Advanced Programming

Yonah Thienpont
yonah.thienpont@student.uantwerpen.be

Universiteit Antwerpen — August 21, 2022

Introduction

In my last project I did not implement any of the design patterns correctly, neither did I have a concrete division between Logic and Representation. These being important parts, I decided to start from scratch, keeping only very basic fragments of my old project.

1 Gameplay

As for gameplay, I have implemented every feature exactly like described in the assignment.

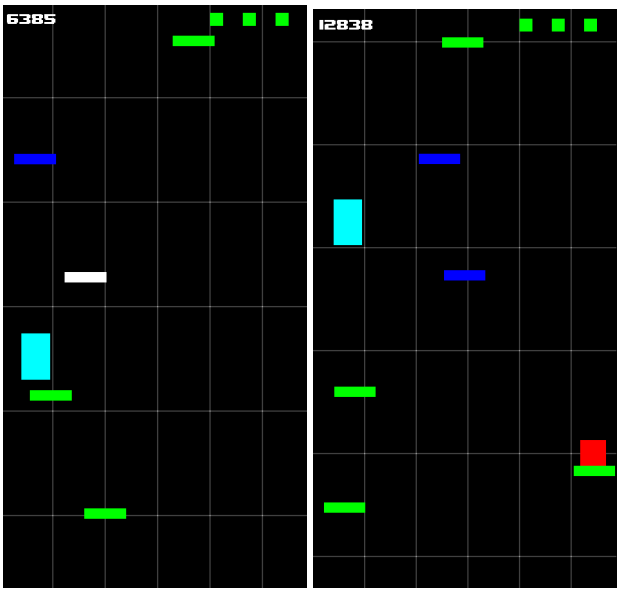

Figure 1: Screenshots

2 Technical Requirements

2.1 Design

The project was designed with the example class hierarchy in mind. I wrote three additional classes, not described in the asignment: *Moving, Living* and *Effect*. The first two are to avoid code duplication for all moving entities and all entities with a health bar. The latter is to create an indication whether the player or an enemy have been hit by a projectile, making the screen flash red or white respectively.

2.1.1 Design Patterns

Model View Controller (MVC): There are two classes to ensure the correct working of the MVC-pattern: *Model* and *View*. Model handles all game logic, interacting with other entities and other behaviour, while View makes sure the visual representation is updated accordingly. I created a View class for every Model class, but in hindsight a simple Platform class with possible arguments like: 'Blue', 'White', etc. would probably have been sufficient. The way it is implemented right now helps to make things clear, but it expands the code by quite a bit, also requiring code duplication. The way the assignment describes the working of the View class calls for an implementation similar to mine.

Observer: The View object is updated using the Observer pattern, with the Model object as Subject. The Score was to be designed, following the same pattern, but because the working of Model and Score classes differ so much, I have implemented the pattern with respect to the Score class implicitly. As World had to check for all score deltas and they are both part from the Logic library anyway, I found a true Observer pattern to be overcomplicated. My method for updating the score is to make the Score object handle every score delta, making the code simpler. Letting World handle the score deltas is too much functionality that could easily be attributed to this class.

Abstract Factory: This pattern should be implemented correctly, creating Logic entities with the corresponding View attached.

Singleton: No further explanation needed.

2.2 World Generation

For generating the world I used a simple system. World has a member called *difficulty*, used for calculating which platform to spawn and how far platforms should be spaced. This variable is decreased by one, every 10.000 units the player travels. A random number $r \in [0, \text{difficulty} - 1]$ is generated and passed to a switch statement with a normal platform as default. Meaning the more difficulty decreases, regular platforms become less likely.

2.3 Code Quality

The polymorphic design in this project is good overall, reusing as little code as possible. The only problems I encountered were with destroying the entities that were out of bounds. I wanted to create a function, similar to the ones I have in place currently, making use of the polymorphic qualities of my Model classes, by having a vector with Model elements to check which one should be removed. However I could only get this done with a cast, while I believe this is possible with raw pointers. Since only two of these vectors had the exact same conditions for deletion, I didn't bother looking for a better solution. I also feel like polymorphism fails with the design of the Bonus class, a class with a million members, only used in very specific situations, feels cumbersome. The polymorphism in the Enemy class is also less than optimal, as I misinterpreted the working of the regular enemy and only added the shooting property later. Another problem with the code is the use of 'magic numbers'. A lot of constants in this program should be worked away. That being said, the overall logic of the code is easy to follow and I don't think a lot of magic happens. The code is easy to understand, without the need for a lot of comments (except some vague functions).