();
    }
}
```

3.12.2 InputEvent

All of the Input methods contain the InputEvent argument, which holds information about the input, such as whether it is a keyboard or mouse input, and what specific input was performed. This is built around classes that inherit from InputEvent, such as InputKeyEvent or Input-MouseEvent. You can read more about the various types of input and their properties in the Godot documentation. [12]

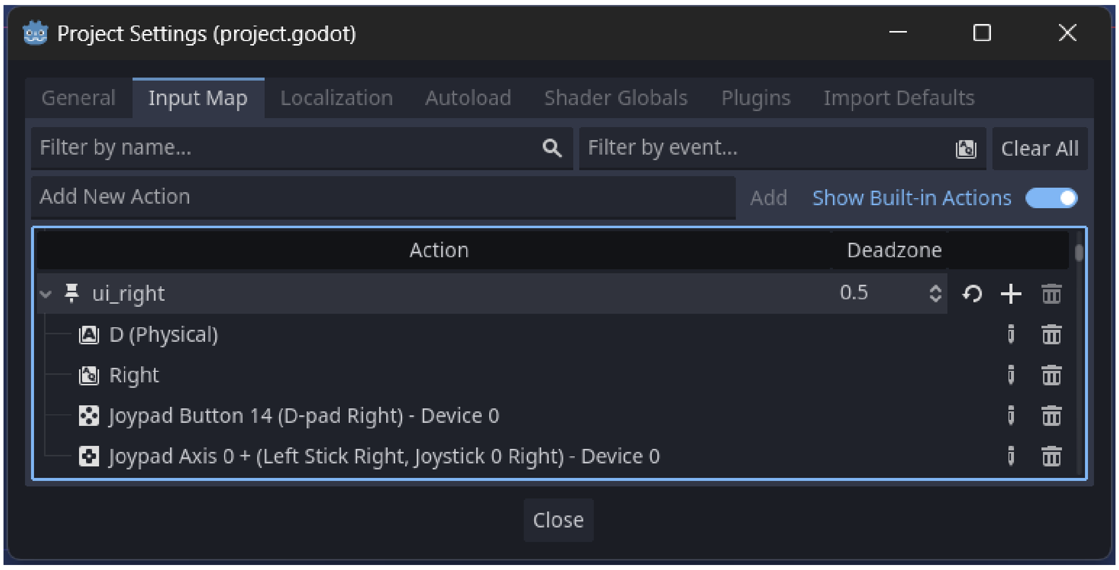
When receiving an input, you can check its type to see what kind of input it is and cast it to access information specific to the kind of input.

Code listing 3.21 Receiving mouse button input

```
public override void _Input(InputEvent @event)
{
    // Check if event is mouse click
    if (@event is InputEventMouseButton mouse)
    {
        // Check if left mouse button was released.
        if (mouse.ButtonIndex == MouseButton.Left && !mouse.Pressed)
        {
            ...
        }
    }
}
```

3.12.3 Input Map and Actions

When reading button input, for example during character movement, hard-coding the reading of specific keyboard keys into scripts is undesirable, as it makes it difficult to facilitate remapping the key or alternative input methods, such as controllers. The solution to this is Godot's "Action" feature. An Action is a grouping of keyboard, mouse and controller inputs that can be used to determine the nature of the input.



Action	Deadzone
ui_right	0.5
D (Physical)	
Right	
Joypad Button 14 (D-pad Right) - Device 0	
Joypad Axis 0 + (Left Stick Right, Joystick 0 Right) - Device 0	

Figure 3.35 The Input Map window

For example, the built-in Action "ui_right" contains the inputs from the keyboard right arrow key, a controller right D-pad press, as well as a controller analog joystick left turn. When receiving an InputEvent, it can then be queried to see if it is "ui_right" using InputEvent's IsAction(), IsActionPressed() or IsActionReleased() methods.

Code listing 3.22 Receiving input and parsing it using Action

```
public override void _UnhandledKeyInput(InputEvent @event)
{
    if (@event.IsActionReleased("ui_right"))
    {
        // Move right
        ...
    }
}
```

To manage Actions, in editor, navigate to Project > Project Settings > Input Map. Here, you can add or edit Actions and assign or remove inputs from them. Note that to show built-in Actions, such as the ui direction ones, "Show Built-in Actions" needs to be toggled on.

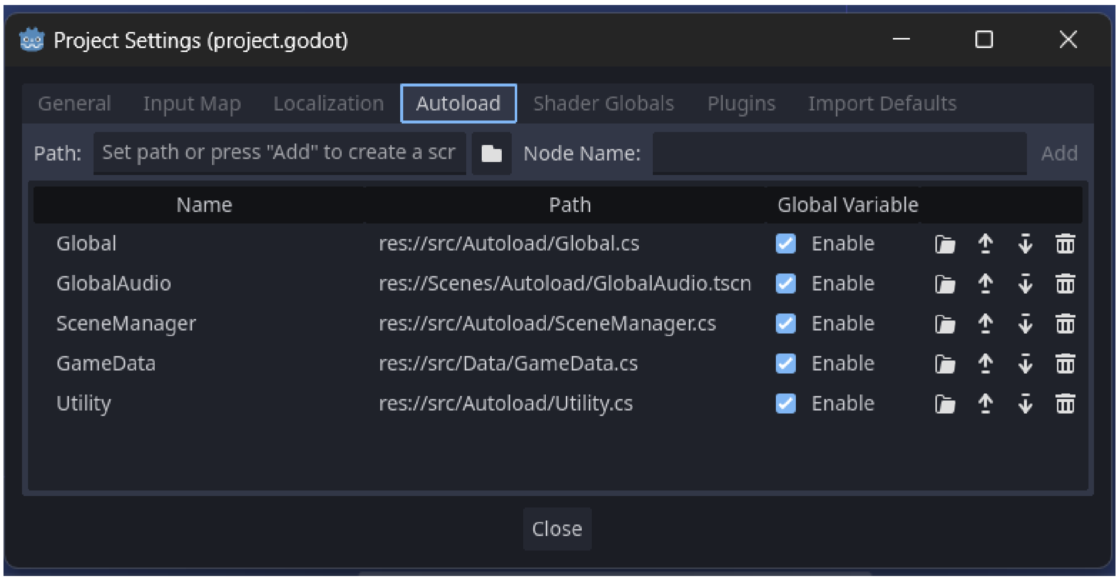
3.13 Autoload

Autoloads are nodes that are instantiated automatically at the start of the program, and persist over the entire lifetime, even when switching between scenes. This means they can be used to store data and methods that need to be accessible at all times from any Node.

Functionally, Autoloads are Godot's implementation of the Singleton pattern. You can read more about the Singleton pattern in "Game Programming Patterns" by Robert Nystrom. [13]

However, it should be noted that Autoloads aren't a true implementation of the Singleton design pattern, as they have some traits that Singletons shouldn't have, such as the ability to have more than one instance of an Autoload. [14]

To manage Autoloads, go to Project > Project Settings > Autoload, where you can add .tscn Scenes as Autoloads, or scripts that inherit from a Node type, which will get instanced as a Node of the type the script inherits from.



Name	Path	Global Variable
Global	res://src/Autoload/Global.cs	✓ Enable
GlobalAudio	res://Scenes/Autoload/GlobalAudio.tscn	✓ Enable
SceneManager	res://src/Autoload/SceneManager.cs	✓ Enable
GameData	res://src/Data/GameData.cs	✓ Enable
Utility	res://src/Autoload/Utility.cs	✓ Enable

Figure 3.36 The Autoload window

In GDScript, if "Global Variable" is enabled, the Autoload can be accessed globally using the name as if it were a static class. However, this is not the case for C#. As such, additional code has to be used if one wishes to achieve similar behaviour. In the example project, this can be seen in classes such as SceneManager:

Code listing 3.23 Making an Autoload Node accessible globally

```
public partial class SceneManager : Node
{
    ...
    private static SceneManager _instance;
    public static SceneManager Instance => _instance;

    public override void _EnterTree()
    {
        if (_instance != null) this.QueueFree();
        else _instance = this;
    }

    public static void StartBattle(BattleSetup setup)
    {
        _instance.CallDeferred(MethodName.DeferredStartBattle, setup);
    }
    ...
}
```

The above code creates a static variable "_instance" that can then be used to access the Node from any script using SceneManager.Instance, or through static methods that call then call the Instance themselves.

Alternatively, Autoload Nodes are added as children to the "/root/" path of the SceneTree, alongside the Main Scene. As such, they can be accessed using GetNode("/root/<Name>"). However, as the GetNode() method is only contained in classes that inherit from Node, this would make Autoloads inaccessible from classes that do not inherit from Node.

3.14 Assets and file paths

In Godot, there are two types of filepaths:

res:// , which is used to access project files.

user:// , which is used to access user data.

3.14.1 File paths at "res://"

All files contained in the folder where you created the project can be accessed by its pathname, beginning with "res://".

Any file you add to the project whose format is recognized, i.e Godot is capable of importing that file type, will automatically have a corresponding .import file created. Files not considered assets by Godot, such as .txt files, can still be used, refer to the "Exporting the game" section at 5.7 for more detail.

To load the corresponding asset from code, use the GD.Load() method.

Code listing 3.24 Loading a file resource from code

```
GD.Load<Texture>("res://Images/testImage.png")
```

It should be noted that this is not a real file system, as Godot converts all assets during export into the aforementioned .import files. The GD.Load() method then seeks out the corresponding .import file based on the file path that it receives.[15] That is to say, if one were to for example: attempt to open an image at "res://Images/testImage.png" using a File reading API, such as

C#'s File, or Godot's FileAccess, while both would function properly when playtesting from the editor, they would fail in the release build as the image is not truly stored at that path in the final executable.

One of the most common assets you will load are likely Nodes: Scenes, which are stored in .tscn files. These .tscn files are of the "PackedScene" class, which then have to be instantiated by calling the Instantiate() method.

Code listing 3.25 Loading a Scene file, instantiating it, and adding it to the SceneTree

```
SomeClass node = GD.Load<PackedScene>("res://Scenes/MyNode.tscn")
                    .Instantiate<SomeClass>();
someNode.AddChild(node);
```

3.14.2 File paths at "user://"

The "user://" prefix is an actual location on the user's file system, and can be used to store persistent user data such as save files and settings.

The actual file path used by "user://" is platform specific. Taken directly from the Godot Documentation, [16] these paths are:

Windows: %APPDATA%\Godot\app_userdata\[project_name]

macOS: ~/Library/Application Support/Godot/app_userdata/[project_name]

Linux: ~/.local/share/godot/app_userdata/[project_name]

When using Godot's FileAccess API to manipulate files, it will automatically convert the "user://" to the appropriate file path in the actual file system. To get the OS file path from Godot's local "user://" one, use the ProjectSettings.GlobalizePath() method.

Code listing 3.26 Converting Godot's "local" path to the OS path

```
ProjectSettings.GlobalizePath("user://settings.cfg")
```

In our example project, on Windows, the code at 3.26 would return:

"C:/Users/[username]/AppData/Roaming/RPG_project/settings.cfg"

This can be useful for example in cases where C#'s File were to be used, as simply passing it a path beginning with "user://", a Godot specific construct, wouldn't work.

Chapter 4

Design Architecture of the Game

This chapter will discuss the design architecture of the game by going over the project folder structure, the Node Scenes used in the game, as well as some of the data structures.

4.1 Data Structures

The game makes use of Godot's custom Resource feature to define Resource files that hold data for various aspects of the game.

BattleActorStats: Resource that contains the various stats and attributes of both player characters and monsters. For player characters, the subtyped CharacterStats is used, and for monster characters, the subtyped MonsterStats is used.

BattleSkillData: Resource that contains a single skill. Each skill is its own class that inherits from BattleSkillData, these are then assigned to BattleActorStats.

DungeonCard: Resource that contains a card during a dungeon. Depending on the type of card, DungeonCard contains further data.

In case of the MonsterCard, which represents a card that activates a battle, it contains the setup of enemies appearing in the battle.

BattleSetup: Resource that contains an array of monsters.

DungeonSetup: Resource that defines the deck of a Dungeon: the cards that can be drawn, as well as the boss card that is drawn last. It also defines additional parameters such as the Dungeon name, hand size and music played.

4.2 Project File Structure

This section will first outline the folder structure of the project root, and then further describe the subfolders contained within some of the more complex folders.

4.2.1 Root folder

The project root has the following folder structure:

`Audio`	audio files
`Font`	all data related to text
`Images`	image files
`Localization`	csv files used for localization
`Objects`	tscn files representing objects
`Resources`	Resource tres files
`Scenes`	tscn files representing levels/main scenes
`ShaderMaterial`	shader materials
`src`	scripts

4.2.2 Project folder: Images

This folder contains all of the images used in the project.

`Images`	
└ `CharacterIcon`	character icon in battles
└ `CharacterModel`	character model images
└ `Monster`	monster model images
└ `MonsterIllustration`	illustrations for "Analyze" skill
└ `UI`	UI elements
└ `VFX`	visual effects

4.2.3 Project folder: Objects

This folder contains .tscn files: Scenes, that represent objects. For Scenes that represent levels and are used as Main Scenes, they are contained in the "res:/Scenes" folder instead.

`Objects`	
└ `Battle`	
└ `Monster`	monster objects
└ `CharacterModel`	character models
└ `UI`	various UI elements
└ `Battle`	battle UI elements
└ `SkillCustomWindow`	custom windows for skills
└ `Dungeon`	dungeon UI elements
└ `Window`	general window elements
└ `VFX`	one-time animation vfx objects

4.2.4 Project folder: Resources

This folder contains Resource files used for data such as characters, skills and dungeon information.

`Resources`	
└ `Dungeon`	dungeon-related resources
└ `Card`	DungeonCard resources
└ `Setup`	DungeonSetup resources
└ `Skills`	BattleSkill resources
└ `Stats`	BattleActorStats resources

4.2.5 Project folder: src

This folder contains the scripts used in the project.

```
src
├── Animation ........ visuals and animation
├── Audio ........ sound-related scripts
├── Autoload ........ scripts used as Autoloads
├── Battle ........ battle-related scripts
├── Data ........ all custom resource scripts, such as BattleActorStats and DungeonCard
├── Dungeon ........ DungeonEngine and related scripts
├── Enum ........ enum scripts
├── Misc ........ miscellaneous utility scripts
├── NodeExtensions ........ node scripts that only add minor extra functionality
├── Serialization ........ game save and load scripts
├── Shaders ........ shader code
└── UI ........ UI-related scripts
```

4.3 Node Architecture

The Godot Node Tree begins with the "root" Node, which can then contain multiple Nodes as its children. By using Autoloads, several other Nodes are instanced as siblings in the root Node.

In the example project, the root Node contains the following children:

- Global
- GlobalAudio
- SceneManager
- GameData
- Utility

In addition to the above, there will also always be a "Main Scene" Node present, which is the current Scene displayed to the player. All of these Scenes implement the "IMainScene" interface.

- TitleScreen
- DungeonEngine
- BattleEngine

4.3.1 Autoload Nodes

Below are brief descriptions of each Autoload Node's function:

Global: tracks game-time and facilitates loading and saving of games.

GlobalAudio: plays music and sound effects. Used to play audio that persists in-between scenes, as audio Nodes contained within Scenes that are removed would abruptly stop playing. GlobalAudio also allows for blending music in and out.

SceneManager: manages swapping between Main Scene Nodes. Can swap using a .tscn path passed as an argument, or start a dungeon or a battle, then subsequently return to the previous Scene after finishing.

GameData: contains all persistent data in the game: CharacterStats of each player character, as well as the Party and Bench layout.

Utility: contains helper functions. Rather than being a simple static class, it is implemented as a Node and instantiated as an Autoload to give it access to Godot methods that are only available from inside the SceneTree.

4.3.2 Main Scene Nodes

Unlike Autoload Nodes, which are typically single Nodes with no children in the example project, Main Scene Nodes feature more complex Node trees. This section will therefore dissect their structure in more detail.

4.3.2.1 Main Scene: TitleScreen

The initial Main Scene that is loaded upon launching the game.

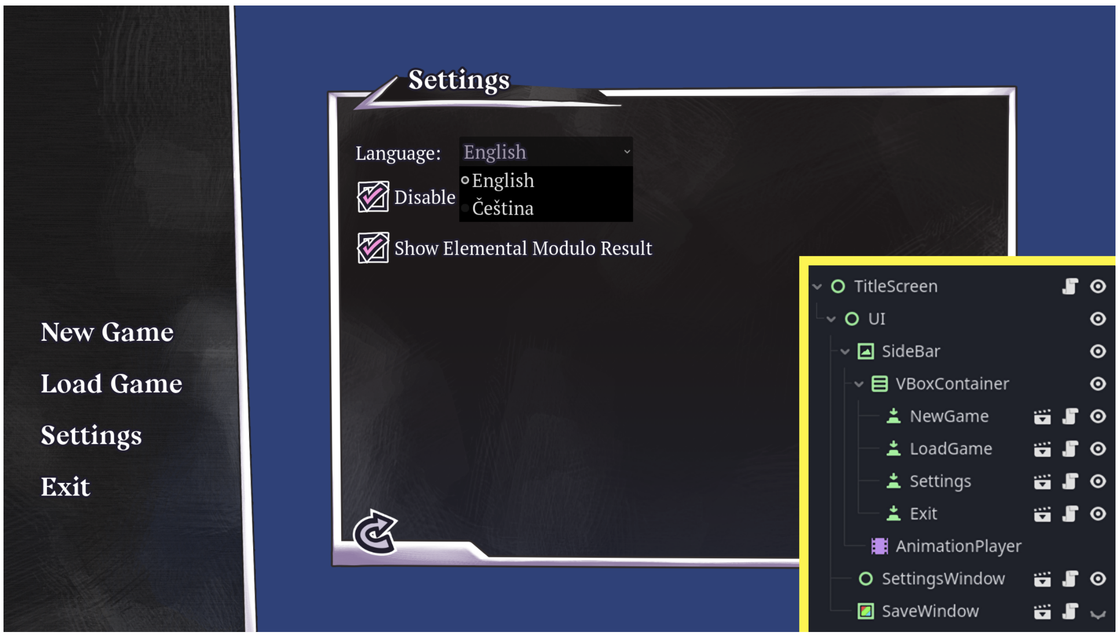


Figure 4.1 Screenshot of the TitleScreen Scene, alongside its Node tree

From this Main Screen, using the sidebar, one can: start a new game, load an existing game, configure settings, or exit the game.

4.3.2.2 Main Scene: DungeonEngine

A Main Scene representing Dungeon traversal. It is transitioned to from TitleScreen, after starting a new game or loading a previous save.

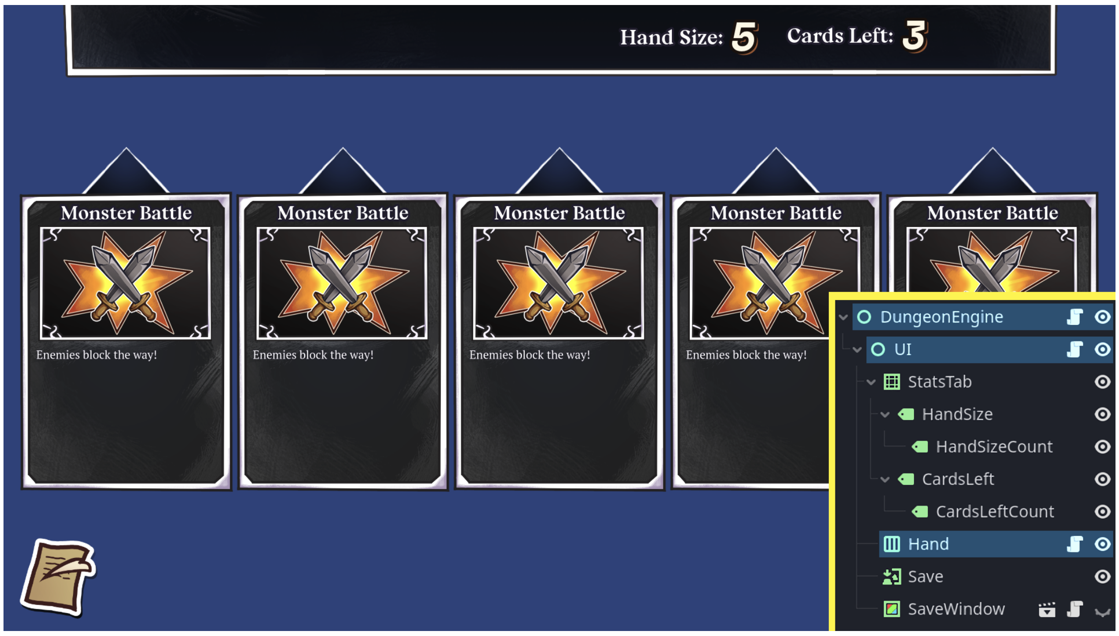


Figure 4.2 Screenshot of the DungeonEngine Scene, alongside its Node tree

DungeonEngine: the root Node that performs all the main logic of dungeon traversal. It holds information such as the DungeonSetup currently being used, the remaining deck, the current cards in hand, as well as the hand size.

UI: manages all UI elements using the DungeonUI script attached to it. DungeonEngine passes all UI-related calls to DungeonUI, which then updates the UI using the data it receives. This way, the data and visuals of DungeonEngine are kept separate.

Hand: the UI element that contains the cards held in the player's hand using the DungeonHandUI script attached to it. When DungeonUI needs to update the hand, it passes the call to DungeonHandUI, which handles updating each card to match the data that was received, and does so while playing appropriate animations for adding new cards.

The cards held in the player's hand are represented using the DungeonCardUI Scene loaded from "res://Objects/UI/Dungeon/DungeonCard.tscn".

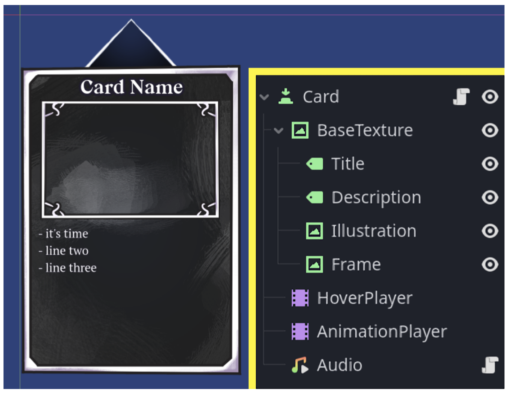
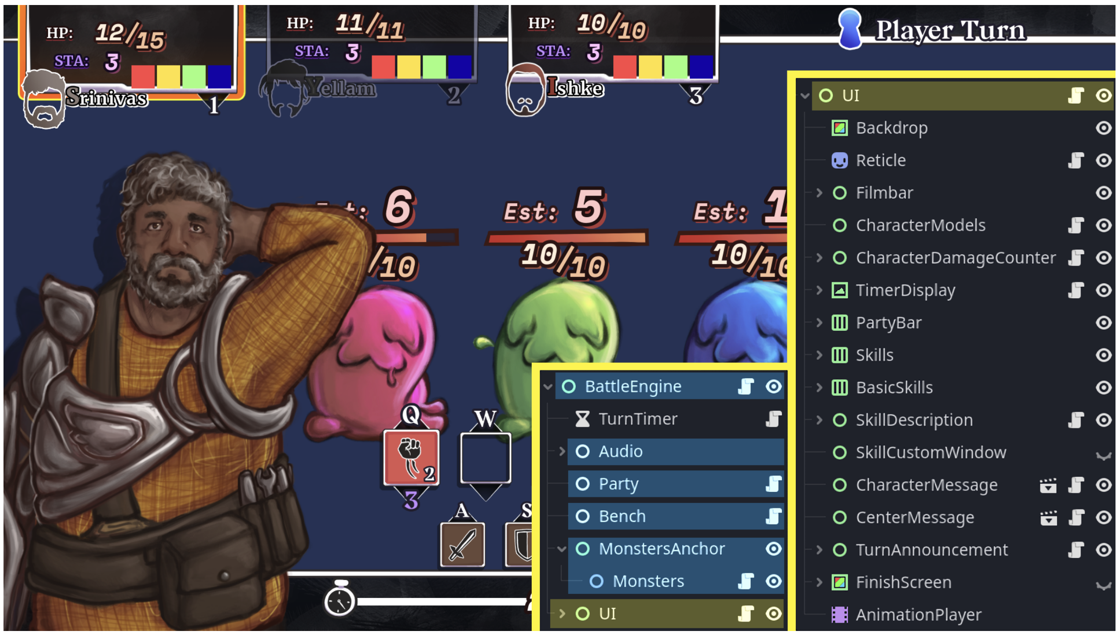


Figure 4.3 Screenshot of the DungeonCard.tscn Scene, alongside its Node tree

4.3.2.3 Main Scene: BattleEngine

A Main Scene representing a battle. It is transitioned to after selecting a Monster Battle card in DungeonEngine. After the battle is over, transitions back over to DungeonEngine.



■ **Figure 4.4** Screenshot of the BattleEngine Scene, alongside its Node tree

Audio: contains child AudioStreamPlayer Nodes that contain various sounds.

Party: contains the player's party as child Nodes, each representing one character.

While it is unnecessary to represent character data as Nodes (they could be more intuitively stored as simple data structures in an array), it is done so for consistency, as player characters and enemy units are both represented by the same class: BattleActor. Enemy units are required to be Nodes to more efficiently facilitate interaction between their battle data and the models on-screen, thus making BattleActor a Node, and consequently making player characters Nodes as well.

Bench: contains the player's bench as child Nodes, each representing one character.

MonstersAnchor/Monsters: "MonstersAnchor" is a Control Node anchored to the center of the screen to facilitate responsive design.

As its child, it then contains the "Monsters" Node, which contains all the enemies that are represented by Node Scenes containing both the model and battle data of each respective enemy.

The Monsters Node is a Node2D with the MonsterRack script attached to it, which allows it to be scrolled to the left and right by the player to better view the enemies, hence the necessity of splitting the Nodes into MonsterAnchor and Monsters.

UI: : contains, manages and updates all the UI elements using the BattleUI script attached to it. The root BattleEngine Node interacts with the BattleUI by passing the corresponding data to it, and lets BattleUI handle the rest. This way, the data contained in BattleEngine and the visual elements displayed by BattleUI are kept separate.

The Control Nodes contained within the UI Node all fulfill various functions, from simple things such as adding to the visuals without any further need to interact with them, to more complex ones such as displaying character stats, skills, etc. Such Nodes have their own scripts attached to them that handle the specific function, allowing the BattleUI Node to further delegate the UI update call.

Chapter 5

Implementing the Game

In this chapter, we will go over the actual game development process of the example project, including code architecture and implementation, creation of visuals, etc. All the project files of the example project are available on the GitHub repository. [17]

This chapter assumes basic knowledge in programming concepts such as variables, classes, class methods and polymorphism. Surface-level C# knowledge, i.e. basic syntax and constructs, is also expected.

5.1 Sprites and Visuals

Since the project is a 2D game, most of the game's visuals will come in the form of 2D images: sprites. In the example project, there are two types of Nodes used to display images: Sprite2D and various Control Nodes.

5.1.1 Textures and Sprite data

Godot supports many image formats, the extent of which you can see in the Godot Documentation on importing images. [18]

The format used primarily throughout the project is PNG. These are simply assigned directly to Sprite2D Nodes as textures to display them in the game.

5.1.2 Character and Monster Models

These are the main visual component of the game's presentation, as they represent all of the units on the battlefield and feature the most art assets and animation.

Both Character and Monster models consist of several pieces of Sprite2D, grouped as children under a CanvasGroup node to make use of its benefits as discussed in 3.5.1.3.

5.1.2.1 Character Models: Visual

The Character Model consists of the following pieces:

- Body
- Head
- Eye
- Eyebrows
- Mouth

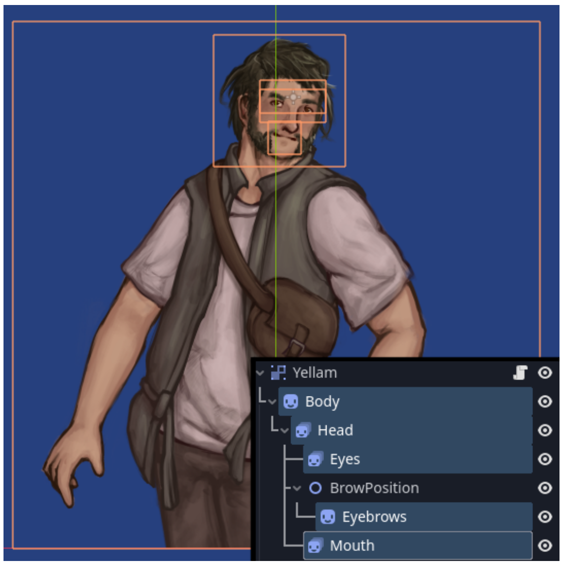


Figure 5.1 The Node structure of a Character Model

By separating these, we can change each part independently from each other, such as making the eyes blink, changing the facial expressions by switching eyebrows and mouths, etc.

The Eyebrows Node is contained within a Node2D called “BrowPosition”. This is because the brow is animated by shifting its position down and back up slightly every time the Character Model blinks, which requires some way of memorizing the original position. Using this structure, the BrowPosition Node contains the actual position of the eyebrows, and the animation of the “Eyebrows” Node is done by shifting its y position from 0px to 2px and then back to 0px.

When it comes to animation, the character models contain two AnimationPlayers:

BreatheAnimator: scales the Body Node along its y axis up and down repeatedly. Because the pivot point of the Body Node is set to the center bottom of the sprite, this makes the entire model, alongside the child Nodes, stretch upwards and then back down, to simulate a simple animation for breathing.

BlinkAnimator: contains a very short animation for a single blink.

The actual blinking and activation of the AnimationPlayer is then handled through code using the CharacterModel script, which randomizes the interval between each blink for a more natural animation than if it was a simple repetitive loop.

Code listing 5.1 Blinking handled through code in CharacterModel

```
public partial class CharacterModel : Node2D
{
    ...
    [Export]
    public Curve blinkCurve;

    public void RefreshBlinkTimer()
    {
        blinkTimer.Start(blinkCurve.Sample(GD.Randf()));
    }

    public void Blink()
    {
        blinkAnimator.Play("Blink");
        RefreshBlinkTimer();
    }
}
```

To track the time between each interval, the script uses a Timer Node that has its “timeout” signal connected to the Blink() method, which plays the animation and subsequently generates a random interval for the next blink.

When randomizing the blinking interval, a random float number from 0 to 1 is generated, which is then mapped to a curve that returns the actual interval in seconds. The curve is represented by Godot’s Curve class, which can be defined in the editor through the Inspector window.

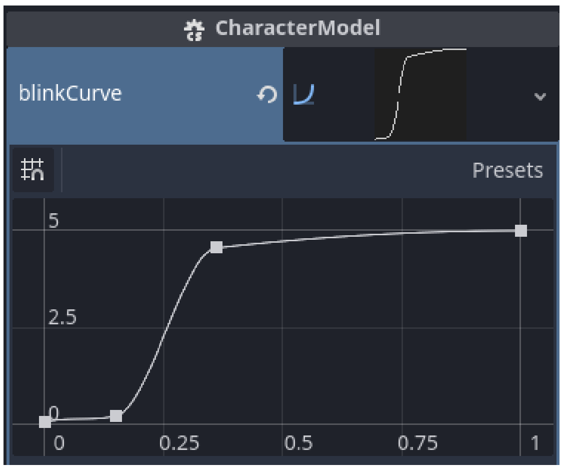


Figure 5.2 The blinking curve defined in the Inspector window

By defining a curve that contains low values at the beginning, before exponentially rising, the resulting blinking intervals tend to be longer at around 4 to 5 seconds, with the occasional rapid blinking.

Because the curve is an [Export] variable that is defined in the editor, this means that different curves can be assigned to various character models to create differing blinking patterns. For example, for a character in distress, a lower curve could be assigned to create more frequent or sporadic blinking.

5.1.2.2 Monster Models: Visual

The monster models in the game are 2D sprites animated using a technique called “cutout animation”. With this technique, each moving part of an image is separated into its own image, similarly to with Character Models.

Unlike Character Models, which only separated the facial features and are mostly static, monster models’ images are separated in a much more granular manner: the limbs and body are typically separated into several pieces each. Each of these pieces can then be rotated and moved independently of each other to create animations, similar to controlling a puppet.

When animating images this way, it’s necessary for each image to extend beyond what is visible at first, as the rotation, movement and scaling can reveal these hidden parts and otherwise cause a gap.

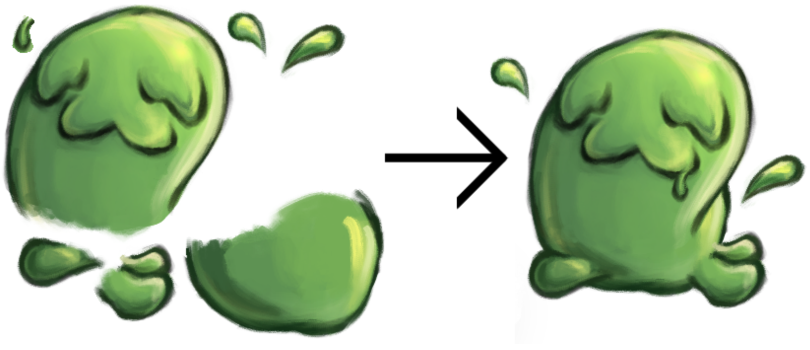


Figure 5.3 Slime sprite parts on the left, assembled on the right

These parts are then exported into standalone images, each loaded into the engine as its own Sprite2D Node, and assembled back together. To help with the assembly, a reference image with the fully assembled model can be placed in the back to be used as a guide.

Rather than only using one AnimationPlayer that animates the entire sprite, sometimes multiple AnimationPlayers are used to animate each piece independently. This can create more natural movement and make the looping less obvious than if it was all in one animation that is always aligned the exact same way. For example, the Slime enemy contains a main AnimationPlayer for its idle animation, as well as two additional AnimationPlayers for the “Splash” Nodes, that animate the small drip effects to the left and right of its body.

One other notable detail about Monster Model is that all of its AnimationPlayer Nodes have an AnimationPlayerRandomStart script attached to them.

Code listing 5.2 AnimationPlayerRandomStart

```
public partial class AnimationPlayerRandomStart : AnimationPlayer
{
        public override void _Ready()
        {
                CallDeferred(MethodName.RandomizeCurrentAnimationPosition);
        }

        public void RandomizeCurrentAnimationPosition()
        {
                Seek(CurrentAnimationLength * GD.Randf(), true);
        }
}
```

The AnimationPlayerRandomStart script randomizes the starting position of the idle animations. Without this script, if there were multiple instances of the same enemy on the screen at the same time, they would play the exact same animation in synchronization. By randomizing the start times, more organic visuals are achieved as the idle movement will be offset randomly in each instance.

5.1.2.3 Atlas Texture

Aside from simply using image textures, the project also makes use of a feature called “Atlas Texture”. An Atlas Texture is an image that contains multiple textures collected into a single image file.

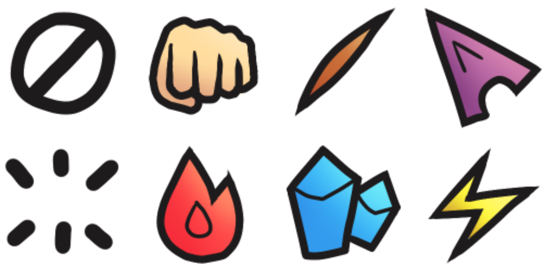


Figure 5.4 Atlas Texture SkillElement icons, comprised of 8 128x128 icons combined into a single 512x256 image

Each individual image can then be obtained from the Atlas Texture by specifying its coordinates and size in the Atlas Texture.

The source Atlas Texture is normally imported similar to any other image texture. Then, a new image resource is by right-clicking inside the FileSystem window, selecting Create New > Resource, and finally “AtlasTexture”.

This will create an new .tres file in the project file system that contains an image texture pulled from a source Atlas Texture. The Atlas texture is defined in the Inspector window upon selecting the .tres file, and assigning an image to it. Then the final texture is obtained by specifying a specific area of the Atlas Texture to pull from, using the “Edit Region” button.

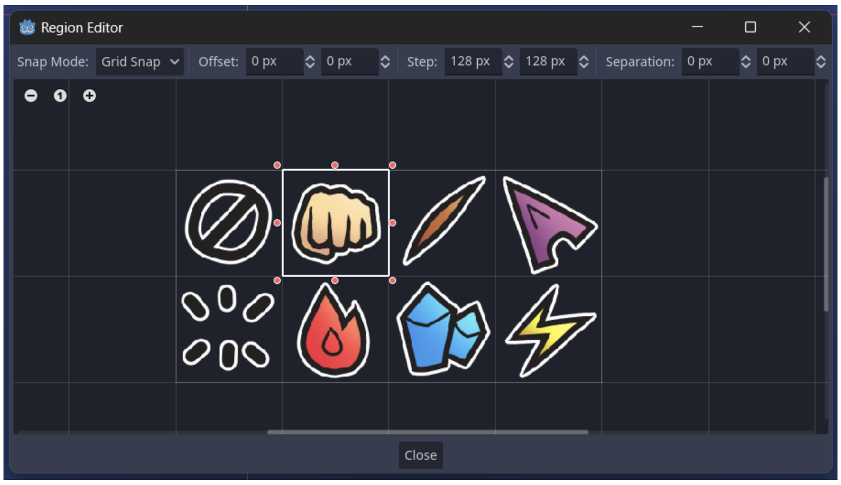


Figure 5.5 Selecting a texture from an AtlasTexture using the Region Editor

This feature is used for skill icons in the game, as they are small 128x128 textures, with one unique icon required for each skill.

5.1.2.4 UI Elements

All Control Nodes in the example project are configured with proper anchors, allowing for the window resolution to be changed without breaking any UI layouts.

Several of the UI elements in the project are more complex Scenes stored in a separate .tscn, with scripts that receive the necessary data, that they then use it to update the UI elements accordingly.

Many of these UI element scripts follow the same structure of using the _Ready() method to obtain references to its child Nodes to store them in variables, and later accessing these variables to update the UI information displayed.

Code listing 5.3 CharacterBar excerpt, which displays information about a character

```
public partial class CharacterBar : Control
{
    ...
    public override void _Ready()
    {
        name = GetNode<RichTextLabel>("Name");
        icon = GetNode<TextureRect>("Icon");
        hp = GetNode<Label>("HPCurrent");
        ...
    }

    public void Update(BattleCharacter character)
    {
        name.Text = Utility.CharacterBBName(character.Who);
        icon.Texture = GD.Load<Texture2D>($"res://Images/CharacterIcon/
                                    {character.Who}.png");
        hp.Text = character.Health.ToString();
        ...
    }
    ...
```

5.1.3 Shaders

In game development, a shader is a program used to alter or define how visuals are drawn. Typically processed by the GPU, they are written using a shading language.

5.1.3.1 Shaders: Brief Overview

This section briefly goes over a few shader concepts, mainly the ones appearing in the shader code used in the example project. For a more detailed explanation, see the “Shaders” page in the Godot Documentation. [19]

Godot uses its own shading language that is syntactically similar to the OpenGL shading language, more specifically GLSL ES 3, which in turn uses syntax similar to C. [20] Godot shaders use the .gdshader file extension.

Shader code begins by defining the shader_type. As the example project is 2D, all the visuals are 2D images such as Sprite2D and Control Nodes. More specifically they are CanvasItems.

Some of the data types used in the shaders in the project are:

int: an integer value

float: a floating-point value.

Floating-point numbers have to be declared explicitly by placing a decimal point after the number, otherwise it is considered an int value.

vec2/vec3/vec4: a vector with 2, 3 and 4 floating-point values respectively.

■ **Code listing 5.4** Shader: defining and assigning various vector values

```
vec2 coord = vec2(1.0, 6.0);
vec3 coord3d = vec3(1.0, 0.0, 2.0);
vec4 colorWhite = vec4(1.0); // vec4(1.0, 1.0, 1.0, 1.0)
```

They can be used to represent various data. For example: vec2 can be used for 2-dimensional x and y coordinates, vec3 for 3-dimensional coordinates or RGB color values, and vec4 for RGBA color values with transparency.

When using vectors to represent colors, each value represents an individual color channel value, ranging from 0.0 to 1.0 (corresponding with 0 to 255 in more conventional RGB representations).

The corresponding value of a vector can be accessed using x, y, z, w for the 1st, 2nd, 3rd, and 4th value respectively. Or alternatively r, g, b, a. These can also be chained using “swizzling” to create new vectors.

■ **Code listing 5.5** Shader: Working with vec values

```
vec4 baseVec = vec4(0.0, 1.0, 2.0, 3.0);
baseVec.r;  // 0.0
baseVec.x;  // 0.0
baseVec.y;  // 1.0
// Swizzling
vec3 newVec = baseVec.wxy;      // vec3(3.0, 0.0, 0.1)
```

sampler2D: type used to bind 2d textures to, such as images. Godot can also generate textures such as color gradients, noise textures, or 2D curves, which can be defined from the Inspector window.

To pass arguments to shaders, the “uniform” keyword can be used before declaration of variables. Uniform values are global to the entire shader and can also be changed from outside, such as from a C# script, or through the Inspector window similar to any other Node property. This also means they can be animated like a property using AnimationPlayer Nodes.

Using the optional “hint” keywords, one can define how the context in which the value is used. which allows the editor to display the proper UI when attempting to assign the value. For example, using the “source_color” hint for vec4 values will make the editor display a color picker when setting the value from the Inspector window.

■ **Code listing 5.6** Defining a shader uniform value

```
shader_type canvas_item;

uniform vec2 offset;
uniform float shadowIntensity: hint_range(0.0, 1.0, 0.1);
uniform vec4 multiplyColor: source_color;
```

■ **Code listing 5.7** Setting a shader parameter from a C# script

```
((ShaderMaterial)someNode.Material)
    .SetShaderParameter("offset", new Vector2(100, 200));
```

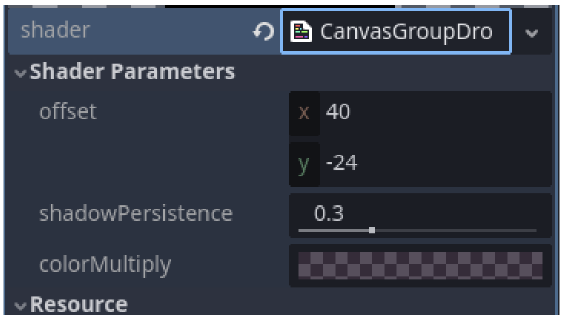

shader CanvasGroupDro
Shader Parameters
offset x 40
y -24
shadowPersistence 0.3
colorMultiply
Resource

Figure 5.6 Shader from code listing 5.6, with its uniform properties in the Inspector window

Using the fragment() function, which runs on every pixel of the object rendered on the screen, one can change the way the object is rendered. To obtain or change the color value of the pixel currently being processed, the "COLOR" variable is used.

Code listing 5.8 Shader code example

```
shader_type canvas_item;

void fragment() {
        vec4 c = COLOR;
        if (c.r > 0.5) c.r = 1.0;
        else c.r = 0.0;
        COLOR.rgb = c.rgb;
}
```

The above code reads the pixel color value into a new variable, sets the red channel to 1.0 or 0.0 depending on which half of the range the original value corresponded to. This new color value is then assigned back to the COLOR, which outputs the resulting color to be rendered.

A shader can also access the texture of the current object using the "TEXTURE" and "UV" variables, where TEXTURE is the texture of the object (in case of Sprite2D Nodes it is the sprite image), and UV are the x, y coordinates of the currently processed pixel. For UVs, note that the (0.0, 0.0) coordinate is the top left corner of the texture and (1.0, 1.0) is the bottom right corner of the texture, they are not pixel coordinates.

Using the texture() function that receives a texture and the UV coordinate to read, the pixel color value at the corresponding UV can be obtained.

Code listing 5.9 Shader code reading the color value from the object's texture

```
void fragment() {
    vec4 color = texture(TEXTURE, UV);
}
```

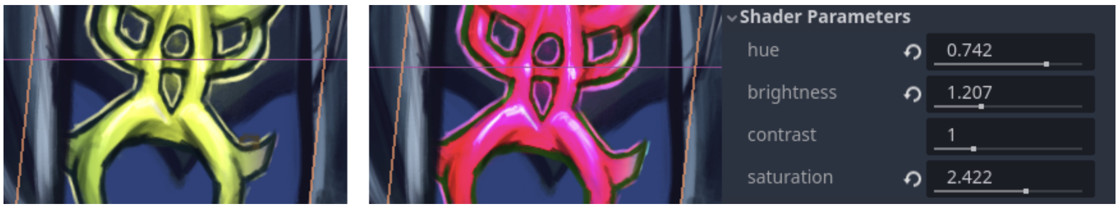
There are also many other variables and functions, such as the "TIME" variable, which contains elapsed total time in seconds and can be used to animate visuals, and mathematic functions such as abs(), clamp(), mod(), clamp().

Finally assigning a shader to a Node is done so in the Inspector window through the "material" property by creating a new ShaderMaterial and assigning the .gdshader file to it.

5.1.3.2 Shaders used in the project

The example project uses three shaders:

ColorShader: a shader that filters the image in 4 aspects: Hue, Brightness, Saturation and Contrast.


Shader Parameters

hue 0.742

brightness 1.207

contrast 1

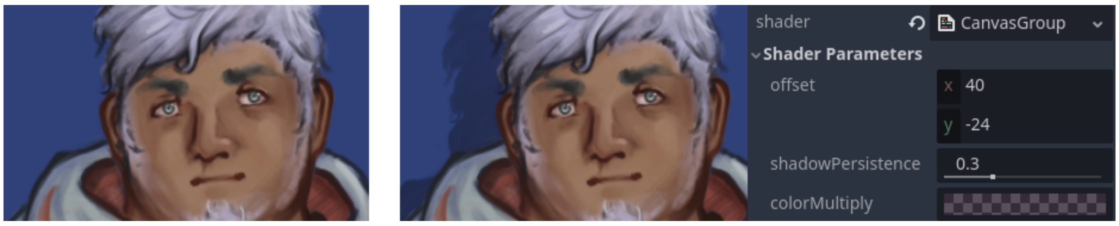
saturation 2.422

Figure 5.7 ColorShader Shader, on the left the original visual, on the right with the shader applied

The hue shift is achieved by converting the RGB value of the pixel to HSV (Hue/Saturation/Value), shifting the hue, and then converting the HSV value back to RGB to display. The implementation of this algorithm was taken from the following source: [21].

The brightness, saturation and contrast filters are then achieved using the mix() function. The implementation of this algorithm was taken from the following source: [22].

DropShadow: creates a drop shadow behind the object with parametrizable color, offset and gradient.


shader CanvasGroup

Shader Parameters

offset x 40 y -24

shadowPersistence 0.3

colorMultiply

Figure 5.8 DropShadow Shader, on the left the original visual, on the right with the shader applied

This shader works by sampling the base sprite texture to be drawn normally, then it samples the same texture with an offset and multiplies it by the colorMultiply color to obtain the drop shadow. Finally, the drop shadow and sprite are combined together using an alpha compositing formula.

Code listing 5.10 DropShadow shader fragment function

```
void fragment() {
    // Get source texture and dropshadow.
    vec4 sprite = texture(TEXTURE, UV);
    vec4 drop = texture(TEXTURE, UV + offset * SCREEN_PIXEL_SIZE)
                * colorMultiply;
    // Make drop shadow lighter towards the top.
    drop.a *= clamp(UV.y + shadowPersistence, 0.0, 1.0);

    // Standard formula alpha compositing
    COLOR.a = sprite.a + drop.a * (1.0 - sprite.a);
    COLOR.rgb = (sprite.rgb * sprite.a +
                drop.rgb * drop.a * (1.0 - sprite.a)) / COLOR.a;
}
```

PulsingColor: overlays a texture over the object that moves to the right indefinitely, creating a “pulsing effect”.

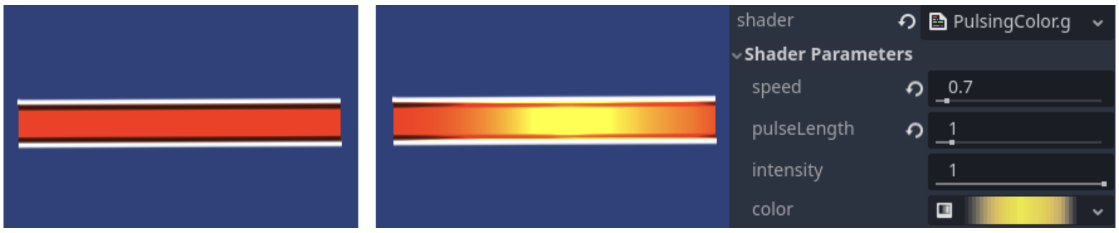

shader PulsingColor.g
Shader Parameters
speed 0.7
pulseLength 1
intensity 1
color

Figure 5.9 PulsingColor Shader, on the left the original visual, on the right with the shader applied

This is achieved by sampling the top texture with an offset using the TIME variable, which creates the pulsing effect. This top texture is then combined with the base texture using the Color Dodge layer blend algorithm.

The shader is also parametrizable: the length of a single repetition of the top texture, i.e. the frequency, the intensity of the overlay, as well as the speed of the pulse.

Code listing 5.11 PulsingColor shader fragment function

```
void fragment() {
    // Get the top layer from the supplied texture,
    vec4 sourceColor = texture(color, vec2((2.0 / pulseLength) * UV.x)
                                  - speed * TIME);

    // Add the top layer the base image using Color Dodge blend mode.
    vec3 c = sourceColor.rgb * sourceColor.a * intensity;
    COLOR.rgb = COLOR.rgb/(vec3(1.0) - c);
}
```

5.1.3.3 Shaders on CanvasGroup Nodes

Some of the above shaders also contain CanvasGroup variants, which require slightly different acquisition of the texture because of the way CanvasGroup implements its rendering. Rather than simply accessing it using “texture(TEXTURE, UV)”, the following code has to be used:

Code listing 5.12 Shader: getting the texture from a CanvasGroup Node

```
uniform sampler2D screen_texture :  hint_screen_texture, repeat_disable,
                                    filter_nearest;
...
void fragment() {
    vec4 c = textureLod(screen_texture, SCREEN_UV, 0.0);
    if (c.a > 0.0001)  c.rgb /= c.a;
        vec4 obtainedTexture = COLOR * c;
```

5.2 Dungeon Traversal

Dungeon traversal is implemented by the DungeonEngine Main Scene Node. When transitioning over to the DungeonEngine Scene, the SceneManager instantiates the DungeonEngine Node, and passes a DungeonSetup to it, which contains the deck setup and various data pertaining to the dungeon.

5.3 DungeonSetup

The DungeonSetup is a class inheriting from Resource. It contains the following data, which is defined through the editor:

- DungeonName
- Music
- StartingHandsize
- BossCard
- DeckDefine

The DeckDefine variable contains all the cards in the deck. It is an array of DungeonCard-Bundle instances, which is a pair of values in the form of DungeonCard, and the amount of it contained in the deck.

The DungeonEngine uses the DungeonSetup data to instantiate a deck containing individual instances of the cards. The deck is represented by the DungeonDeck class which manages the functions of the deck.

5.3.1 DungeonDeck

The dungeon deck obtains all the cards yet to be drawn by the player. This includes the “boss card”, the card that is always drawn last. The regular cards are stored in an array, containing an instance of each card, whereas the boss card is stored in a separate variable.

The DungeonDeck class then contains the DrawFromDeck() method, which attempts to draw a card from the regular card array, and if it is empty, it draws the boss card instead.

DungeonDeck also contains the ShuffleDeck() method and CardsLeft() method that are used by the DungeonEngine.

5.3.2 DungeonCard

DungeonCard is the base class representing an instance of a card. As it is a resource, it’s parameters are defined in the editor, these parameters are things such as the card name, description, and the card illustration image.

It also contains the abstract UseCard() method, which executes the effects of the card, and is implemented by classes inheriting from DungeonCard that represent a specific function. DungeonCard also contains definitions for custom Signals, which are connected to the DungeonEngine, and are how cards communicate with the DungeonEngine to request it to perform an operation, such as starting a battle.

The MonsterCard class represents a battle, and contains a BattleSetup, which is a Resource class containing an array of enemy instances. It also contains the IsBossBattle boolean, which is used to mark and differentiate between regular battles and boss battles.

5.4 Battle System

The main piece of data, around which battles revolve are battle actors, the participants of a battle: player characters and monsters. They are represented by classes that contain the various stats and skills of a battle actor. The base class facilitating this is called “BattleActorStats”

Skills are each represented by a class that contains information such as cost, cooldown, as well as the implementation of the logic and execution of the skill. The base class facilitating this is called “BattleSkillData”

These classes ar used only to store information about each battle actor and skill. When a battle begins, all this data is passed to a main class which handles all of the battle logic, the “BattleEngine” Node. The BattleEngine will then use this data to instance all battle actors using its own internal classes that contain additional variables and functions that it uses to facilitate the flow of battle and interaction between battle actors.

5.4.1 BattleActorStats and BattleActor

BattleActorStats is a base, abstract class, from which both the player character data (Character-Stats) and monster data (MonsterStats) inherit. It contains stats such as attack, defense, speed, as well as elemental affinities and skills.

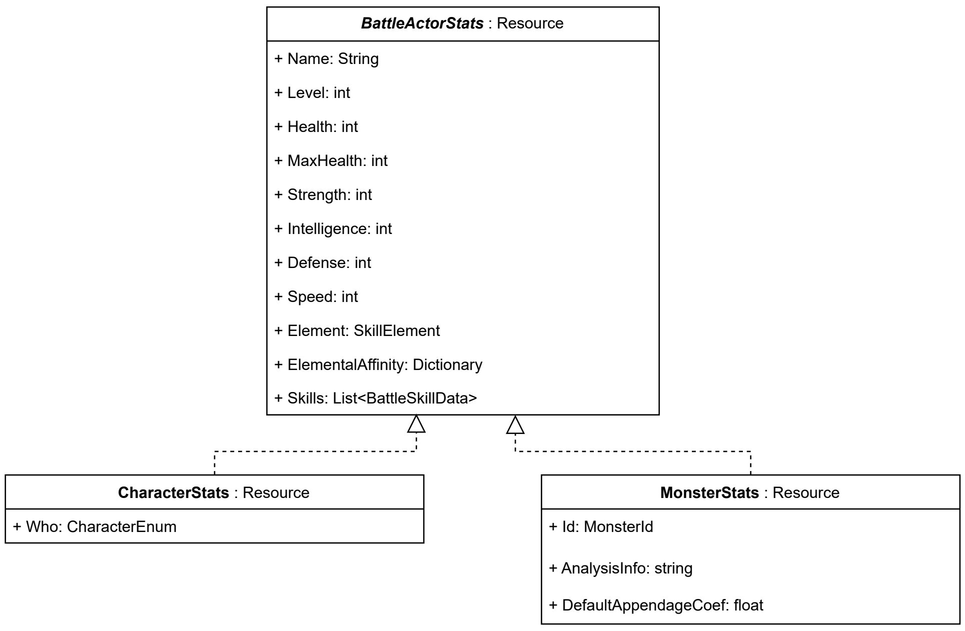


Figure 5.10 BattleActorStats Diagram

BattleActorStats is a custom resource, which means it contains stats of various characters and monsters, that are defined through the editor, and stored into .tres files. These .tres files are used to define the starting stats of player characters, and to define monster stats.

When a battle begins, BattleActorStats are used to instantiate BattleActor, which is the class used by the BattleEngine to facilitate battle interactions. BattleActor contains additional variables only relevant during battles such as Stack count, and whether the actor’s turn has been expended. Similar to BattleActorStats, BattleActor is an abstract class, that is inherited by BattleActor and BattleMonster.

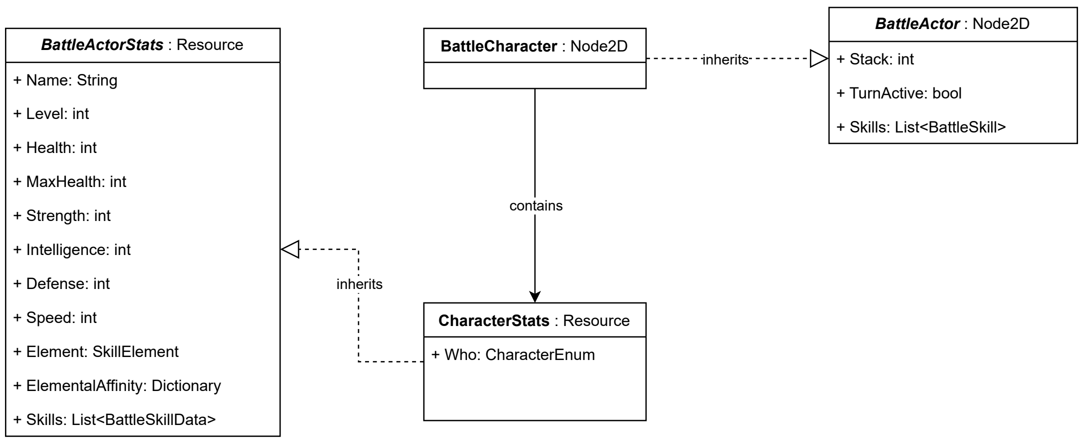


Figure 5.11 BattleCharacter diagram. For BattleMonster, same composition.

5.4.2 BattleSkillData and BattleSkill

BattleSkillData is a base, abstract class that all skills inherit from. It contains variables such as skill name, type, element, targetting style, cost, cooldown and whether the skill is a Snap Skill. It also contains the definition of methods called by the BattleEngine, such as Execute which performs the skill, and EstimateDamage which calculates and returns the expected damage output to be displayed to the player.

Similar to BattleActorStats, BattleSkillData is also a custom resource, and is used to only contain information, which is then instantiated into BattleSkill, which contains additional data used during battles. This is automatically handled by BattleActor when assigning an instance of BattleActorStats to it: it reads the BattleSkillData and instantiates them as BattleSkill.

BattleSkill contains a cooldown counter variable, and several methods that allows the BattleEngine to interact with it, such as IsUsable(), which returns a boolean about whether the conditions required to use the skill are fulfilled, or Use() and Miss() which are called when the skill is used successfully or unsuccessfully.

5.4.3 BattleEngine

BattleEngine oversees the overall logic of a battle, including interaction between battle actors, and parsin player input. The BattleEngine implementation consists of 5 files, separated into groups that each handle a specific aspect of the battle. By using the partial keyword, the definition of the class is broken up into separate files, which are combined back together during compilation.

BattleEngine.cs handles general functions of the BattleEngine.

BattleEngineActionControl.cs handles viewing and selection of skills, as well as displaying the appropriate UI based on type of skill selected.

BattleEngineActionExecution.cs handles execution of skills, i.e. executing the skill's effects, and cleaning up after resolving it.

BattleEngineUiInteraction.cs handles passing data to BattleUI, which handles displaying data.

BattleEngineStateLogic.cs handles ControlState and its transitions.

5.4.3.1 ControlState

At any point in time, the BattleEngine is in one of the several ControlStates. The current ControlState dictates which player inputs are read, and how they are interpreted. ControlStates are constantly transitioned between, as a result of the player's inputs. These control states are:

ENEMY_TURN: Enemy turn currently taking place.

In this state, the player is currently waiting for all enemies to finish their turns. The player can swap the currently selected party member, but cannot activate skills.

PLAYER_DEFAULT: The default state when it's the player's turn.

In this state, the player can select the current party member to act with, view their available skills, skill details, and select a skill to activate.

PLAYER_TARGETTING_ENEMY: The control scheme used by skills that have the ENEMY_TARGET TargettingType.

This is the game's main method of performing offensive skills, where the mouse cursor changes into a less accurate target mark, and awaits further input from the player by aiming for and clicking at a monster's appendage.

The player can cancel out of this state by changing the active character, or by pressing ui_cancel. This will return the state back to PLAYER_DEFAULT.

PLAYER_CUSTOMWINDOW: The control scheme used by skills that have the CUSTOMWINDOW TargettingType.

In this state, a UI element in the form of a "custom window" supplied by the skill is opened. The control and function of the window is implemented by the custom window itself, which allows it to flexibly define how it wants to be used.

The player can cancel out of this state by changing the active character, or by pressing ui_cancel. This will return the state back to PLAYER_DEFAULT.

PLAYER_SELECTING_ENEMY_CUSTOMWINDOW: The control scheme used by skills that have the ENEMY_SELECT_CUSTOMWINDOW TargettingType.

In this state, the game awaits the player to select an enemy by simply clicking it with the cursor. This is different from PLAYER_TARGETTING_ENEMY, where the cursor changes to a target mark.

After the selection is made, a custom window is opened and passed information about the selection. Continue into the PLAYER_CUSTOMWINDOW state.

The player can cancel out of this state by changing the active character, or by pressing ui_cancel. This will return the state back to PLAYER_DEFAULT.

END_SCREEN: The state entered after a battle is finished, i.e. all enemies are defeated.

The only accepted input in this state is clicking to exit the BattleEngine and return to the DungeonEngine.

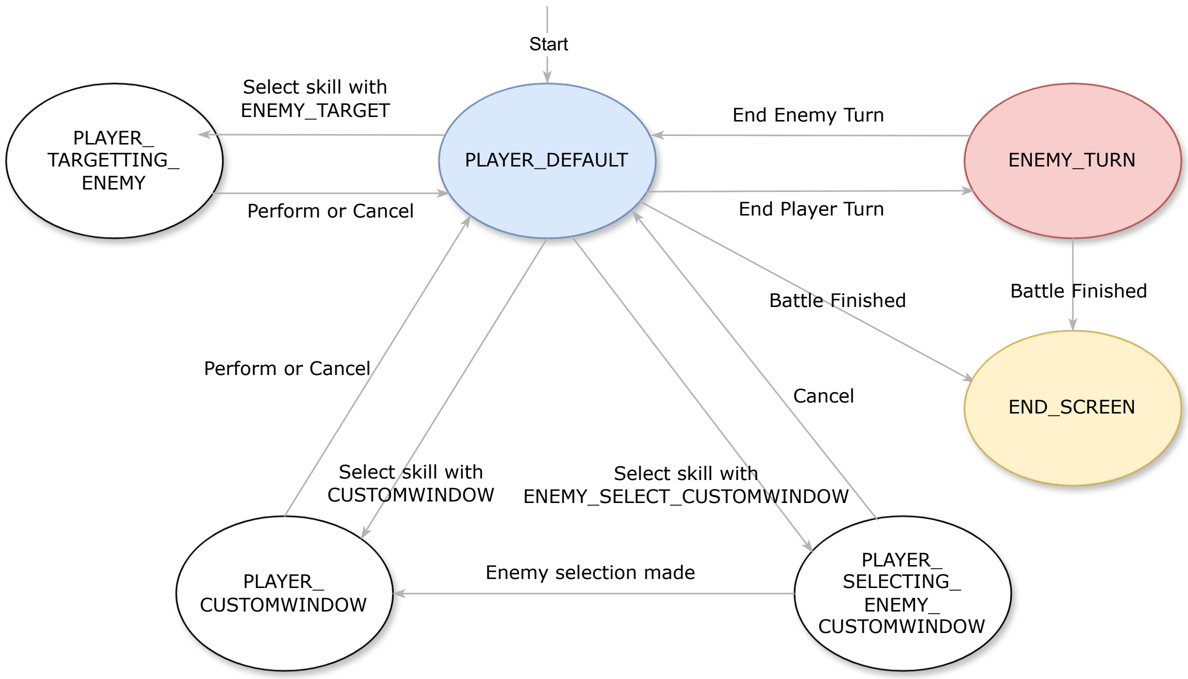


Figure 5.12 ControlState diagram

5.4.3.2 Interactions and passing around data

Anything that alters the state of any BattleActor on the field could be considered an "Interaction": attacking a monster, being attacked, gaining Stacks, swapping with a party member, analyzing a monster, etc.

The primary way of triggering these actions is through skills, which have to be able to do all of the above. To do this they need access to the relevant data, which is managed by the BattleEngine. Because of the flexible nature of what various skills are capable of, the data that is needed can range from things like a single user, a user and a target, or a user and multiple targets.

In order to be able to pass all of this information around, there are two structures that are be passed to skills that provide all necessary data: BattleFieldData and BattleInteractionData

BattleFieldData contains references to the player Party, player Bench, and the enemy Party. When a skill is performed, BattleFieldData will be passed to it by argument, allowing it to access any BattleActor in the field as desired.

BattleInteractionData contains the actors involved in the interaction: a user only, a user and their target, or a user and multiple targets. Contains multiple constructors depending on how many BattleActors are involved.

Aside from BattleActors, BattleInteractionData can also carry float or int values by calling the AddFloatValue() and AddIntValue() methods respectively. This can be used by skills that require parametrization of a skill.

While it may seem like BattleFieldData is redundant as the involved BattleActors are already included in BattleInteractionData, there may be cases where a skill can target only one main BattleActor, but have side effects on any number of BattleActors on the field.

5.4.3.3 BattleMonster

Unlike the player characters, monsters require a more complex solution than just storing MonsterStats in BattleMonster, as they need to have actual sprite models on the screen that the player can interact with by clicking on them. This is handled by a base “MonsterVisuals” class, which is a Node structure that contains all of the monster’s sprites, alongside AnimationPlayer Nodes that animate them.

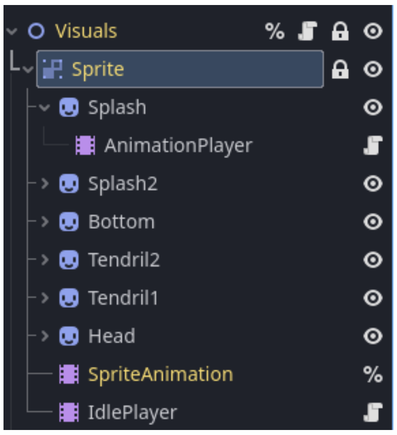


Figure 5.13 The MonsterVisual Node tree of the Slime enemy

Because of how tightly coupled the visuals and actual data of monsters are, instead of keeping BattleMonster and MonsterVisual separate, BattleMonster is represented as a Node, which contains MonsterVisual as its child, so that it can manage and interact with MonsterVisual directly by itself.

5.4.3.4 Monster Appendages and Interaction

One of the unique features of the game is the monster appendage targetting system, which has the player attempting to aim at a specific spot on a monster’s model to hit its most vulnerable spot. To implement this feature, a system akin to a hitbox is needed, i.e a way to define which area of a monster’s sprite is how vulnerable, and which areas are outside its body entirely, resulting in the player missing. There are several ways this can be achieved in Godot:

- Appendage detection using Button

  One could simply use a Button Node, which is already meant to be used for mouse input, we can then simply use the “pressed” signal as having registered the hit. However, detection would only be limited to rectangular hitboxes, as Button nodes do not support any shapes.

- Appendage detection using Area2D and CollisionPolygon2D

  These Nodes are used to implement physics and collision, making them commonly used in action and platformer games. In this turn-based project, there are not many areas where these Nodes would be necessary, but they are one of the potential solutions for this problem.

  An Area2D is a Node that can be used to detect collision between objects. In this case, it can also be used to detect a mouse click inside of it.

  To use an Area2D, it needs to have one or more child Nodes of CollisionShape2D (used for simple shapes such as rectangles, circles) or CollisionPolygon2D (polygon shape defined by user). These Nodes are used to define the shape and area of detection for the Area2D.

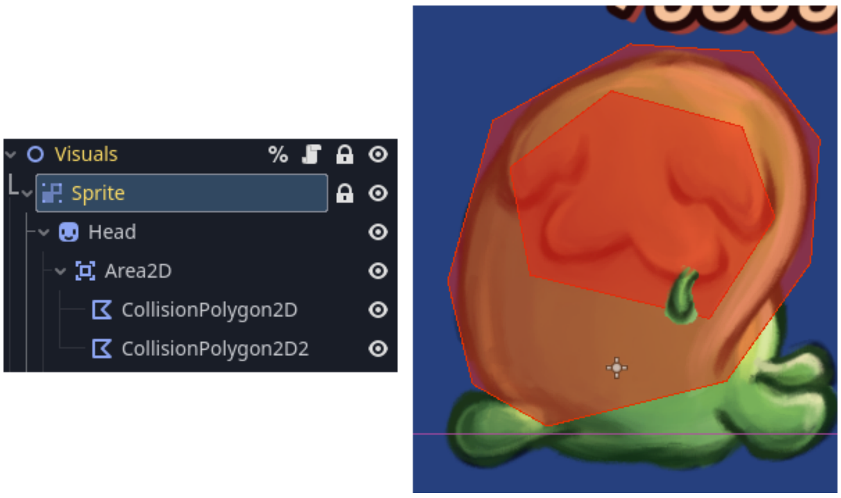


Figure 5.14 Using Area2D to define hitboxes on a monster's model

A script attached to the Area2D Node could then receive mouse input using the following code:

Code listing 5.13 Receiving mouse input in an Area2D

```
public override void _InputEvent(   Viewport viewport,
                                    InputEvent @event, int shapeIdx)
{
    // Receive input if it is a mouse click.
    if (@event is InputEventMouseButton mouseEvent && mouseEvent.Pressed)
    {
        EmitSignal(SignalName.AppendageHit);
    }
}
```

This solution works well enough. However, it feels somewhat clunky, as it is makes it necessary to create at least two Nodes for each part of a monster's sprite: an Area2D and its child CollisionPolygon2D. In addition, it also requires the polygon hitbox to be drawn by hand for each monster part.

Appendage detection using Sprite2D

This is the solution used in the project.

While not quite designed specifically for this use case, the Sprite2D Node has the IsPixelOpaque() method, which receives a Vector2D, and returns true if the pixel at the Vector2D position in the sprite is opaque, i.e. its alpha channel is higher than 0.

This means Sprite2D can be used to represent the hitbox, with its image texture being set to a hitbox image to function as a mask. The hitbox mask can be fairly easily obtained from the monster sprite textures, by converting them to monochrome images with minimal manual adjustments.

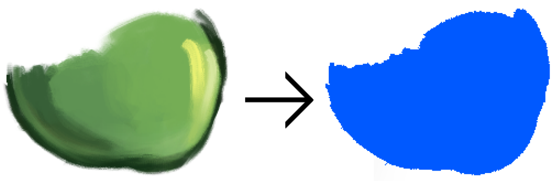


Figure 5.15 Creating a mask from a part of the Slime's sprite

The monster model is composed of several, independently moving pieces. Each of these pieces will need an appendage mask, and one piece can have multiple masks if it contains appendage areas of different efficiencies.

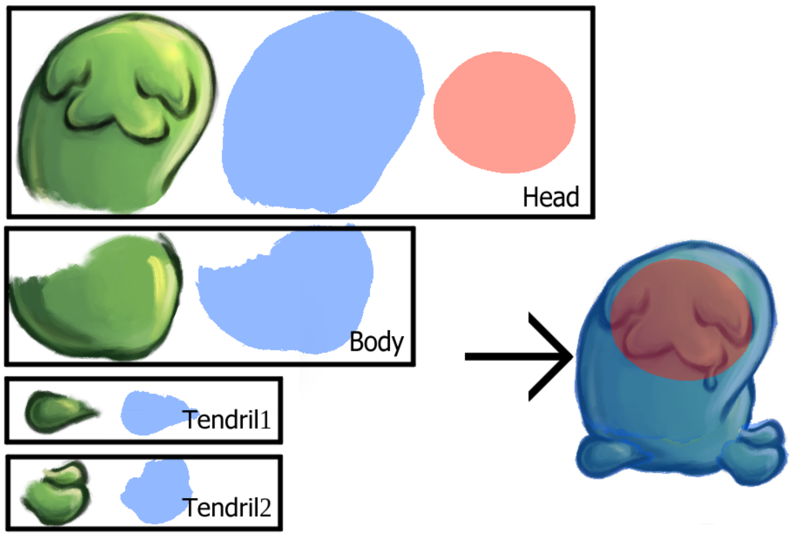


Figure 5.16 Assembling all appendage masks on the Slime model

The resulting Node tree will then contain a Sprite2D for each appendage area, attached to each part of the monster's sprite.

Finally, in a script attached to each of these Sprite2D appendage masks, MonsterAppendage, mouse input is received, checked if the mouse position is within the appendage mask, and a signal about successful hit is emitted if yes.

Code listing 5.14 Receiving input on MonsterAppendage

```
public partial class MonsterAppendage : Sprite2D
    [Signal]
    public delegate void HitEventHandler(MonsterAppendage where);
    [Export]
    public int appendageId = 0;
    public override void _Input(InputEvent @event)
    {
        if (@event.IsActionPressed("battle_clickTarget"))
            if (targettable && IsPixelOpaque(GetLocalMousePosition()))
            {
                EmitSignal(SignalName.Hit, this);
            }
    }
```

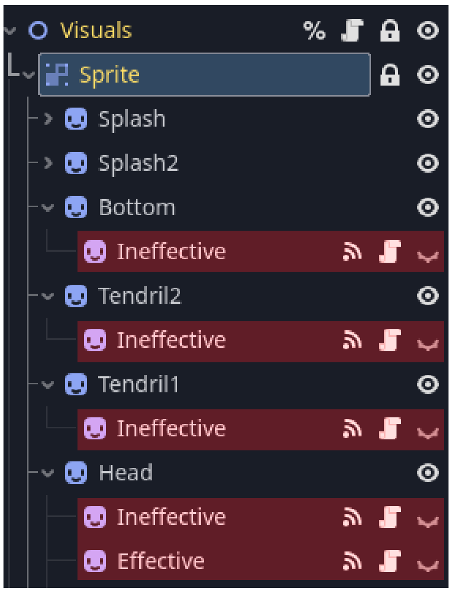


Figure 5.17 Node tree of the Slime model with MonsterAppendage Nodes highlighted in red

And so, the appendage hit is registered by the MonsterAppendage at the bottom of the Node tree, where it gradually bubbles upward through each parent until it reaches the BattleEngine Node at the top.

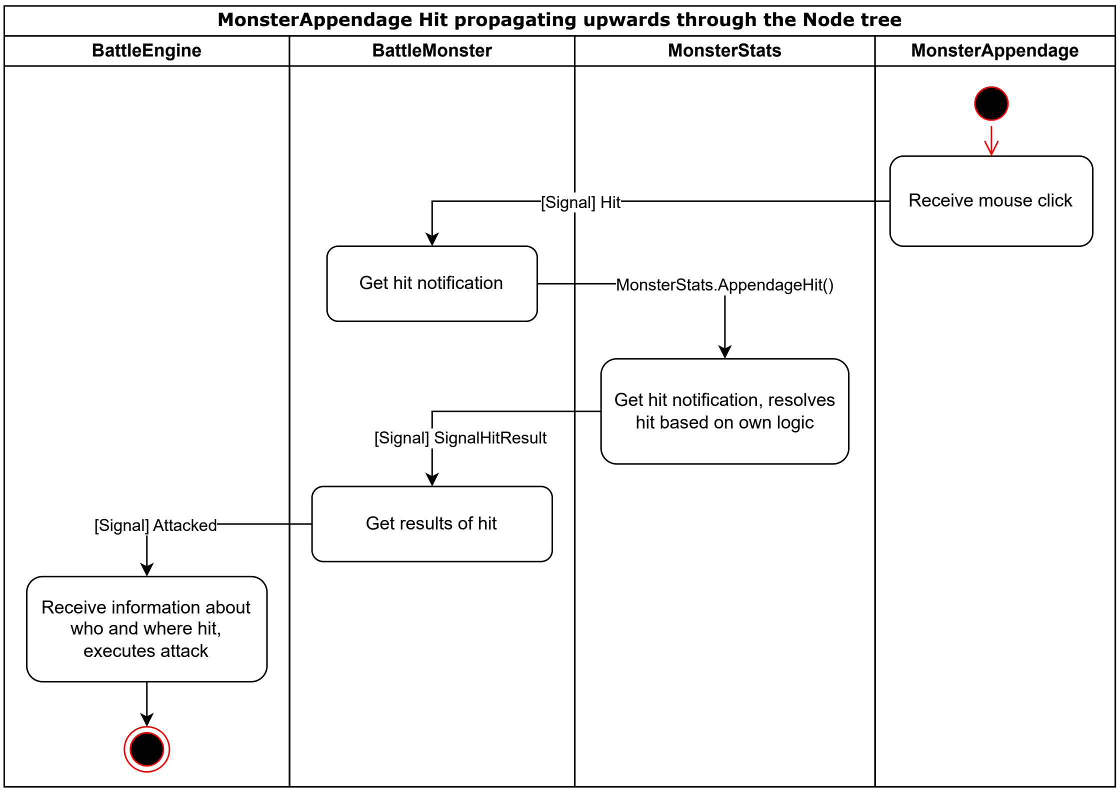

MonsterAppendage Hit propagating upwards through the Node tree
BattleEngine
BattleMonster
MonsterStats
MonsterAppendage
Receive mouse click
[Signal] Hit
Get hit notification
MonsterStats.AppendageHit()
Get hit notification, resolves hit based on own logic
[Signal] SignalHitResult
Get results of hit
[Signal] Attacked
Receive information about who and where hit, executes attack

Figure 5.18 MonsterAppendage hit propagating upwards towards BattleEngine

First, the MonsterAppendage is hit and emits a Hit signal. The Hit signal is connected to the BattleMonster Node, the root of the monster model, which then passes the data to the MonsterStats it contains.

MonsterStats can then parse the information based on which appendage was hit. For this, the MonsterAppendage's editable "appendageId" variable is used to differentiate and identify the various appendages.

For example: in case of the Slime enemy, all ineffective (blue) appendages were set to appendageId 0, and the effective (red) appendage is set to 1. The SlimeStats class then receives the appendageId and returns a damage coeficient of 0.5 if the appendageId is 0, and a damage of coefficient 1.0 if the appendageId is 1.

Code listing 5.15 SlimeStats receiving Appendage Hit information and returning damage coefficient

```
public override void AppendageHit(MonsterAppendage appendage)
{
    if (appendage.appendageId == 1)
    {
        EmitSignal(SignalName.SignalHitResult, 1.0f);
    }
    else
    {
        EmitSignal(SignalName.SignalHitResult, 0.5f);
    }
}
```

The MonsterStats then returns this damage coefficient to BattleMonster using a signal, which once again passes the information upwards to the BattleEngine. where the hit is finally registered and parsed.

5.5 Localization

Localization, or multiple language support is a very common feature in games nowadays. In Godot, localization is done through "comma separated value" files, or "CSV', which is a text format that stores table-like data using commas to separate each cell.

In Godot, CSV translation files consist of a key, and a column for each supported language. Here is an example of a CSV that contains text for UI elements on the title screen:

Code listing 5.16 CSV for translation

```
"keys","en","cs"
"T_UI_TS_NEWGAME","New Game","Nová Hra"
"T_UI_TS_LOADGAME","Load Game","Pokracovat"
"T_UI_TS_SETTINGS","Settings","Nastavení"
"T_UI_TS_EXIT","Exit","Exit"
```

Or, displayed in the form of a spreadsheet for more legibility:

Table 5.1 CSV for translation, spreadsheet visualization.

keys	en	cs
T_UI_TS_NEWGAME	New Game	Nová Hra
T_UI_TS_LOADGAME	Load Game	Pokračovat
T_UI_TS_SETTINGS	Settings	Nastavení
T_UI_TS_EXIT	Exit	Exit

The "en" and "cs" fields are locale codes, which are based on the Unix standard locale strings, a list of which is available in the documentation. [23]

Adding this CSV file to the project directory will prompt Godot to generate .translation files for each language. These translation files need to be added to a list the engine pulls from by going to Project > Project Settings > Localization > Add...

All text in Control nodes is automatically translated by default.[24] Thus, in the editor, the Control node will contain the key text, and during runtime, the engine will automatically substitute the appropriate translation.

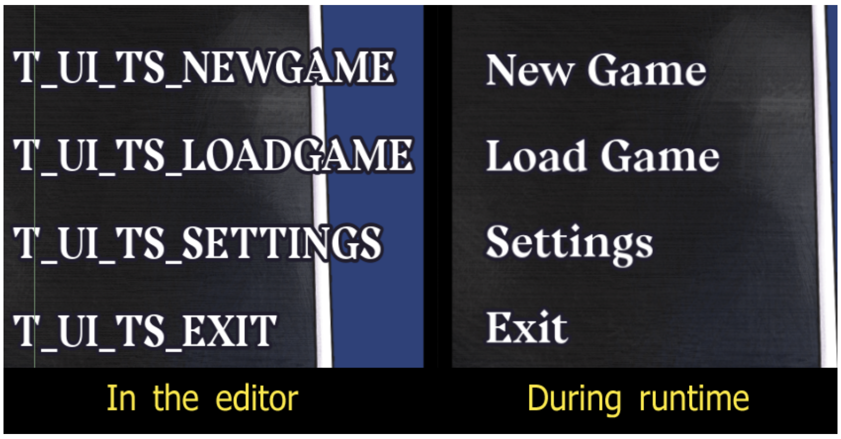


Figure 5.19 Localization example in-game

The default language upon startup is based on the user's system locale. If the locale doesn't have any translations for a given key, a fallback locale will be used. By default, this locale in the English en, but can be changed in Project > Project Settings..., and under the field "locale/fallback".

To change the locale manually from code, the TranslationServer's SetLocale() method can be used:

Code listing 5.17 Setting the locale from code

```
TranslationServer.SetLocale(someLocaleString);
```

In case of dynamic text, i.e. text with a string variable inserted into it, the string.Format() method can be used.

Code listing 5.18 String Formatter

```
string.Format("{0} uses {1} on {2}!", "ATTACKER", "SK", "TARGET"));
```

The above code results in the text "ATTACKER uses SK on TARGET!" being printed. However, it does not allow for localization as it features a hard-coded string.

To integrate localization into the above example, the string that is passed to the Format method has to be the already retrieved translation of the string key. For that, Godot's Tr function can be used, which receives a key string, and returns the appropriate localized string.

With this localization csv:

Table 5.2 CSV using dynamic text

keys	en	cs
T_BM_MONSTERATTACK_SINGLE	{0} uses {1} on {2}!	{0} použil {1} na {2}!

The following code will result in the same string output as the 5.18 code.

Code listing 5.19 String Formatter together with localization

```
string.Format(Tr("T_BM_MONSTERATTACK_SINGLE"), "ATTACKER", "SK", "TARGET"));
```

5.6 Save files

To implement save files for the game means to be able to convert all necessary game data into a format that can be stored and persisted on the player's computer. This process is called "serialization". The opposite process, converting this format back into data that can be used by the program, is then called "deserialization".

There are a few methods that can be used to accomplish this, ranging from Godot-specific solutions, to more general ones. Here are some of the options to consider:

- ResourceSaver and ResourceLoader
- BinaryFormatter
- JSON

5.6.1 ResourceSaver and ResourceLoader Serialization

ResourceSaver and ResourceLoader are a Godot features that automatically handle serialization and deserialization of classes inheriting from Godot's Resource type. This is one of the easiest methods to store saves in Godot, as all serialization could be handled by calling following line of code:

Code listing 5.20 Serializing data using ResourceSaver

```
ResourceSaver.Save(someResource, "user://file_name.tres");
```

To load the data back into the game, the ResourceLoader class would be used:

Code listing 5.21 Serializing data using ResourceSaver

```
Resource someResource = ResourceLoader.Load("user://file_name.tres");
```

However, this method is prone to code injection and therefore poses a security risk. [25] As such, it is not an ideal solution.

5.6.2 BinaryFormatter Serialization

Another option is to save the data into a binary file using C#'s BinaryFormatter library.

However, BinaryFormatter suffers from a similar issue to Godot's ResourceSaver, with code injection as a potential security risk.[26] This is explicitly stated in the documentation page for BinaryFormatter, instead recommending people to use other formats such as XML or JSON. As such, BinaryFormatter isn't the ideal solution either.

5.6.3 JSON Serialization

JavaScript Object Notation, as the name suggests, is a format derived from JavaScript. However, the format itself is language-independent, as it is simply plain text. A JSON file is built on two main structures: a collection of key and value pairs, and ordered lists such as arrays.

Out of the three discussed options, JSON is the least prone to the aforementioned security issues, and is thus the one used in the example project.

Godot has its own functions for manipulating JSON files. However, there are a few caveats:

- There is no automatic serialization, and thus it would be necessary to manually handle the conversion from class instance data to something that can be more easily converted to JSON.
- Godot's JSON API was designed for use with the dynamically typed GDScript, making it somewhat difficult to work with in the statically typed C#, as it would also require working with Godot's Variant type, which facilitates the dynamic typing of GDScript.

  This is discouraged by the Godot Documentation, which recommends taking advantage of C#'s type safety.[27]

Therefore, the ideal direction to take would be to use a C# specific solution. While C# has a native JSON library, it is not powerful enough for use with the current class structures, and so the Json.NET library will be used.

5.6.3.1 Adding external libraries

Adding an external library to a Godot C# project should be the same as with any other C# project, without any additional steps. To add a library to the project, the "dotnet add package" command can be used. In VSCode, this can be done through the terminal using View > Terminal, and entering the following command:

Code listing 5.22 Adding the Json.NET library

```
dotnet add package Newtonsoft.Json --version 13.0.3
```

5.6.3.2 Serializing simple objects

To serialize an object, the JsonConvert.SerializeObject() method can be used. However, this will serialize every field that the object contains. In case of classes such as BattleActorStats, which inherits from Resource, this means that even internal fields that are part of Godot's Resource class will be serialized.

To address this, the Opt In attribute can be used, which will only serialize fields that are explicitly mark as serializable.

Code listing 5.23 Annotating a class for serialization

```
[JsonObject(MemberSerialization.OptIn)]
public partial class MyClass: Resource
{
    [JsonProperty]
    private string serializedVariable;

    [JsonProperty]
    public int SerializedField { get; set; }

    public int thisWillNotBeSerialized;
}
```

5.6.3.3 Serializing BattleSkillData

Serialization of primitive data types such as int or string, and classes that only contain these primitive data types, doesn't require any further input. However, in case of classes such as battle skills, which are polymorphic classes inheriting from BattleSkillData, the serialization needs to be handled manually. There are two options:

- Serializing the class type using TypeNameHandling, which stores information regarding the class type.

  This would result in the JSON data looking something like this.

Code listing 5.24 Serializing polymorphic data by storing type

```
{
    "$type": "SkillPrecisionNeedle, Rpg_Project",
    "Id": 11,
    "DisplayName": "Precision Needle",
    "Type": 1,
    "Element": 3,
    "Targetting": 4,
    "IsAoE": false,
    ...
}
```

  However, this introduces a security risk, as the JSON data can be modified and used to inject malicious data.[28]

- Write a custom JsonConverter.

  This solution can take advantage of the fact that BattleSkills are stored as Godot resources, that is to say .tres files within the Godot filesystem. The serializer can then simply store the skill's path to the .tres file and later simply load the resource containing the skill from the path.

  This is achieved by writing a custom JsonConverter for the BattleSkillData class, which tells the Serializer how to serialize and deserialize BattleSkillData. First, an attribute has to be added to the BattleSkillData class that marks it as using a custom JsonConverter and which class to use as the Converter.

Code listing 5.25 Using a custom JsonConverter for a class

```
...
[JsonConverter(typeof(BattleSkillDataSerializer))]
public abstract partial class BattleSkillData : Resource
{
    private string name;
    public SkillId Id { get; protected set; }
...
```

  The BattleSkillDataSerializer class will then look like this:

Code listing 5.26 Implementing a custom JsonConverter

```
public class BattleSkillDataSerializer : JsonConverter
{
    public override bool CanConvert(Type objectType)
    {
        return objectType == typeof(BattleSkillData);
    }

    public override BattleSkillData ReadJson(  JsonReader reader,
                                               Type objectType,
                                              object existingValue,
                                              JsonSerializer serializer)
    {
        return GD.Load<BattleSkillData>((string)reader.Value);
    }

    public override void WriteJson( JsonWriter writer, object value,
                                   JsonSerializer serializer)
    {
        writer.WriteValue(((BattleSkillData)value).ResourcePath);
    }
}
```

Combining all of the above and applying it to CharactersStats that stores the data of playable characters, calling JsonConvert.SerializeObjec t will result in the following JSON:

Code listing 5.27 JSON output from serializing CharacterStats

```
{
    "Who": 2,
    "Name": "Srinivas",
    "Level": 1,
    "Health": 15,
    "MaxHealth": 15,
    "Strength": 3,
    "Intelligence": 1,
    "Defense": 3,
    "Speed": 6,
    "Element": 0,
    "ElementalAffinity": {
      "BLUNT": 1.0,
      "FIRE": 1.0,
      "ICE": 1.0,
      "LIGHTNING": 1.0,
      "NONE": 1.0,
      "PIERCE": 1.0,
      "SLASH": 1.0
    },
    "Skills": [
      "res://Resources/Skills/Offensive/Physical/Blunt/FirstStrike.tres"
}
```

5.6.3.4 Serializing the main scene

Serialization for the main scene, the DungeonEngine Node, is done similarly. However, in the future, the game could have multiple different types of main scenes aside from just the dungeons (i.e. an overworld, dialogue, exploration scene), and so additional information needs to be stored so the game can determine which Node to load and instance.

To facilitate this, the IMainScene interface was created, which all Nodes that are used as a main scene should implement.

Code listing 5.28 IMainScene interface for main scene Nodes

```
public enum MainSceneEnum
{
    TITLESCREEN,
    DUNGEONENGINE
}
public interface IMainScene
{
    public MainSceneEnum MainSceneType { get; }
}
```

Code listing 5.29 IMainScene implemented in DungeonNode

```
public partial class DungeonEngine : Control, IMainScene
{
    ...
    public MainSceneEnum MainSceneType => MainSceneEnum.DUNGEONENGINE;
    ...
}
```

The save method can then retrieve this information and store it accordingly, so that it can be used later to determine the type of main scene to load.

However, this information needs to be read before attempting to deserialize the scene, as the class type of the deserialized JSON object has to be known beforehand. It should ideally be stored somewhere before the actual serialized scene then.

5.6.3.5 Save Header

It would also be useful for the save to contain metadata of some sort, such as time played, date of creation, and a description of the current location. This information could then be displayed on the save screen to make each save file more distinguishable at first glance. This information can be stored these alongside the MainSceneType value in some SaveFileHeader structure as the first chunk of data to be read.

Code listing 5.30 SaveFileHeader struct

```
public struct SaveFileHeader
{
    public string date;
    public ulong gameTime;
    public MainSceneEnum mainScene;
    public string mainSceneDescription;
}
```

The date is stored as a simple string, as there is no necessity to perform any operations with it, it's only used as an indicator for the player on the save file screen.

For game time, rather than tracking it using a timer that has to constantly run in the background, it can instead be calculated by storing the time elapsed in milliseconds since the game has started, which is then subtracted from the time elapsed at the time of saving to get the game time. These timestamps can be obtained using Godot's Time API.

Code listing 5.31 Tracking game time

```
private ulong _gameTime = 0;
private ulong _timeSinceLastSave = 0;

// The game time of the current save in seconds.
public ulong GameTime
{
    // Update _gameTime and _timeSinceLastSave, then return value.
    get
    {
        _gameTime += (Time.GetTicksMsec() - _timeSinceLastSave) / 1000;
        _timeSinceLastSave = Time.GetTicksMsec();
        return _gameTime;
    }
    // Sets the _gameTime and updates _timeSinceLastSave to begin
    // tracking time.
    set
    {
        _gameTime = value;
        _timeSinceLastSave = Time.GetTicksMsec();
    }
}
```

Through the use of properties and fields, the game time is calculated, and the various helper variables are updated every time the value is accessed.

5.6.3.6 Combining all serializers and outputting to file

With all of the above combined, using the JsonConvert.SerializeObject() method returns a string. Finally, to write the text output to a file, Godot's FileAccess API is used.

Code listing 5.32 The full method for creating a game save

```
public bool Export(string filePath)
{
    using (FileAccess file = FileAccess.Open( filePath,
                                              FileAccess.ModeFlags.Write))
    {
        SaveFileHeader header = new() {
            date = Time.GetDateStringFromSystem(),
            gameTime = Global.Instance.GameTime,
            mainScene = scene.MainSceneType,
            mainSceneDescription = scene.MainSceneDescription
        };
        file.StoreLine(JsonConvert.SerializeObject(header));
        file.StoreLine(JsonConvert.SerializeObject(GameData.Instance));
        file.StoreLine(JsonConvert.SerializeObject(scene));
    }
}
```

It is important to define the scope of the FileAccess by closing it around the ”using” keyword. As the API does not offer a method to manually close the file, it is instead closed when the object is disposed of, which is handled by the garbage collector in C# and isn't always necessarily when the method is finished.

5.6.3.7 Encrypting the save file

One of the JSON format's strengths is that it is human-readable. However, in this case, it is undesirable as players could easily edit the values in the file by simply using a text editor. To combat this, the save file can be encrypted to obfuscate the data. With Godot's FileAccess API, instead of Open, the OpenEncryptedWithPass() method can be used, which handles all of the encryption and decryption automatically. The method receives a string password upon opening a file that it uses to encrypt or decrypt the file.

Code listing 5.33 Encrypted file write

```
FileAccess.OpenEncryptedWithPass( filepath,
                                  FileAccess.ModeFlags.Write,
                                  "somePassword")
```

In the example project, rather than being hard-coded in, the password is stored in the Project Settings, which is accessed from the editor using Project > Project Settings...> General. By toggling on "Advanced Settings", custom fields can be added.

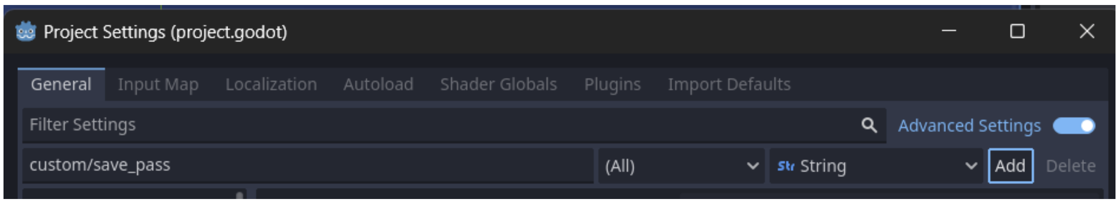


Figure 5.20 Adding a field to Project Settings

Here, the path "custom/save_pass" means a string variable is stored under the "custom" section, with the variable named "save_pass". This variable can then be accessed from code by calling the following method:

Code listing 5.34 Accessing a ProjectSettings variable

```
(string)ProjectSettings.GetSetting("custom/save_pass"))
```

It is important to note that, regardless of if the pass is stored directly in the code or in the Project Settings, it is not secure, as it will inadvertently be included in the executable. This means it can easily be obtained and used to decrypt the save, since all the necessary information is stored locally on the player's computer.

In this case, this is not an issue, as the goal is to merely make save editing slightly more difficult. Keep in mind, however, that with this method is not cryptographically secure, and can be easily cracked.

5.7 Exporting the game

Finally, to build a project into the final, distributed executable file, the game will need to be exported. Export configurations are stored in the "export_presets.cfg", which need to be configured for each project.

The Export window is found by navigating to Project > Export...In the window that opens, targeted platforms can be added by clicking Add.... This will add the selected platform to the list of presets, where one can then set various settings for that particular platform, or add other platforms. Note that at the time of writing (Godot 4.2.1), Godot does not currently support HTML export for Mono/C# builds of the engine.

5.7.1 Export Templates

When exporting for the first time, it is necessary to download export templates. The editor will display a warning, and an option to "Manage Export Templates". This option will open the Export Template Manager window, through which official templates can be installed.

After the installation is finished, the setup required is different depending on the targeted platform.

5.7.2 Export Settings

Each export preset contains an Export Path where the resulting executable will be created. This should generally be outside the root of the project folder.

Files that aren't considered Resources by Godot, i.e. ones that don't appear in the editor's FileSystem such as .txt, are not included in exports by default, and will need additionally specified in each Export preset. To do this, navigate to the "Resources" tab on the Export screen and add the appropriate filter to export non-resource files. This has to be done for each preset/targeted platform.

Next, the Options tab contain various settings for the export, which depend on the targetted platform. Some platforms also require some extra setup before being able to be exported to. Here is a brief overview of each platform:

Linux: doesn't require any extra setup.

The export is split into two files: the executable, and a .pck file containing the data of the game. By toggling the "Embed PCK" option on, the .pck will be embedded into the executable, resulting in a single file.

Windows: can be done without any extra setup. However, to change the executable file's metadata, such as its application name or icon, the rcedit tool needs to be installed and specified through Editor Settings at Editor > Editor Settings...at "export/windows".

Alongside the rcedit, the signtool can also be specified, which is used to sign the executable. More details on both of these processes can be read about in the "Exporting for Windows" page of the Godot documentation. [29]

Similar to Linux, Windows exports also separate the executable into two files: an .exe and .pck file. Toggling on "Embed PCK" will result in a single file.

macOS: devices running macOS will by default not run any apps that are not signed and notarized. To notarize an app, an Apple Developer ID Certificate is required. More information on this is available on the "Exporting for macOS" page of the Godot documentation. [30]

iOS: as of Godot 4.2, C# export to iOS is considered experimental. Projects need to target .NET 8.0 or higher rather than the .NET 6.0 used by default, as described in the documentation page on C# platform support. [31]

iOS builds cannot be created directly by the Godot engine. Instead, the iOS export template outputs an Xcode project, which then has to be imported to and built on a macOS device running XCode. More information on this is available on the "Exporting for iOS" page of the Godot documentation. [32]

Android: as of Godot 4.2, C# export to Android is considered experimental. Projects need to target .NET 7.0 or higher rather than the .NET 6.0 used by default, as described in the documentation page on C# platform support. [31]

To build for android, OpenJDK and the Android SDK need to be installed. Additionally keystore files need to be generated using the JDK. A separate keystore is required for debug and non-debug builds. These are then set to the export preset setting file, alongside their user/alias and password/keypass.

Note that the alias and keypass are both sensitive data. Because they are stored unencrypted as part of the export preset settings, contained in the “export_presets.cfg” file created at the project folder root, care should be taken to not publicize this data, such as not versioning “export_presets.cfg” using version control system to avoid exposing them in repositories.

5.7.3 Finalizing Exports

Once all Export presets are setup and ready to be used, the game can be exported to a platform by selecting its preset and using the Export Project...button in the Export window. When finalizing the export, the “Export With Debug” option is toggled on by default. This will export a debug build, which will print more detailed error messages to logs and commandlines, and contain fewer optimizations than a release build, which may cause it to run more slowly. As the name suggests, this is the build that should be distributed to testers.

To export to all platforms contained in the presets, the Export All...button is used. This will export all presets marked as “Runnable”. Each platform can only have one preset marked as Runnable. After clicking Export All..., the engine will then prompt you to select whether to export Debug or Release builds, which is the same as leaving the “Export With Debug” toggle in single exports on and off respectively.

Chapter 6
Conclusion

The example project was successfully designed and implemented. While there are still many missing features and unfinished systems, the project does a sufficient job of demonstrating the various capabilities and features of Godot, ranging from work with the editor, the various Nodes and their capabilities, the scripting API, C#-specific issues stemming from the GDScript centric design of the engine, as well as solutions using C# specific features that would otherwise not be available in GDScript.

The thesis has gone over the various tools and features used to realize the project, most notably a lot of work with the editor, and the creation and configuration of Nodes, which isn't necessarily apparent at first glance from simply opening the project files in the engine. The chapter on implementation also contains explanations behind the design of various systems and the interaction between the various classes and structures, as well alternative solutions to some of the problems discussed in the project, and why they weren't chosen in favor of the final, used solutions.

From a gameplay standpoint, the current state of the game is very much only a prototype of the presented ideas, as there is currently no goal, game over or victory state. As the full source of the project is available on GitHub, including code, image and audio assets, further work can be done by extending the existing codebase, which was written and designed with the intention of being expanded upon in the future.

Bibliography

1. *Most used Engines* [online]. 2024. [visited on 2024-03-10]. Available from: https://itch.io/game-development/engines/most-projects.
2. *Cross-language scripting* [online]. 2023. [visited on 2024-04-21]. Available from: https://docs.godotengine.org/en/4.2/tutorials/scripting/cross_language_scripting.html.
3. *What is GDExtension?* [online]. 2024. [visited on 2024-05-15]. Available from: https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html#supported-languages.
4. *C# basics* [online]. 2024. [visited on 2024-04-20]. Available from: https://docs.godotengine.org/en/4.2/tutorials/scripting/c_sharp/c_sharp_basics.html.
5. *Using an external text editor* [online]. 2023. [visited on 2024-04-23]. Available from: https://docs.godotengine.org/en/4.2/tutorials/editor/external_editor.html.
6. *Overview of Godot's key concepts* [online]. 2023. [visited on 2024-04-21]. Available from: https://docs.godotengine.org/en/4.2/getting_started/introduction/key_concepts_overview.html.
7. *BBCode in RichTextLabel* [online]. 2024. [visited on 2024-05-02]. Available from: https://docs.godotengine.org/en/4.2/tutorials/ui/bbcode_in_richtextlabel.html.
8. *Godot's design philosophy* [online]. 2023. [visited on 2024-04-21]. Available from: https://docs.godotengine.org/en/4.2/getting_started/introduction/godot_design_philosophy.html#object-oriented-design-and-composition.
9. *Observer · Design Patterns Revisited · Game Programming Patterns* [online]. 2021. [visited on 2024-04-21]. Available from: https://gameprogrammingpatterns.com/observer.html.
10. *GD0001: Missing partial modifier on declaration of type that derives from GodotObject* [online]. 2023. [visited on 2024-04-15]. Available from: https://docs.godotengine.org/en/4.2/tutorials/scripting/c_sharp/diagnostics/GD0001.html.
11. *C# signals* [online]. 2024. [visited on 2024-04-29]. Available from: https://docs.godotengine.org/en/4.2/tutorials/scripting/c_sharp/c_sharp_signals.html.
12. *Using InputEvent* [online]. 2023. [visited on 2024-05-07]. Available from: https://docs.godotengine.org/en/4.2/tutorials/inputs/inputevent.html#anatomy-of-an-inputevent.
13. *Singleton · Design Patterns Revisited · Game Programming Patterns* [online]. 2021. [visited on 2024-05-08]. Available from: https://gameprogrammingpatterns.com/singleton.html.

14. *Singletons (Autoload)* [online]. 2024. [visited on 2024-05-08]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/scripting/singletons_autoload.html#introduction`.
15. *Import process* [online]. 2023. [visited on 2024-04-15]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/assets_pipeline/import_process.html#importing-assets-in-godot`.
16. *File paths in Godot projects* [online]. 2022. [visited on 2024-04-15]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/io/data_paths.html#accessing-persistent-user-data-user`.
17. *RPG_project* [online]. 2024. [visited on 2024-05-15]. Available from: `https://github.com/itsBoopity/RPG_project`.
18. *Importing images* [online]. 2024. [visited on 2024-04-20]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/assets_pipeline/importing_images.html`.
19. *Shaders* [online]. 2023. [visited on 2024-05-12]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/shaders/index.html`.
20. *Shading language* [online]. 2024. [visited on 2024-04-21]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/shaders/shader_reference/shading_language.html`.
21. *Godot Hue Shader* [online]. 2020. [visited on 2024-05-16]. Available from: `https://gist.github.com/McSpider/c8c693f1065fdb5ec44c64cb9fdac340#file-hue-shader`.
22. *Godot engine godot-demo-projects* [online]. 2023. [visited on 2024-05-16]. Available from: `https://github.com/godotengine/godot-demo-projects/blob/master/2d/screen_space_shaders/shaders/BCS.gdshader`.
23. *Locales* [online]. 2022. [visited on 2024-04-10]. Available from: `https://docs.godotengine.org/en/3.6/tutorials/i18n/locales.html`.
24. *Internationalizing games* [online]. 2023. [visited on 2022-09-04]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/i18n/internationalizing_games.html#converting-keys-to-text`.
25. PLOEGER, D. *Provide a way to load resources without running scripts* [online]. 2022. [visited on 2022-09-04]. Available from: `https://github.com/godotengine/godot-proposals/issues/4925`.
26. BRODERICK L., et. al. *Deserialization risks in use of BinaryFormatter and related types* [online]. 2023. [visited on 2024-04-10]. Available from: `https://learn.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide`.
27. *C# Variant* [online]. 2023. [visited on 2024-04-11]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/scripting/c_sharp/c_sharp_variant.html`.
28. MING P., et. al. *CA2328: Ensure that JsonSerializerSettings are secure* [online]. 2023. [visited on 2024-04-11]. Available from: `https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2328#cause`.
29. *Exporting for Windows* [online]. 2023. [visited on 2024-05-13]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/export/exporting_for_windows.html`.
30. *Exporting for macOS* [online]. 2024. [visited on 2024-05-13]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/export/exporting_for_macos.html`.
31. *C#/.NET* [online]. 2023. [visited on 2024-05-13]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/scripting/c_sharp/index.html#c-platform-support`.
32. *Exporting for iOS* [online]. 2023. [visited on 2024-05-13]. Available from: `https://docs.godotengine.org/en/4.2/tutorials/export/exporting_for_ios.html`.