Project Assignment

Advanced Programming

2021-2022

1 Introduction

This year's project consists of designing and implementing an interactive game, inspired by Doodle Jump[1], in C++ and using the SFML[2] graphics library. The main goal of this project is to demonstrate that you are able to create a well-designed architecture, fully utilize advanced C++ features and provide high-quality code that implements the requirements. Of course, it's great if you add creative extra features or fancy graphics and animations, but make sure the basics work well first and you have a good, extendable codebase to work with.

2 Gameplay

The game consists of three main entities: a player, various platforms and potential bonuses. The player is always jumping up and down automatically, without any user interaction. The user can however move the player character to the left or to the right by pressing the A or left arrow and D or right arrow keys respectively. If this movement is done correctly, the player will jump on top of a higher platform, helping it move further up in the world. Hitting a platform from below allows the player to completely pass through it unhindered. But when landing on it from above, the player will bounce off the platform, allowing it to reach a new maximum height. The goal of this game is therefore for the player to climb as high as possible, without falling off the world. When the player falls to the bottom of the screen, where there is no platform to stand on, the game ends. A current score is shown on the top of the screen (or anywhere else) during gameplay and an all-time high-score is shown when the game ends. Each time the player reaches a new maximum height, the current view of the world is moved upwards such that this newly reached height is in the middle of the screen. All platforms that are no longer within view are removed from the world and new platforms are generated for the section of the world that has recently come into view. In theory, this world generation extends upwards infinitely, as long as the player does not fall off the platforms.

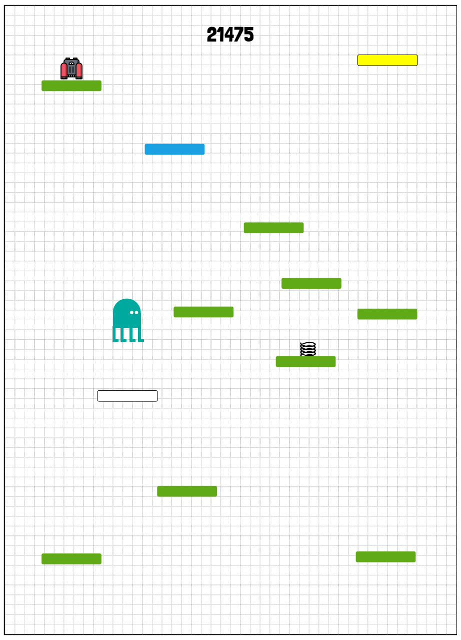

Figure 1: Example Game Design

There are 4 different platform types that can be generated:

Static platforms: occur the most often and simply stay in a fixed position within the world. (Green)

Horizontal platforms: move back and forth horizontally across the screen. (Blue)

Vertical platforms: move back and forth vertically between a fixed minimum and maximum height. (Yellow)

Temporary platforms: disappear after they have been jumped on once. (White)

On top of some of these platforms, bonuses can also be present, with which the player can interact to gain one of two benefits:

Springs: When jumping on one of these, the player is given an extra boost, making the jump 5x as high.

Jetpacks: These allow the player to shoot up by a much more considerable distance (up to you to decide).

[1] `https://en.wikipedia.org/wiki/Doodle_Jump`
[2] `https://www.sfml-dev.org/`

3 Technical Requirements

3.1 Design

An important part of this project is creating a flexible design of the game entities and their interactions, as well as the correct use of design patterns. You'll need to design a class structure for the different game entities that facilitates this and keep in mind that your game should be easily extendable. An essential aspect of this design is that there needs to be a clear separation between game logic and representation. Classes that contain game logic should not contain any code related to representation or the other way around. By having this clear separation, you could easily make an alternative representation using a different graphics library, without needing to change any code related to the game logic. To further facilitate this, you'll have to encapsulate the game logic into a standalone (static or dynamic) library using CMake[3] and link your representation code with this library in order to generate the final binary. In theory, you should be able to compile this logic library without having SFML installed. A possible (incomplete) hierarchy could for example look like the following:

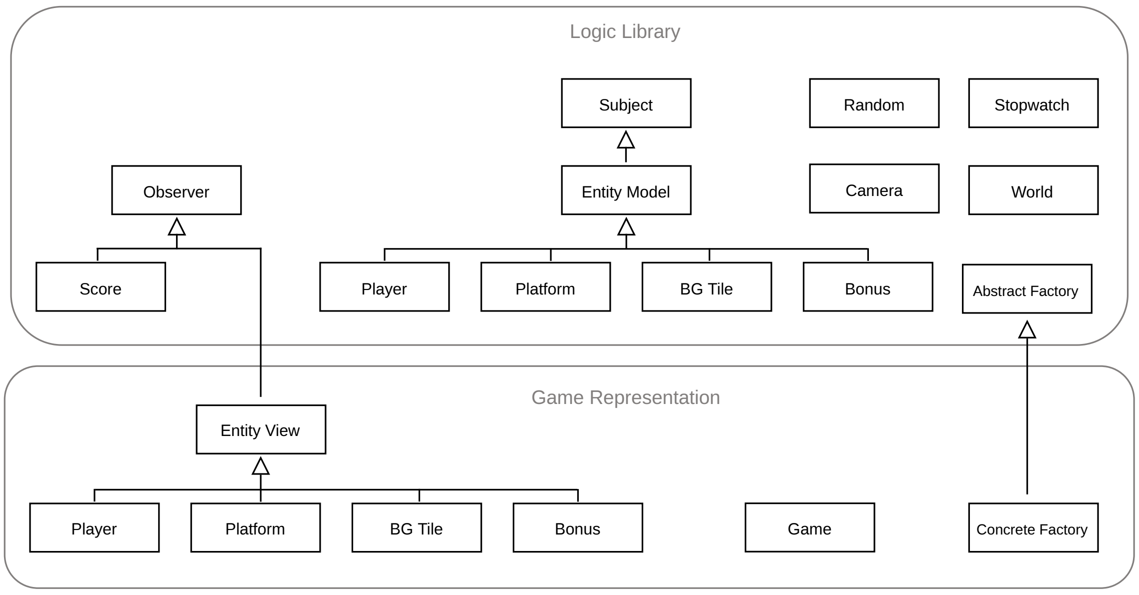

Figure 2: Example class hierarchy with clear separation between logic and representation

You are free in how you design this class structure, but the following key classes need to be present, as well as the use of the design patterns discussed in section 3.1.1:

Game: As part of the game representation, this class is responsible for setting up anything that is not related to the core logic of the game. Examples of this include creating the SFML window, receiving user interaction and passing this to the World, instantiating concrete factories and running the main game loop.

Stopwatch: This class keeps the difference in time between the current update step (tick) and the previous one. This is used to ensure that the game logic runs at the same speed, regardless of the speed of the device it is running on. You're not allowed to use busy waiting to slow down devices that are running too quickly, the frame rate needs to be dynamic. The only exception to this is that you can cap the frame rate at a certain maximum value (for example 60 FPS), to match the maximum refresh rate of your display. To implement this class, you must use C++ functionality, not the SFML Clock class.

Camera: As the world is dynamically created and destroyed based on the current position of the player, the current view of the player needs to be explicitly modelled, which is done by the Camera class. The entire world should be modelled using a normalized coordinate system, which extends infinitely in one direction. The camera is then used to project coordinates that lie within a rectangle defined over this world space to an independent coordinate system that has fixed boundaries. It also needs to rescale these coordinates such that they can be used as pixel values within the SFML window. This functionality again needs to be implemented manually, without relying on SFML utilities.

[3]`https://cmake.org/`

Random: This class is used to generate all the random numbers you'll need throughout the game. Your world generation needs to make use of these random numbers to ensure that platforms are not always in the same positions, as well as generating a completely different world each time your game is run.

World: All entities are stored in the world, which is responsible for orchestrating the overall game logic and the interactions between the entities it contains, such as the creation and destruction of entities and collision detection between them. You can detect these collisions using basic intersecting rectangles. There's no need to focus on this too much, such as making it work with more complex shapes or predicting collisions. But also don't use SFML utilities here, since this class is part of the logic library.

BG Tile: These form the background for your game by tiling them in a grid. In order to have these tiles scroll together with the rest of the world, they will also have to be dynamically created and destructed or automatically recycled from the bottom to the top.

Bonus: A bonus should (temporarily) modify the state or behaviour of an Entity in a generic and self-contained way, such that you can easily add a new bonus type, without necessarily needing to change the interface of the Entities themselves. This logic should therefore not be encoded as a *set_spring_bonus* method in either World or Player for example, but rather in the Bonus class itself or a purpose-built one with a suitable life cycle. How you implement this specifically is up to you, either by designing a unique solution or by making use of existing design patterns.

3.1.1 Design Patterns

You will need to incorporate the following design patterns in your design:

Model-View-Controller (MVC): This pattern is used in order to clearly model the separation between game-state, graphical representation and the logic of the game. In this pattern, the World can be seen as an *Entity Controller*, or you could have multiple controllers in your design to better delegate responsibilities. You will need to use the **Observer** pattern for updating the *View* when the *Model* state changes. By attaching the *View* observers to the *Model* subjects directly when they are created in your concrete factory, you can separate the logic from the SFML representation completely transparently.

Observer: In addition to using the Observer pattern for updating the corresponding *View*, you also need to apply this pattern for computing the current score of the game. This score depends on the current height of the *Camera* view, how many and which bonuses the player collected and finally it is reduced for every time the player jumps multiple times on the same platform. You also need to make sure that a different amount is added or subtracted, depending on which type of bonus or platform it is triggered by. Each different type should result in a unique score delta.

Abstract Factory: This pattern is used to provide an easy interface which the World can use to create new Entities, without it needing to be aware of how to create instances of SFML-specific *View* classes. The logic library defines a simple abstract factory interface, which is adhered to by a concrete implementation in the representation code. Finally, the Game class provides a pointer to this concrete factory to the World, which can then use it to produce Entities that already have the correct *View* attached.

Singleton: The Stopwatch and Random helper classes need to be implemented using the Singleton pattern. This ensures that only one instance will be present during the execution of your program and they are easily accessible to all classes that will need to use them.

3.2 World Generation

As mentioned during the gameplay section, your game will need to contain an infinite world generation procedure, which in theory allows the player to continue playing indefinitely. The game does become more difficult as the player progresses through the game however. This is done by having the platforms become less frequent and using more advanced platform types, so jumping onto higher platforms becomes more difficult. During world generation, you'll need to keep into account that it should always remain possible for the player to reach a higher platform at all times, so there is a limit to how difficult the game can get. When platforms disappear from view, they will also need to be destructed properly, such that your program does not keep using more and more memory.

3.3 Code Quality

Below you can find a list of things that need to be present in your code to improve its quality:

- Use **namespaces** to clearly divide modular sections of your code.
- Include **exception handling** to catch and deal with possible errors, such as the absence of required files.
- Proper use of the **static**, **const** and **override** keywords where they can be applied.
- Make sure to avoid memory leaks by explicitly creating **virtual destructors** where necessary.
- Always explicitly initialize **primitive types**. (Hint: check these last two with *valgrind*.)
- Avoid unnecessarily copying objects where they can be passed as a reference or pointer.
- Refrain from relying on **dynamic casts**! This usually means your design is lacking proper polymorphism.
- Avoid duplicate code, solve this by using better **polymorphism** or **templates**.
- Use **clang-format** with *this configuration* to format your code.
- Write proper **code comments** and **API documentation**.
- Use of **smart pointers** throughout the whole project is obligated. This is used to test your insight on where to use unique, shared or weak pointers, depending on the type of ownership. No raw pointers are allowed, except in certain design patterns where the use of smart pointers is prohibitive. But if you need them, please discuss this with me and provide your reasoning first. Passing references is also perfectly fine; objects don't need to necessarily always be pointers.

4 Practical Information

4.1 Additional Resources

Below you can find some useful extra resources to help you get started on your project:

- *SFML Game Development* book: highly recommended to get started on working with SFML and some general game concepts, such as how a main game loop works. The pdf can easily be found online, but let me know if you have trouble finding it.
- *Head First Design Patterns* book: contains an explanation for all the design patterns mentioned in this assignment. Again, the pdf can easily be found online, but there are also more than enough other free resources available that explain these patterns well.
- *The C++ Programming Language* book: contains everything you may need to know about how C++ works to complete this assignment and much, much more.
- An example of the gameplay of the original *Doodle Jump* game can be seen here:
  `https://www.youtube.com/watch?v=wjofzwaC_Oo`
  Or you can play it yourself at the link below:
  `https://cdn.wanted5games.com/games/doodle-jump-new-en-s-iga-cloud/index.html?pub=157`

4.2 Grading

The grading of your project will consist of five separate criteria:

- **40%:** Core game requirements: you have a basic (working) implementation that implements all the gameplay elements.
- **40%:** Good design and code quality: you have properly implemented the design patterns and you make good use of polymorphism in your well-designed class hierarchy.
- **10%:** Project defense: 3 minutes of gameplay demonstration and 7 minutes of discussion on design choices and implementation details.

- **10%:** Documentation: report and comments. In your report of around two (A4) pages, please include an overview of your design choices and use this as an opportunity to convince me to give you good grades for your project. If you are able to explain your choices well, this can also result in higher grades for the other criteria.
- **10%:** Bonus points: you can earn these by implementing creative extra gameplay mechanics, making the game look and feel extra fancy or making correct use of additional design patterns. These extra points are simply added to the grade of your project, so your total becomes $\min(40 + 40 + 10 + 10 + 10, 100)\%$ in case all parts of your project are perfect. If you added any of these extra features to your project, make sure to also document them thoroughly in your report.

4.3 Submitting

Take the following things into account before submitting your code:

- Your code should **compile and run successfully on the reference platform at the university computer labs**:

  Ubuntu: 18.04, **SFML:** 2.4, **CMake:** 3.10.2, **G++**: 8.0.1, **Clang**: 6.0.0
- This project has to be completed **individually**, plagiarism will not be tolerated. You can of course freely discuss design decisions or implementation problems with other students.
- Create a private repository on GitHub and invite **thomasave** as a collaborator. Frequently commit code increments and set up CI to automatically test whether the project still compiles successfully. The final commit of your project must show a **successful build on a CI platform that is linked to your GitHub repository**. We recommend you use CircleCI[4] for this, since their free plan allows for considerably more than enough weekly builds for this project and it's easy to set up. You are also allowed to use an alternative CI platform, but make sure that I can see the build results for each commit on GitHub and the build configuration files (e.g. *.circleci/config.yml*).
- If you have any questions, don't hesitate to contact me (Thomas.Ave@uantwerpen.be) or ask a question on the designated forum on Blackboard.
- The project deadline will be in **January 2022**. The exact date will be announced later on Blackboard.
- The final project needs to be submitted on **Blackboard**, on **GitHub** and by **mail** to (Thomas.Ave@uantwerpen.be) and (Jose.Oramas@uantwerpen.be)

Good Luck!

[4] `https://circleci.com/`