CS3307A Assignment 3

Project Documentation

by Chun Ho Chan

(ccha232@uwo.ca)

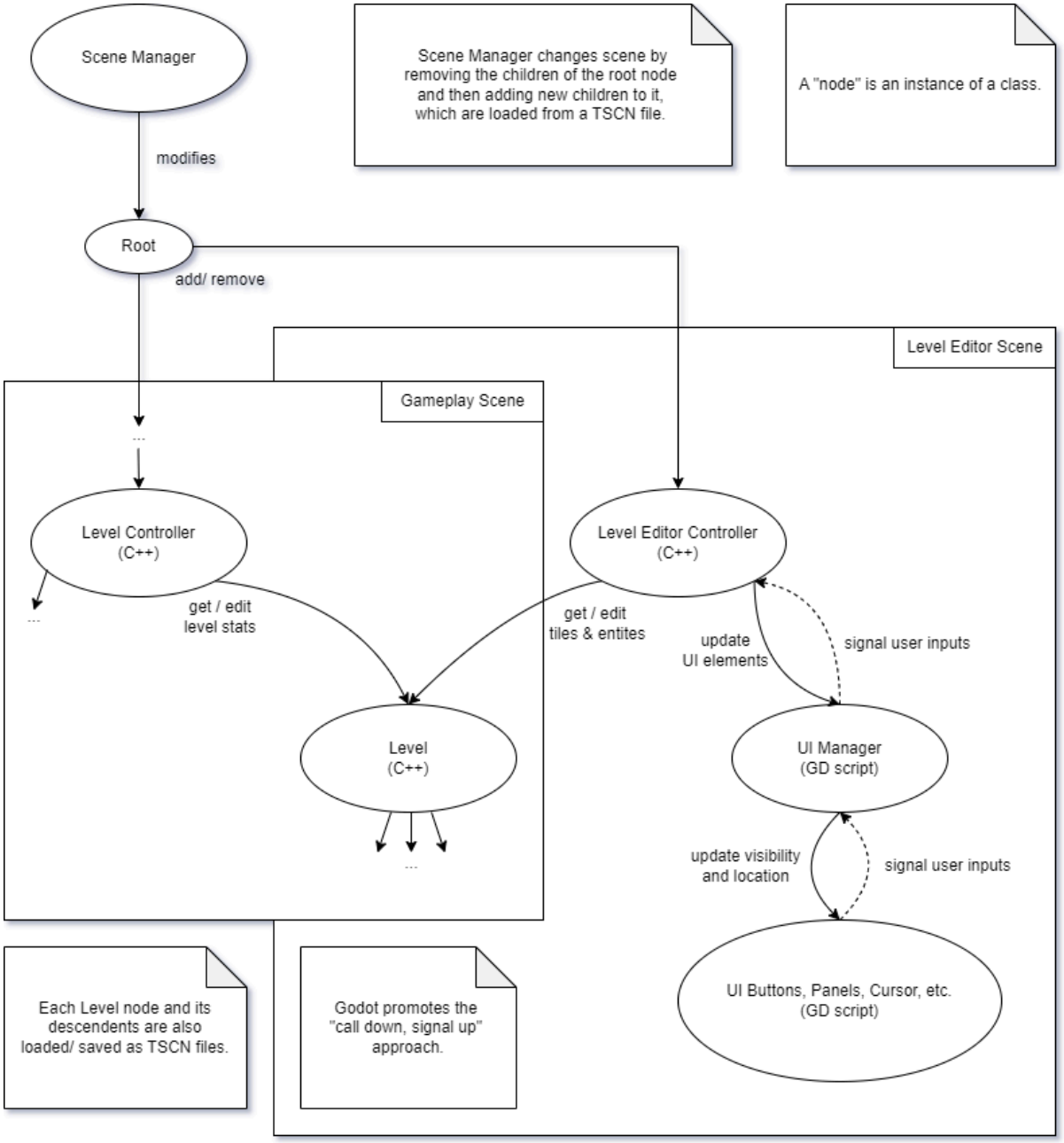
Team Ornithophobia

Design Documentation

System Architecture

The Level Editor Controller and Scene Manager are critical components of our overall game project.

The following (not UML) diagram illustrates how they interact within Godot:



Note:
Details about Godot's scene structure can be found in the Godot 4.3 documentation.
Details about the Level Controller can be found in Jacob's documentation.

OOD Principles

Our project code follows the SOLID design principle, for example:

1. **Single Responsibility** is achieved by a clear separation of the responsibilities of classes. For example, Level only stores/ updates level data, UI Manager only receives user inputs and updates UI elements, and LevelEditorController only manages the communication between the two, without any direct control over level data or UI elements.

2. **Open / Closed Principle** is achieved by having all level entities inherit Godot's **Node2D** class, and the **Level** class stores level entities as a list of **Node2D**. This allows new types of level entities to be added without modifying the **Level** class.

3. **Liskov Substitution** is achieved by having all screens (e.g. gameplay screen, level editor screen) inherit Godot's **Node** class. This allows all screens to be added/ removed from the scene root just like any normal node.

4. **Interface Segregation** is achieved by having only small and specific interfaces. The **IPrototype** interface is tailored to the **Level** class so that **Level** is not forced to implement any unnecessary functions.

5. **Dependency Inversion** is achieved by injecting the **Level** object into **LevelEditorController** at runtime. This also allows stub **Level** objects to be easily injected into **LevelEditorController** in unit tests.

Design Patterns

(Note: same as Deliverable 2)

Mediator Pattern (MVC)

The Mediator Pattern solves the issue of complex communication between Godot built-in classes (written in GD Script) and our custom classes (written in C++). It ensures that a Controller class handles all the cross-language function calls, thus decoupling our other custom classes from the Godot UI classes. This reduces the coupling and complexity of our custom classes, as only the Controller class has to use the Godot API.

In our case, we follow the Model-View-Controller structure. All of the UI inputs (such as a button being pressed) first notify the Control class (in GD Script), then the Control class emits a signal in Godot which the Level Editor Controller class (in C++) listens to, and finally, the L.E. Controller calls the corresponding Level class to edit its stored data.

Singleton Pattern

Having only one Scene Manager instance is critical, as it is responsible for loading/ unloading scenes and tracking all previously loaded scenes. If more than one Scene Manager is in use, it would create dangling pointers (and crashes the game by freeing null pointers or having memory leaks) and an invalid scene transition history.

Applying the Singleton Pattern to the Scene Manager class fixes all these problems by ensuring that only one global instance is in use.

Prototype Pattern

The Prototype pattern solves the issue of the complex initialization of Godot objects. It allows us to copy all data from a pre-initialized "template" Level object, created in the Godot Editor, and later modify its data in runtime based on the player's input. This approach not only skips the 20+ steps required to initialize each Level object, but it also ensures that all Level objects reuse existing references to share expensive data elements (e.g. the huge PNG sprite sheet of tiles).

Other Loosely-Followed Patterns

We partially followed the Observer Pattern by using Godot's built-in Signals subscription system. We partially followed the Command Pattern by using Godot's Callable class to store a function call (i.e. a function pointer and a list of parameters). For more details, please refer to the Godot 4.3 documentation.

Testing Report

Testing & Mocking Framework

We are using the Godot Unit Test (GUT) addon to implement our unit tests. It has features equivalent to GTest & GMock and direct integration with the Godot engine. The professor approved it as an alternative to Google Test.

We cannot use Google Test because it is incompatible with GDExtension, our development framework. GDExtension compiles our C++ code as a dynamic-link library (DLL) for Godot, without compiling the Godot engine from source. The DLL is loaded into Godot during runtime and uses its internal functions. Since Google Test cannot access Godot's internal functions, it also cannot execute our source code.

Godot Unit Test works by loading our compiled DLL into the Godot engine and using Godot's internal functions to compare the expected outputs. By validating the correctness of our compiled DLL, we can infer the correctness of our C++ code.

Unit Test Coverage

Key classes:

- **Level** (100%)
    - All functions are tested at least once.
- **LevelEditorController** (33%)
    - The key functions are tested at least once; the other 66% of its functions are just facade/ redirected calls to **Level**, hence unit testing Level is sufficient.
    -
- **SceneManager** (100%)
    - All functions are tested at least once.
- **EdwardMain** (100%)
    - All functions are tested at least once.
- **IPrototype** (0%)
    - **IPrototype** is not tested as it is a pure abstract class/ interface for **Level**.

Explanations

1. In GUT, the equivalent function for *EXPECT_CALL()* is:
stub(stub_object.function_name).to_return(stub_value).

2. Every function has a corresponding test case, and within it are multiple *assert_eq()* to check several edge cases, such as uninitialized/ missing data and invalid data. All unit tests have passed for the classes listed above, and GUT's output log can be found at:
"{project_folder}/game/gut_test/xml/output_edward_chan.xml"

3. Mocking is used in *LevelEditorController.undo_action()* and *LevelEditorController.redo_action()* by first supplying **LevelEditorController** with a mocked **Level** object, which has the *Level.get_level_info()* function overridden to always return a known constant. Then, *LevelEditorController.undo_action()* or *LevelEditorController.redo_action()* eventually calls the mocked function and returns a value. An *assert_eq()* checks if the returned value matches the known constant. Mocking **Level** allows unit testing of **LevelEditorController** regardless of whether *Level.get_level_info()* is implemented.

Challenges and Solutions

(Note: same as Deliverable 2)

Lack of official documentation

Since "GD Extension for C++" is a new replacement for "GD Native", the previous Godot C++ plugin, it has zero official documentation available online. Whenever we need to use a Godot built-in function/ feature, we have to search through the uncommented source code of the GD Extension. Sometimes, even that is not enough (i.e. we reach an "extern C" function) and we have to search online on Godot's GitHub Issues page, Godot official forums, and even the wiki page of "GD Extension for Rust" to find any related answers.

There is no magical solution to this challenge except spending half of my project development time on online research. Towards the end of our development, we had a pretty solid idea as to which pointers should/ cannot be freed, and memorised most of the Godot API calls by heart.

Lack of guidance

Since our project uses Godot and GD Extension, our TAs were unable to offer us much help beyond suggesting OOP design patterns and teaching us basic C++ syntax (which is very helpful in that regard). As for questions related to game development/ design, or using the "GD Extension for C++", we have to answer them ourselves by relying on our past experiences with the Godot Engine.

Fortunately, I have a friend who is an experienced game developer. I have been calling him weekly to ask him for advice on game development/ design and his experience with the Unity Engine and the Unreal Engine proved invaluable. Thanks to his advice, we now know how to better optimize our game and provide a more enjoyable gameplay experience.

Lack of developer tools

"GD Extension for C++" requires a lot of boilerplate code in our custom classes. It is a time-consuming and repetitive process to set up a new class, and any mistakes/ typos made in that process lead to unexpected and unexplained crashes.

Hence, it motivated me to spend a week creating our own one-click compiler, using a mix of Bash scripts and Python scripts. We have been using that for a month and our number of compilation errors and unexpected crashes decreased by about 80%.