Games PRogramming Custom Project Distinction Report

Or how I learnt to stop worrying and Love C

STEWART GRANT

101082708@student.swin.edu.au

Graphical user interface

Description automatically generated

**Technologies, Tools, and Resources used:**

* [Visual Studio Code](https://code.visualstudio.com/) w/[SGDK](https://marketplace.visualstudio.com/items?itemName=zerasul.genesis-code) & [C/C++](https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools) Plugins
* [Java Runtime Environment](https://www.java.com/en/download/manual.jsp) (Links are provided on the Repo for SGDK however best to go to the source)
* Git command Line
* Asprite
* Audacity
* [Ohsat Tutorials](https://www.ohsat.com/tutorial/)
* Stack Overflow
* [Sega Genesis Development Kit (SGDK)](https://github.com/Stephane-D/SGDK)
* SGDK Help Discord

**Setting up the Environment (For Windows)**

The source code of the project is written in C however a special compiler will needed to create the roms that the console can read. Java RE should be installed as some tools that come with the framework requires it to run them. These tools are essential for converting assets into formats the Mega Drive can process. After the SGDK has been extracted to a location, the library needs to be compiled. Call the following command in the command console: “SGDK\_PATH\bin\make -f SGDK\_PATH\makelib.gen”. Replace the SGDK\_Path with the location of the extracted folder.

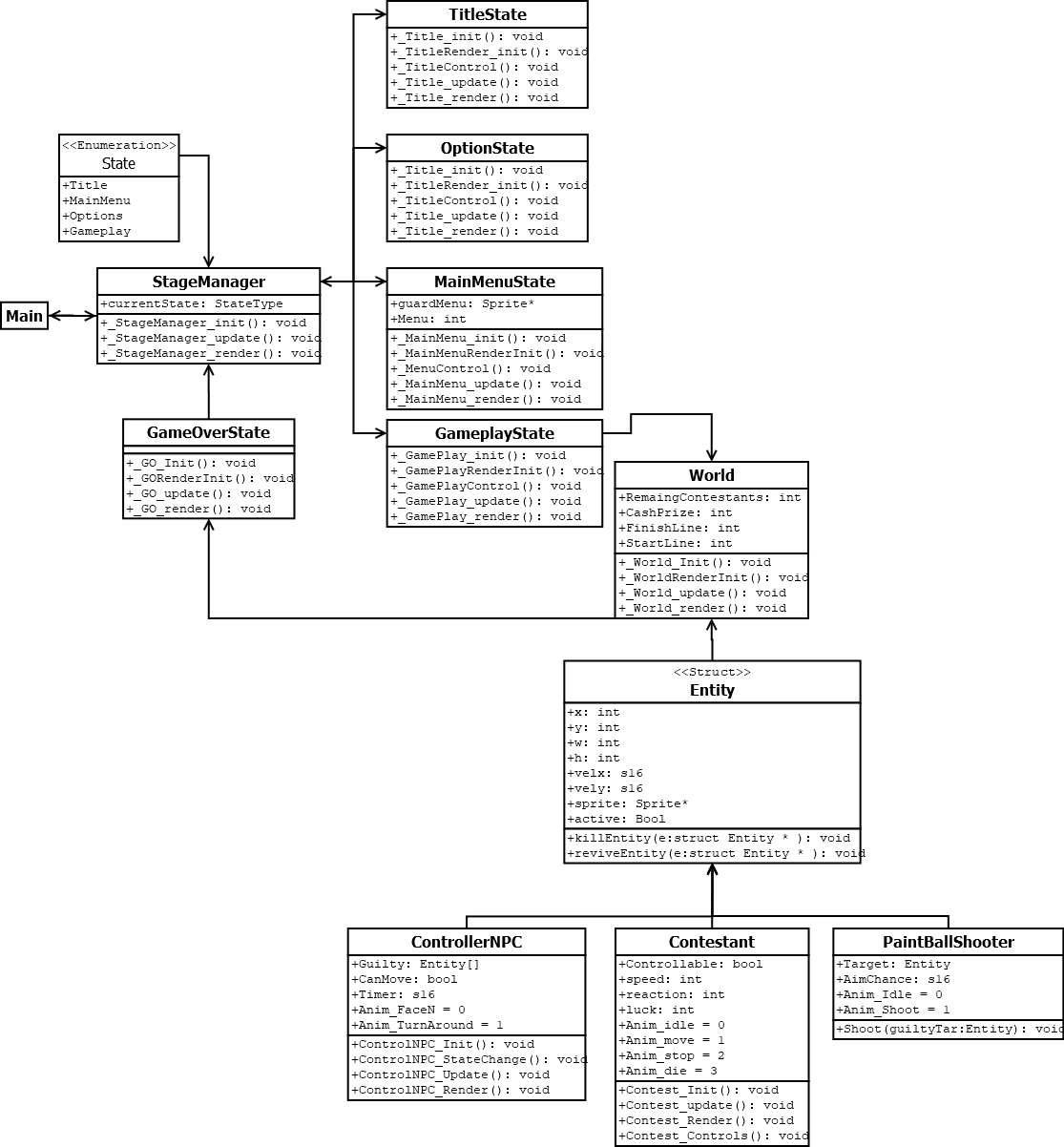
Visual Studio Code has a plugin that simplifies the process by adding shortcuts to compile roms and build project files. Install it, make sure the integrated Terminal is set to gens-code and the environment should be good to go. Roms that are compiled will be found in the out folder named “Rom.bin” It is possible to install it on other OS’s however no attempts were made to learn how. The main Github Repository includes ways to install SGDK on Mac OS X, Linux and through Docker.

**Tasks undertaken**

**Designing the Program:**

Before attempting any programming, a basic UML is drafted of the design of the project. This UML graph will help with development of the system and ensures that an outline is followed however there will be some deviations from the intended design. Using the UML graph as the design, the steps involved would be to:

1. first get the Main Game Loop running,
2. To get the Game States functional and ensure each C script is interacting with the game loop
3. Get the Non-Gameplay specific states working
4. Get the world and the NPC Controller functional
5. Get the Contestants and Guards working.
6. Get the Messaging System working to send messages across the system.



**Implementing the Game State:**

The next step taken was to get the Game Loop and States working via enums and through various scripts. Since SGDK runs in C, many luxuries of C++ are absent so alternative methods are required. Since Class encapsulation was not an option, I opted to use a function pointer that takes in a Enum variable called current State to select the appropriate Game State.

A screenshot of a computer

Description automatically generated with medium confidence

The Game Loop doesn’t need to quit like earlier pass tasks so it doesn’t need a Boolean to exit the loop. However, SGDK specific functions need to be called beforehand. To get Game States working, an Update and Render class are created in the StageManager script and are called in the main game loop.

The various game states are now assigned their own scripts and are filled with the init, init\_Render, update and render script accordingly. In C, I thought it functions with similar sounding names would not be an issue since they are in their own script however some research and compilier errors taught me otherwise. To solve that, each function is given a unique name related to the script they are a part of.

Before attempting to get the game working, the other game states are implemented. The Menu, Title screen and Options are implemented to ensure the Function pointer in Stage Manager is finding the appropriate function.

**Implementing the Traffic Controller:**

The traffic controller determines the flow of the state of the game. It’s designed to call when game objects are allowed to move. Should a Game Object move during a period when they shouldn’t they are out of the game. The Controller uses a simple timer to call the state change upon it will change its sprite and then call a second timer to reset the movement state for all game objects. To tell the other objects to stop moving, the Controller will call a function in the level script that broadcasts to all Contestant Game Objects to stop moving.

**Implementing the Contestants:**

The Contestants have a simple Ai and Object which is to reach the finish line, move only when allowed, stop when told. Starting off their implementation, 32 were initially created however the amount was later dropped to 12. On initialisation, a struct array is created taking in a size specified in the level script. The loop will proceed to instantiate each Contestant by assigning their position, velocity, sprite, active + Safe tags and 3 stats. When the level calls recieves the call to stop, the contestants use their reaction stat to stop their movement. The reaction integer adds stakes to the project since not one contestent will react the same to another. When the Traffic Controller calls the stop command, it sends the current global move boolean to the level which is turn is passed to the Contestents. The Contestants upon receiving the boolean will run a check to see if they react in time. If the Contestant doesn’t roll a value below 75 out of 100, they are passed to the Guard for removal. Once they cross the finish line, the safe boolean is checked.

**Implementing the Basic Shooty Boys:**

The guards were designed to be nothing more than Game Object removers. Their design has been slightly altered from the UML to include an extra necessary functions as they now now have a Init function. Otherwise, their responsibility is to destroy instructed contestants. The Shoot Command looks gets an array of the remaining Entities and checks to see if they have broken the Don’t Move rule. If said entity has, the Entity Kill function is called to remove that contestant from the game.

**What I found out developing this project (Or how much I miss the OOP Concepts):**

When developing the custom project, applying many of the concepts taught in the unit were a challenge since C forgoes many of the luxuries C++ and OO Languages provided so alternative methods were needed to emulate programming patterns and concepts learnt in the unit.

Graphical user interface, application

Description automatically generatedBefore even a line of code was typed for the custom project, I already had a Software design planned and the steps written. The UML shown above while early was what I predicted to get the program running and demonstrating concepts of the unit. As the program was being developed, some deviations from the intended steps were made as new discoveries which prompted investigating (Sprite Limitations, Difficulties with C, System crashes).

As I was developing for the custom project, performance was already a big factor to consider due to the limited hardware spec. As a result, I did run into said hardware limitations which both confused and excited me since it was confusing why something acted this way but excitement to talk about the limitations and how to work around them. The most notable was the issue with trying to draw too many sprites on screen and learning about how the system renders it’s visuals.

To explain what is going on in the previous image, the idea was to have over 32 Contestants on screen to have enough variation with results however, the mega drive can only display 80 Hardware sprites or 20 Sprites per scanline. I say I had 32 sprites however that’s not the entire truth since the Mega Drive renders sprites via 8x8 tiles which create the entire sprite. For example, a sprite of Sonic would consist of 1x2 tiles to create his sprite while a sprite of Dr Eggman may be 4x8 Tiles. Simply put, I wanted to draw 32 sprite/tiles, but I was attempting to draw 32 sets of 4x4 Tiles so the options are to shrink the sprites or reduce the amount of entities on screen which I chose the latter. However, doing the math suggests that even with my current amount of entities on screen, I am still surpassing the sprite limit which confuses me. Were I to have more time, I would use the Performance of the System for a research topic as it would be a great way to not only gain an understanding of the console but learn where performance and optimization techniques can be applied.

Since classes are non-existent in C, the best alternative was the use of Structs to simulate objects and individual scripts personally were seen as classes. To try and limit how much a script could access, scripts were only allowed to include scripts that they absolutely need values from allowing me to reduce the amount of unnecessary inheritance.

**Issues and Challenges:**

Some of the biggest issues faced developing the custom project was not the limitations of the hardware but adjusting to C’s lack of OOP concepts and having to find Procedural Programming alternatives to the concepts taught from both OOP and this unit. An example of this would be my attempt to incorporate the ECS and the Component Pattern into the project and while the project compiled, the game object behaviour was completely wrong.

Graphical user interface

Description automatically generated

Debugging the issue is difficult due to a lack of a debugger so I can only speculate why it behaved this way.

That’s not to say there were no hurdles with the systems limitations. As mentioned above, I managed to hit the sprite limit and had to choose between less sprites or smaller sprites which I chose the former to avoid drawing new sprites. Unlike SDL, the Mega Drive can only display a limited number of colours on screen which is a restriction that must be respected. While I’m not close to hitting the colour limit for the system, I’m aware that the system can only display a maximum of 16 colours on screen colours per palette where there are only 4. Currently, I am using 2 of the palettes in the custom program to display the colours. There are from the Background Tiles and the Contestant Sprites. The Swinburne Logo also has a colour palette, but it is overwritten once the game state changes from title screen to menu since it and the background sprites share the same palette. If I were not to assign the palette, sprites will display with the incorrect colours like so.

A screenshot of a video game

Description automatically generated with medium confidence

Additionally, SGDK comes with a very powerful sound driver that allows higher quality sounds to play on the system and for some reason, the sounds would not be played. With more time, I would inquire online to find out why sounds are not playing and if it was a implementation issue, learn how to properly implement sounds.

**Screenshots (Post Repository creation):**

**Repository Link:**

<https://github.com/FusionAura/SwinburneCustomProgram>

**References:**

[1] <https://megacatstudios.com/blogs/retro-development/sega-genesis-mega-drive-vdp-graphics-guide-v1-2a-03-14-17>