Design Rationale Document

CS 3307A Deliverable 3

By Jacob Couture - jcoutur9@uwo.ca
Member of Team Ornithophobia

Design Documentation

The overall goal of this project was to create a 2D platformer game where you try to blast yourself at high speeds into enemies to defeat them.

This project was split into three separate sections to be done by three different people: gameplay, level editor, and animation. I was in charge of designing and implementing the gameplay, and this section will explain how it works.

There are three main classes that make up the core gameplay:

- `PlayerController`: Handles movement and physics for the player.
- `EnemyController`: Handles states and signals for the enemies
- `LevelController`: Initializes and manages levels at runtime

All these classes are part of larger objects that contain multiple Godot nodes, but these classes are the root of these objects and control any child nodes.

PlayerController

This class mainly consists of a very large `_physics_process()` loop that runs lots of different calculations for the player's physics every frame. It's mainly just math, but it does have some interaction with the `Debug` class, which is used to track information about variables on a toggleable debug menu during gameplay.

The `PlayerController` class has one other function that interacts with other classes, and it is a signal receiver for colliding with an enemy's hitbox. This signal is emitted by the `EnemyController` class if the player is NOT moving fast enough to defeat it when they collide.

SlowMotionMeter

A class that acts as a child node of the player object. It allows the player to slow down time when holding down the SPACE key, making movement somewhat easier to control.

EnemyController

A shorter class that controls the logic of an enemy. It mainly sends out signals upon colliding with a `PlayerController`, either to let the `PlayerController` know it should bounce off the enemy, or to let itself know that it was defeated.

Rather than giving the `PlayerController` the variable that dictates how fast they must be moving to defeat an enemy, each enemy has its own variable for that. This would allow for easy creation of different enemy sub-types that could have different properties, such as one with a lower minimum kill speed or one with a higher minimum kill speed.

LevelController

This class handles both the setup and the managing of levels at runtime. Its primary purpose is initializing all level components that every level will share regardless of level content, such as UI elements, a pause menu, and a results screen. When switching to a level scene, the `LevelController` class acts as a Facade by initializing many complex classes with a single function call that only takes a file path for the level being loaded.

The `LevelController` also handles ending levels. When it is initialized, it checks the Level node's EnemyList to see how many enemies are in the level. Each time an enemy is defeated, the `LevelController` receives a signal from the `EnemyController` that will decrease the number of enemies remaining by 1. When it reaches 0, the `LevelController` will broadcast a signal to the results screen that includes all relevant information about the level, indicating that the level has been completed and the results screen can now be shown.

LevelTimer

An extension of the built-in Label class that acts as an in-game level timer that counts up. Used as a part of the level UI.

Debug/DebugController

The `Debug` class is a singleton that holds a dictionary of values that any class can add to. This dictionary can be retrieved with a getter function and is used in `DebugController` to display the values in the dictionary on-screen at runtime.

Design Patterns

The following is a list of all the design patterns used in this project and who was responsible for implementing them:

- Singleton - Jacob Couture/Chun Ho Chan
- Facade - Jacob Couture
- Observer (partially implemented) - Jacob Couture/Chun Ho Chan
- Mediator - Chun Ho Chan
- Prototype - Chun Ho Chan
- Command (partially implemented) - Chun Ho Chan
- State - Sameer Bugti

The rest of this section will go into detail on my own implementations of the design patterns I used.

Singleton

I implemented the singleton pattern through the `Debug` class. I knew from the very start that being able to see values on-screen while testing the game was going to be very important. Being able to see how values and states change during gameplay naturally can reveal issues that would not be easy to see otherwise, so I decided to implement the singleton pattern to allow any class to register one of its variables to be tracked on-screen.

Implementing this pattern was difficult at first due to GDExtension (the tool we used to integrate custom C++ classes into the Godot Engine), though after a couple of weeks I found out how to properly make singletons with it.

Facade

When developing the `LevelController` class, the plan was to have it be the parent node of every other level component. All the other level components would exist ahead of time, but they would not be properly initialized. Rather than saving all these components to a single scene, the `LevelController` class instantiates them all at runtime, which both saves us a lot of copy and pasting when creating levels, but also ensures that it will be able to access every child node it needs access to.

Observer

Partially implemented/sort of scrapped? Initially, I had planned to update my `Debug` class to also implement the observer pattern, but I eventually decided to scrap it for two main reasons:

1. Time was running out, and there were more important features to get done
2. It didn't really make a whole lot of sense to use the pattern with it

The reason it didn't make a lot of sense comes down to how the variables in the `Debug` class are updated. Since the main purpose of it is to track variables that can change lots during gameplay, most (if not all) of the values that are tracked get registered in the `Debug` class from a process loop, which means they are sending their information to the class at least 60 times per second. This results in the dictionary in the `Debug` class always being up to date, and since there's only one class that actually does something with the dictionary it doesn't need to be alerted every time a property is updated.

That said, the observer pattern is semi-present all around the project. By using Godot's signals, we can notify classes that are connected to that signal when the signal is emitted. This works exactly like the observer pattern does, but since a part of it is handled entirely by Godot behind the scenes, I only consider it a partial implementation.

Testing Report

Before getting into the test results, I need to bring up why Google Test was not used for our test cases. Google Test is not compatible with Godot/GDExtension due to the nature of the Godot Engine. To be able to use Google Test, we would have had to recompile the entire engine with our code integrated into it, which is a very difficult and time-consuming process that we were not equipped to do.

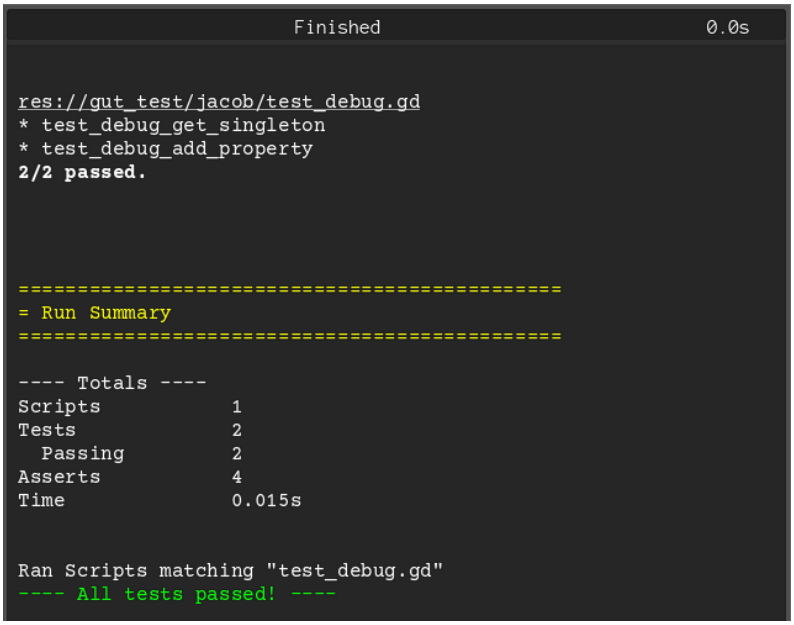
As an alternative, we used a community unit testing framework called Godot Unit Test (GUT). This allowed us to write our tests as GDScript files (Godot's native scripting language), and while it allowed us to run unit tests on our C++ files, there were still some complications due to GDExtension.

Debug Tests

`test_debug_get_singleton()`: The objective of this test case is to ensure that the instance of the `Debug` class never changes.

`test_debug_add_property()`: The goal of this test case is to test that properties can be added and retrieved from the `Debug` dictionary without any issues.

Testing Outcomes:



LevelController Tests

`test_level_controller_set_empty_level()`: This test checks to see how the `LevelController` handles loading a level with no metadata, ensuring that it creates proper placeholder data and doesn't leave anything null, as that could cause issues when trying to access the metadata later.

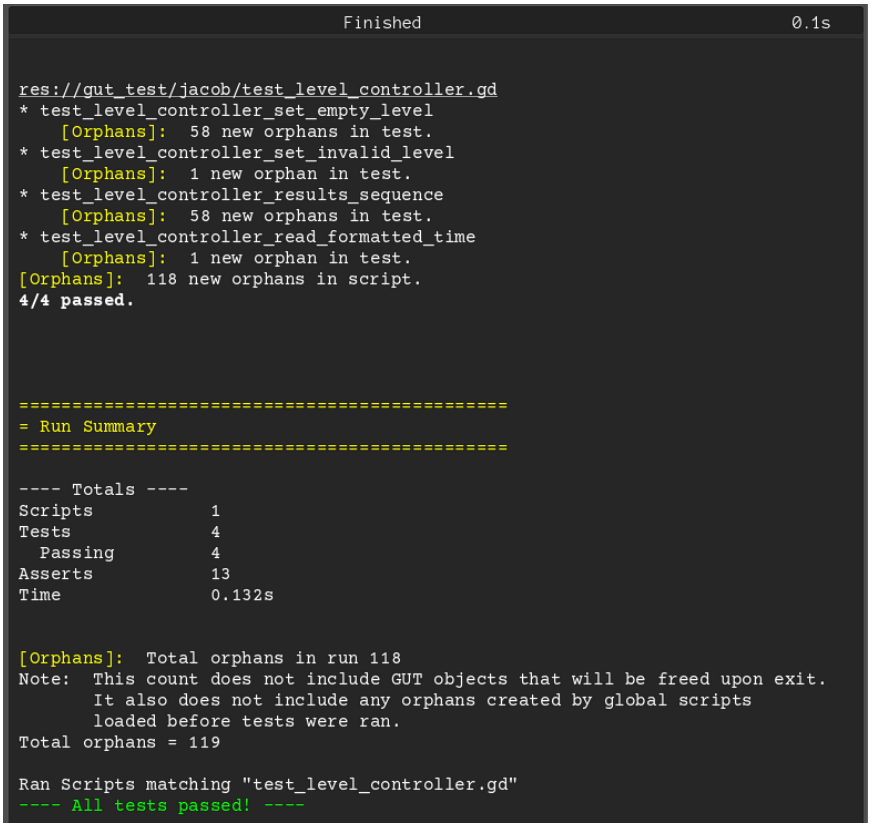
`test_level_controller_set_invalid_level()`: This test checks to see how the `LevelController` handles loading a level that doesn't exist. The ideal outcome is that it catches the error and throws its own error, as the alternative would be crashing the program with no errors at all.

`test_level_controller_results_sequence()`: This test mocks enemies in the level being defeated to ensure that the `LevelController` properly initiates the results screen sequence.

`test_level_controller_debug_reload_scene()`: This test case mocks user input to attempt reloading a level. The goal of this test is to make sure that the `LevelController` performs the correct actions when receiving user input.
* This test case doesn't function and is disabled due to GDExtension not interacting nicely with GUT. See comment above function in `game/gut_test/jacob/test_level_controller.gd`

`test_level_controller_read_formatted_time()`: This test case ensures that my helper function for reading time from a string works the same in its C++ implementation and GDScript implementation (this function is only used once in C++ but many times in GDScript, and unfortunately only GDScript can call functions from the C++ files, not the other way around).

Testing outcomes:



Challenges

The largest challenges in the project remained the same as the previous challenges. The lack of debugging for our C++ code massively slowed down development, the overall lack of GDExtension documentation made it super difficult to find solutions to any of our problems, and Godot still sat there providing much easier and efficient solutions to our issues if we just switched from using C++ to using GDScript or C#.

Now at the end of the project, I can reflect on a few final challenges. The first has been present all along: GDExtension just hates working with anything. Using GDExtension caused almost all our problems with this project, and in hindsight it was likely not the best decision to use it. It doesn't even have comments in its source code, and it has been a massive pain to work with.

The second was the time crunch. As the final due date came closer, we ran out of time to implement certain things. The largest features that got cut were a leaderboard for best level times, the settings menu, and the ability to save user data. That second one was hard to let go of since being able to save your game is such an important feature, but we decided to focus on implementing the system to save and load custom levels

instead, as that felt like a more important feature than your own best time saving. Our system is currently set up to easily be able to work with saved user data, but with the short amount of time we had to get everything done we had to make the tough decision to cut it.

All that said, the project still came together quite well. This was a massive learning experience for me, and I'm very glad I decided to take this path. My time working on this project has given me extremely valuable experience with C++ and Godot now, and I suspect I will be coming back to both again in the future.