Khemin Van Gestelen January 2022
s0205633

Project Advanced Programming: Space Jump

In this document I want to describe the process of working on this project and point out some features/implementations that I am proud of or that have a clever solution to them. For the most part I have kept myself to the given project assignment and added extra features.

First of all the class hierarchy is mostly based on the given example in the project assignment. In some cases I chose to add an extra class in order to correspond more clearly to the MVC pattern. To clearly separate the logic library and the visual representation, the code is separated into two libraries (Logic and Visuals).
The Logic library contains all the computational code for Entities, the Score, the World, … . The Visuals library contains classes that visually represent Objects that have to be displayed on screen. In this library I added an extra class (ScoreView) that holds a visual representation of a Score object in order to display it on the screen. The Game class is also located in the Visuals library. It will draw the window, run the game loop and render all sprites.
The only way these two libraries are connected is through the Entities and Score. Each Entity (and/or Score) holds its corresponding visual representation as a data member ('view').

Next, an important aspect to point out is when visual representations are updated. In most cases a View class object holds one or more sf::Sprites. In the case of ScoreView these are a sf::Text and sf::Font instead.
In most cases the SFML data members are left unchanged when the corresponding Entities are updated. The only exceptions to this rule are BonusView and PlayerView. This is because of the animation these sprites have. The current frame of these sprite is updated as frequent as possible, i.e. when the corresponding Bonus, respectively Player, is updated.
In all other cases the SFML data members are updated just before they are rendered. This is to reduce the amount of calculations to a minimum and also because of convenience. Just before rendering all sf::Sprites' location is calculated and updated. For this operation the Camera is needed. The Camera will calculate the sprite's coordinates within the window based on the coordinates of the corresponding Entity in the world. The update of the sf::Sprites is done by the updateSpriteCoord() function of Game. Game holds the World as its member, which in turn holds the Camera and all Entities. This way I am able to access the camera and pass any Entity to the Camera to calculate its position in the window's coordinate system. The Entity's 'view' can then be updated to this position just before drawing to the window.

Furthermore, two classes I specifically want to discuss are Vector2D and TextureLoader. These are 2 standalone classes that provide extra utility to the project.
The Vector2D class is a simple representation of a 2D vector with a X and Y component. Via operator overloading the class supports mathematical operations between instances of this class. It supports addition, subtraction and scalar multiplication. The Vector2D class is used to calculated distances in the 2D World and to move Entities by adding its values to the Entity's position.
The TextureLoader class simply loads sf::Textures from the png files in the Assets folder. During the construction of the TextureLoader object all the textures that will be used during the execution of the Game are loaded into an sf::Texture object. If the texture is successfully loaded it is added to a

map along with a string as a key. Upon load failure an error message is printed to the console. This class follows the Singleton design pattern. This way only a single instance is created and the textures are only loaded once during the construction of this one instance. Any other objects can access this instance and request the map with the sf::Textures.

In addition, let's take a look at the already briefly mentioned Assets folder. This folder contains all assets that are used to visually represent the different elements on screen. These are mostly just png files that are loaded in as sf::Textures by the TextureLoader. Most of these are simply outsourced from itch.io . In some cases I have made some modification to the files. There are 2 png images that I created from scratch: GameOverScreen.png and JumpPadBonus.png . The folder also contains 1 ttf file. This is used to load the font for the ScoreView's sf::Text object.

In order to realize the endless world generation I took a rather simple approach. When the game starts the player is initialized along with the background. A number of platforms are also initialized, enough to fill 2 or 3 screens of play area. From that point on, when the World updates, it will check whether the lowest platform is out of the viewable screen. If this is the case it is removed from the worlds vector of platforms and a new Platform is generated just above the highest platforms in the world (this is well beyond the current viewable screen). If the lowest platform can move vertical, it is only removed when its entire movement range moves below the viewable screen. This way the platform is only removed when it can no longer move back into the Camera view.
When a new platform is generated it has a minimum and maximum height between which it can randomly choose a height to spawn at. This interval increases with difficulty to an extremum, which is just below the maximum distance the Player can jump (without a Bonus).

Finally, let me point out the working behind the Background. As mentioned before it is initialized at the start of the game. The Background is partitioned in tiles of 1024 px by 1024 px. Upon initialization 2 tiles are placed above one another. When 1 tile moves outside of the Camera view it is moved above the other tile.
To create a parallax effect, I added extra layers to the background. Each layer has its own 2 Background tiles. Every layer apart from the backmost layer also moves in the opposite direction to the Camera, whenever the Camera moves. The closer the layer is to the Camera, the faster it will move. This creates the illusion that certain layers of the background are closer to the Camera than others (Parallax effect).

To conclude I believe this project is fairly complete when it comes to the given assignment. All required mechanics are present and function as intended. Of course there is still room for improvement. I could add sound effects and a soundtrack. It would also be useful to implement a main menu and some dialog during gameplay to introduce the controls. Maybe add some extra enemies or mechanics the player should avoid. The current code can also certainly be optimized further. However these improvements are fairly extensive and were skipped for now due to the available time.