Games Design Report

Callum Darling

May, 2020

1 Introduction

Neon Knight, is an action platformer developed with C++. It uses the cross platform media library SFML and the entity component system library EnTT. It takes inspiration from the games of 80s and early 90s such as "Castlevania".

2 Changes from plan

In the original designs for the game I had envisioned an Entity Component System driven design. The core of the design and the final product are still very similar although there have been a number of smaller changes to the structure of components and Entities.

The largest change has been to one of the design patterns. I will elaborate further on this in Section 3. Having read up more on the singleton pattern I decided that the context I had planned to use it in was not appropriate for its actual intent. While a useful pattern it should be used sparingly. I replaced this pattern with the sequencing pattern Game Loop. This was due to the fact that I was already implementing a game loop and I didn't feel any of the other design patterns fit well with the overall design.

When implementing the game it became clear that more entities and components were required than were accounted for in the initial plans.

The designs had 8 components, the final product has 10, these are due to oversights related to the media library being used. The way that objects are drawn to the screen in SFML is sufficiently different between some kinds of objects that is was prudent to separate this functionality into different components. Aside from this the planned components covered all needed eventualities.

The designs had 11 entities, the final product has 14, these similarly were created to make interfacing with SFML easier. For example the original designs had not accounted for the ability to draw shapes (rectangles, circles, etc.) directly on the screen without using a sprite with a texture. To account for this a new entity was created. The distinction between Entity and Enemy also proved to not be useful in this context. While the enemies are still built from a different base to the other entities, there was no reason to keep the distinction between the types of object as this can be expressed through components within the entities.

Almost all of the character designs and user interfaces have changed from the initial design, as more complete and polished versions have been produced.

3 Design Patterns

The three patterns that I chose were the architectural pattern Entity Component System, the design pattern Factory and the sequencing pattern Game Loop.

3.1 Replacement of Singleton with Game Loop

As mentioned in section 2 I was initially planning on using the singleton design instead of game loop. There are two classes the deal with files in the design, ResourceHandler and LevelHandler, initially I believed that due to this file access I did not want multiple instances of these classes existing at once, so I thought that it would be appropriate to use a singleton pattern. Having made this choice I began researching implementations of this pattern to try and find a good way to use it in my code. While researching I came across many warnings against the overuse of this pattern and examples of its use in areas that it was not really required. Looking at this and also situations that it is more appropriate to use this pattern I realised that it may not be appropriate for this project. ResourceHandler and LevelHandler are implemented in such a way that there is never more than once instance of either of them operating at once and there is no way for the same file to be simultaneously accessed. This proves that the singleton pattern would have been unnecessary.

3.2 Entity Component System

I used a C++ library called EnTT to implement ECS. The Components can be found in the src/components. Each component is a simple file containing a struct containing variables for the data that the component will encompass.

Entities initialised and stored in a registry (the registry for this project can be found in game.h and game.cpp). The entity class is simply a reference that can be used with the registry to access components that have been emplaced into the entity itself. The entities are initialised in the EntityFactory (EntityFactory.h and EntityFactory.cpp), e.g. they are created and have components emplaced into them. The 'System' in ECS is encapsulated into the game class, the components and entities are created externally to this but the actual operations of the game happen here.

I chose to use the Entity Component System (ECS) architectural pattern for several reasons. In past projects completed using a traditional inheritance based object oriented approach I had run into a trouble with large inheritance trees. I knew that I was going to be using entities of some kind in the game whichever approach is took so it seemed reasonable to use the system that was oriented towards this approach. ECS systems have a reputation for being resource efficient and fast to execute, as the project was going to be written in C++ I thought this would make a good combination.

3.3 Factory Pattern

The factory pattern is implemented by the EntityFactory.

As ECS is being used as the architectural pattern a factory pattern is a logical continuation of that, the entities need to be created. As there are many types of entities and the type of entity that may is needed may change at runtime, a factory pattern fits this need well.

While the main function of the factory pattern is to create an abstract method for creating entities, to maintain flexibility I also make the individual functions for creating the individual entities public as I wanted to (and do) access these independently from the abstract method. This adapted factory implementation was what worked best for this project.

3.4 Game Loop

Game loops are an essential part of most games, therefore this pattern is also a logical inclusion into this project. The pattern is split into three main parts. All of these parts are processed inside game::run() (game.h, game.cpp), where the actual loop is held

- **Processing user input**, this is handled by several functions, ProcessEvents() is called as part of the game loop. This function processes game events and calls various functions depending on the event. If the user is using the keyboard to input into the game ProcessEvents() calls handlePlayerInput(), if they are using the mouse it calls handleMouseInput(). These three functions make up the processing user input part of the game loop.
- **Update**, this is handled by the update() function, depending on the state the game is in (main menu, level 1, level designer, etc.) it will call a different update function. for example if the game is in a level it calls updateLevel().
- **Render**, this is handled by the run() class itself, after calling the appropriate functions to process input and update the game, it draws all the visible entities to the screen.

4 Games Concepts

While the game does feature some physics (collision detection, gravity) and basic AI (simple pathing and enemy response to player) The main concept used was a level designer.

As the game is a 2D action platformer with levels made up of defined blocks and enemies a level designer is a useful feature that is not only good for the player but the developer as well. Since there was a plan from the start to create a level designer the game could be designed around that detail. The levels were loaded in from a file from a very early point of the development, this meant that later in the process when it was time to implement the level designer itself it was easier. The level designer is similar to a level in the game except that the user is able to place the blocks and enemies instead of them being pre-placed.

The level designer relies on LevelHandler::saveLevel() a function which accepts a map containing coordinates and the type of entity, it then writes these to a file. The rest of the designer functionality takes part in game. The updateDesigner() function deals with the user's input, the keyboard is used to switch which entity is being placed and the mouse is used to place and remove entities. If the left mouse button is being clicked it calls addBlockToEditor() which adds the block to the screen and adds it to the map with its coordinates. If right click it calls removeBlockFromEditor(), which removes the entity from the screen and the map. Since what is on the screen correlates exactly with what is in the map, when saveLevel() is called the level is saved as the user expects.

5 Security Requirements

There are several sections to security which I will cover:

1. Player Data
2. Code Security
3. Other Concerns

5.1 Player Data

The game does not gather any information about the player and does not attempt to access anything about the player or their machine. It does not interface with the internet or attempt to send data back to the developer. Therefore this area of security is not relevant to the game.

5.2 Code Security

This game is licensed under the GNU General Public License Version 3 (GPLv3). It will be freely available on github.com for anyone to access, copy, and distribute the source code according to the terms of the License. As such code security is not important to this project. However, I will touch on some methods / precautions that could have been taken if code security was a concern.

- **Decompilation:** C++ is already reasonably decompile resistant and this scales with the size of the application, by removing all debug information and other evidence from the release build you can ensure that decompilation is as difficult as possible
- **Using Safe APIs:** Make sure that you are using the latest standards and ensure that you are not using any unsafe APIs. When used incorrectly some APIs can create security vulnerabilities.

5.3 Other

If a hostile party wished to interrupt the game while it was running there are several surfaces that they can use to attack the foremost of those being the levels. The level files are loaded into the game and used then casted to int, causing the game to crash by putting unexpected input into one of those files would be trivial.

They could also attack the resources, (textures, sounds, etc.)

The solution to these problems would be input verification, and if necessary hash verification of files.

6 Reflections

In hindsight i would do some things different:

Libraries: The majority of this project was programmed from scratch, including some quite complex systems. Undoubtedly there are libraries for some of these systems which would save a huge amount of time and effort that ultimately didn't add much to the final product. An example of this is a physics system, while the game only makes use of simple physics such as gravity and linear motion, it would have been easier (and possibly more efficient) to use an already available library. Using libraries like this would have freed up developer time for features that could have enhanced the final product.

Research: While I did research on similar games and potential target audiences for my game, more emphasis could have been placed on researching implementations of certain features. A great example of this is the collision algorithm. I rewrote the collision algorithm 3 times trying to remove bugs and increase efficiency. the first two collision detection algorithms were close to $O(n^2)$ time complexity, it is certainly possible to reduce this with appropriate algorithms and data structures. In future I would make sure to do further research of critical systems like this.

Planning: While some things were initially planned I started prototyping before the full structure of the code had been planned, this led to systems having to be reworked later in development to fix the more concrete plan. Better planning would have saved a lot of time reworking code.

Time: Unfortunately I had less time that I had hoped due to COVID-19 disruption.

7 Test Strategy

Unfortunately due to the currently ongoing COVID-19 pandemic I was unable to conduct as many user tests as I had initially planned for.

Those tests that I did manage to carry out were conducted in the standard manner:

- Get the tested to start playing with as little background as possible
- Ask the tester to think out loud as they play the game
- After 5 minutes stop them and ask questions about their experience
- Record their answers
- Use answers from multiple tests to inform changes to the game

The questions I decided to ask my testers were as follows:

1. How would you describe your overall experience?
2. What did you enjoy?
3. What did you dislike?
4. Did anything surprise you?
5. Did anything frustrate you?
6. What were your thoughts on the interface?
7. What did you think of the level design?

While the number of tests I could complete was very low they still gave me some insight into things worth changing about the game. Both users I tested with described the overall experience as enjoyable, I did not take any action as a result of time. One of the users said they enjoyed the platforming most while the other said they enjoyed the combat. Both users said they disliked the lack of instruction on controls, mechanics, etc. In response to this I added some information on controls to the start of the first level. One user said they were surprised by the difference between the levels, the other said they didnt find much surprising, I took no action in response to this. neither user said anything frustrated them, I took no action. Both users had some specific small comments on the interface design which I took on board. Both users were satisfied with the level design.

The user testing was successful in my opinion as it lead to the game improving, although I would have much preferred to have a larger number of testers to get a move even view.

8 References