Gem Boy by Niels Van den Broeck

The ‘Meat boy’ based game I made for the course Advanced Programming is called “Gem Boy”. In this game, you are a cave explorer who is seeking yellow Tourmaline, some of the rarest and finest gemstones. The explorer has a magical torch which can make him jump faster, further and higher than any other human.

The making of this game had some ups and downs. The process included multiple changes in making choices and plans. Firstly, I made sure to do some research about the design patterns and I planned out as much as possible such that I should not change a big part of my implementation while adding something new. I knew that the separation between logic and representation was going to be an important part of the assignment, so I created two separate directories including all the required classes. I started with implementing the representation part first, because it will be easier to have some visual feedback when implementing the logic part later.

Here is where the first design pattern appears: **the state pattern**. The game object creates the game loop and updates every frame. With each frame, the game gets the current state that is stored by the state manager and asks to update it. There are 2 different states: the menu state and the level state. This design pattern makes it extremely easy to add new things in the future like a level creator without having to worry about transitions between different states and determining in which state you are located. The menu state lets you choose a level from all the available levels. You can scroll through them by pressing the arrows or just by scrolling with your mouse. If you pick a level, a level state object is created. Whenever that happens, the corresponding level gets loaded by the level handler. The level handler also throws exceptions when something goes wrong, like for example when there are no levels found, or when there are multiple players in 1 level configuration file. The state then handles the thrown exceptions to make sure the game keeps running. The level handler reads the configuration file and creates a world with a player, a goal and walls.

These entities are created by using **the abstract factory pattern**. This way, entities can be generated without having to know any information about them. Because the logic part cannot contain any code related to the representation part, I created a subclass called concrete factory that creates the entities from each side. The logic part which, for example, needs to detect and handle collisions and the representation part that for example holds a texture.

This is part of **the model-view-controller pattern**, where the logic entity stands for the model that holds some data and executes calculations. On the other hand, the representation entity is the view that is responsible for showing the right texture on the

screen. The world is seen as the controller that manages user input and the cooperation between entities.

When these objects get created, they also need to get linked to each other. Otherwise, the representation part would not know what the position of its logic part is. Here is where the **observer pattern** comes in play. The representation entity is a subclass of the observer, which holds a logic entity. When the position or animation of the logic's entity changes, it updates its observers, who check their position and animation to display on the screen. This makes it more efficient because the observers do not get updated when not needed to.

Now when the world and its corresponding entities are created, the fun can start. The player can move around and jump in the environment of the level through keyboard input. Press A/left to go left, D/right to go right and space to jump. These inputs create an increment of their local variable horizontalSpeed or veritcalSpeed. Whenever a new frame starts, the player's position gets added by the horizontal and vertical speed. And over time, traction and gravity will be applied on the horizontal and vertical respectively to make its movement more realistic.

This idea is great, but this would not be fair for users on a slower device. The number of times the player's position gets updated depends on the framerate. A lower framerate results in less updates, which makes the player move slower. While a higher framerate results in more updates, which makes the player move way faster. To avoid this problem, I implemented the stopwatch, that checks the time between each frame and multiplies every needed value by that delta time. For example, the player's position gets increased by the vertical speed multiplied by the delta time of the stopwatch. This way the player with a slow framerate will have less updates, but a higher vertical speed added, while the player with a fast framerate has more updates, but less vertical speed added.

This is not the only place where the stopwatch is needed. The camera must also use this technique to make the level rise when needed. That is why I implemented the stopwatch as **a singleton**. By doing this, every class can get access to the stopwatch without having to pass it in any way. Speaking of the camera, it converts the position of logic entities to a position on screen. When the player reaches 80% of the camera's view, the camera gets up a little. This way the calculated position on screen will go down respectively.

While the camera rises, the player should rise too. Otherwise, the player gets below the camera's view. This will lead to a game over and the level will restart. Finally, when the player reaches the yellow gemstone, the level is finished and the next one gets loaded in. When the last level is finished, the game ends and the menu will appear for further playing. Have fun and go find those gemstones!