Project Assignment

Advanced Programming

2022-2023

1 Introduction

This year's project consists of designing and implementing an interactive game inspired by Meat Boy, the predecessor of the popular Super Meat Boy[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. It is perfectly sufficient to create a game design based on coloured rectangles, but some basic textures go a long way into making it look nice.

2 Gameplay

Upon first launching the game, the user is presented with a menu containing a list of levels to select from, by either using the mouse or keyboard (your choice). Such a level contains three main entities: a player, walls and a goal. The player consists of a red cube-shaped character, which the user can control by moving left, right and jumping off walls in order to reach the goal object. After reaching the goal, a transition is made to the next level. The gameplay is characterized by fine control and split-second timing due to the very fast movement of the player. Each time the player reaches a new height above 80% of the visible screen, the current view of the world is moved upwards at the same speed as the player, such that the player stays at 80% height or below, unless the current world view has already reached the maximum height of that level. If the player falls back down into a region of the level that is now outside the world view, the player dies and the level resets from the beginning. In some levels, the world view also gradually moves up as time progresses, forcing the player to reach up to a safer area more quickly.

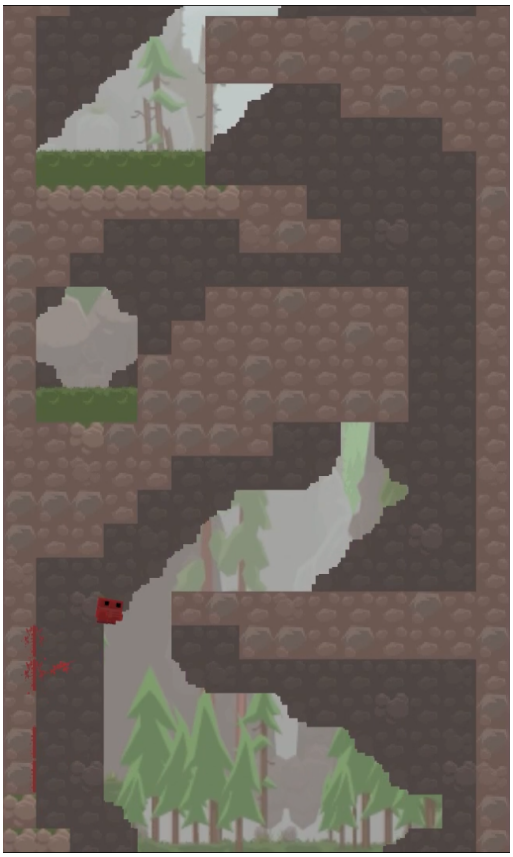

Figure 1: A gameplay example where the player jumps off a wall to reach a higher platform.

2.1 Player Movement

In order to reach higher, the player can jump vertically against a wall and press the space key, resulting in being launched upwards at a 45° angle away from the wall. The player can use this to repeatedly jump between walls on opposite side of each other, or they can press the arrow keys or A/D to immediately move back to the wall they just jumped off, thereby only needing a single vertical wall to jump higher. The speed of the player is represented by a horizontal (v_h) and a vertical (v_v) component, on which various forces are applied through a positive or negative acceleration (a). One such force is that of gravity (g), which is constantly applied to gradually slow down the upwards vertical movement after a jump, and will eventually cause the player to fall back down. You also need to implement a terminal velocity, which limits the horizontal and vertical speed to a maximum value. While the user presses the left arrow or A key, a horizontal acceleration is applied on the player, causing it to slow down if they were already moving to the right or start moving to

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

the left otherwise. The opposite is true of course while pressing the right arrow or D key. The force applied by jumping can be simplified by setting the vertical velocity to an initial fixed value upon pressing the spacebar, except when jumping off a wall, where the same velocity also needs to be applied horizontally. Jumping is of course only possible when the player currently collides with a wall from the side or from above. Hitting a wall from the side reduces your horizontal velocity to zero, while hitting a ceiling from below negates any vertical velocity. When landing on a floor from above, the vertical velocity is set to zero and the horizontal velocity is maintained as long as the player explicitly presses a key to move in a certain direction.

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:

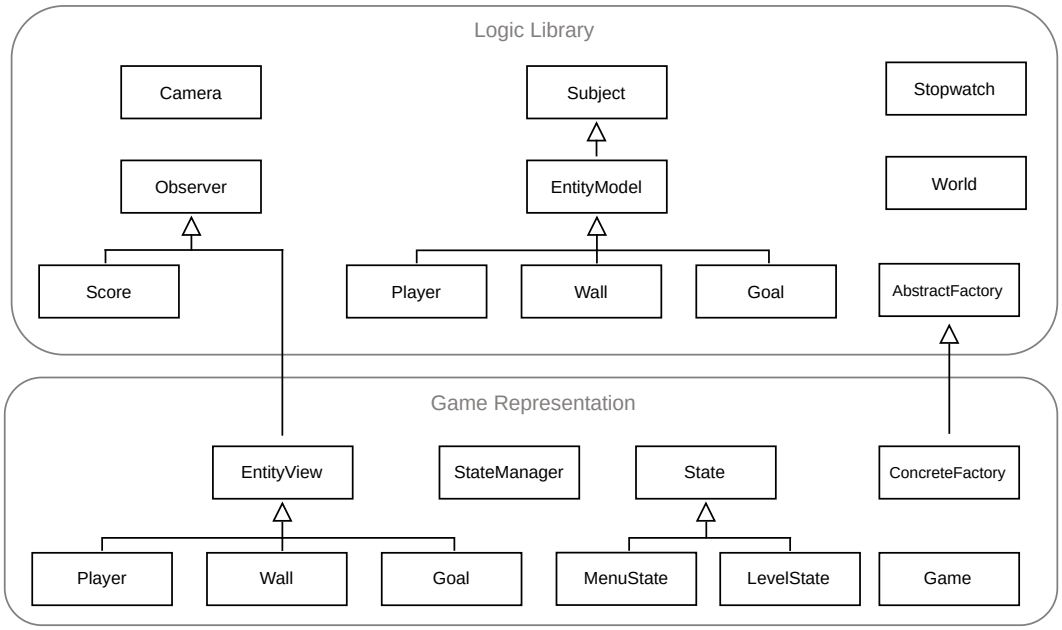

Figure 2: Example class hierarchy with clear separation between logic and representation

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, running the main game loop and setting-up the StateManager (see section 3.1.1). Other responsibilities may be delegated to the StateManager or concrete States, such as instantiating a concrete factory and processing user interaction. Note that if the user is currently playing a level, the game representation can translate specific key presses to actions in the World, such as *move left*, *move right* or *jump*. But it should not be responsible for how these actions influence the actual game logic.

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

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

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.

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.

Camera: The Camera class models an explicit view of the World that is currently visible to the player. The positions of entities in a World should be modelled using a normalized coordinate system, with a level width specified by [-1, 1] and height by [-1, h], where h is the height of the furthest Wall, while ensuring that vertical distance is proportional to horizontal distance. The camera is then used to project coordinates that lie within a rectangle defined over this world space to an independent coordinate system with a smaller vertical range, effectively translating the coordinates based on the height of the current World view. It is this height that is increased when a player exceeds 80% of the current view or automatically as time progresses, while keeping the player y-coordinates in [-1, h]. Everything in this rectangle will be drawn on the screen, while other entities remain out of view. To actually draw these entities, the Camera also needs to scale 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.

3.1.1 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* that manages the interactions between entities, or you could have multiple controllers in your design to further delegate responsibilities. Your *Model* is then responsible for holding any data related to a certain entity and to provide some methods that manipulate this data, while the corresponding *View* is responsible for what this entity looks like and for drawing this to the screen.

Observer: 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. A *View* can then receive an event for every update step (tick) to make sure the *Model* is drawn on the window. To make this easier, I suggest you hold a (smart) pointer to your window object in your *View*.

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 helper class needs 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. You could additionally opt to encapsulate your *sf::RenderWindow* in a singleton instead of storing a reference or pointer to it as member in the *View* classes.

State: This pattern is used to facilitate the transition between the level-selection menu and the actual levels with gameplay, as well as between different level-instances after reaching the goal. A *StateManager* (sometimes called the *Context*) is used to orchestrate the registration and execution of different *States*, in our case a *MenuState* that is responsible for the level selection and a *LevelState* that handles the actual gameplay. In particular, the *StateManager* will start with an initial state of *MenuState*, which will register a transition to a specific *LevelState* when a level is selected from the menu. This *LevelState* will transition back to *MenuState* when the escape key is pressed, or to a different *LevelState* when the goal is reached. You could also include a *PausedState* in a similar fashion with a more complex *StateManager* that can switch back to an existing state, but this is not required for this project.

3.2 Level Loading

Your game needs to support the loading of multiple levels, which can be selected from using the menu that is presented to the user when starting your program. This list of levels needs to be dynamically populated based on the presence of configuration files that specify the properties of each level. You are free to design the specification for these configuration files, using for example *JSON*, *XML* or even your own custom file format. It should be relatively convenient to add new levels or modify existing ones by changing these files. You need to create at least one level where the world view moves up automatically and one where this only happens due to the player reaching a new maximum height. It's nice if you can design some fun and challenging levels, but the only requirement is that you make sure that each gameplay feature can easily be demonstrated.

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 objects by reference is also perfectly fine, this does not necessarily always need to happen through the use of pointers.

4 Practical Information

4.1 Additional Resources

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

- The *SFML Game Development* book: highly recommended for getting started on working with SFML and some general game concepts, such as how a main game loop works. There's also *SFML Game Development By Example*, which is a similar book that explains the same concepts by guiding you through the process of creating snake clone. The pdfs for both 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.
- You can play the original *Meat Boy* game yourself at the link below:
  `https://archive.org/details/flash_meatboy#`

  Or you can watch a complete playthrough here:
  `https://www.youtube.com/watch?v=Bze5Wx9-H0s`

Note that the gameplay features you need to implement are more limited than in the actual game this project is based on, but you can use it as an example to better understand the required features explained in this document. Do try to more or less match the movement speeds of the original game in your implementation.

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 defence that will be organized during the examination period: 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: 22.04, **SFML:** 2.5.1, **CMake:** 3.22.1, **G++**: 11.3.0, **Clang**: 14.0.0

- 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.
- Accept the GitHub Classroom assignment and push your project to the associated repository. Make sure to include your name and student number in the *README.md* file. 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 2023**. 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/`