SKARTA, Semester Project Report

Object Oriented Programming

Sarosh, CMS 305963 CS 9A, Ahsan Tahir, CMS 321888 CS 9A, Tayyab Ahmed, CMS 294827 CS 9A

***Abstract*— SKARTA is a 2D side-scrolling level-based platformer where the player plays as a combination of the three characters Ahsan, Sarosh and Tayyab. The player can switch between these characters at will and must use their unique abilities to complete all the levels and defeat all the bosses.**

I. Introduction

Initially, our aim was to create a Platforming game around our class of BSCS 9-A. It was to have a Main Menu that also included a level selection screen. Upon choosing a level, the player would then choose someone from our class to play as to complete the level. The level would be composed of different puzzle sections that the player will have to solve with a Boss Battle at the end of the level.

This, however, was impossible given the strength of our class, and so we narrowed the characters down to the three people working on the project. We soon realized that the game could be made even more interesting if we were to merge the three characters and allow the player to switch between them. With this, framework in mind, we started researching on how to create games in Java.

We mainly followed the first few episodes of the series Java Programming: Let's Build a Game series on YouTube to get a basic idea of how to open a window, how to draw objects on the window, how to take input from keyboard and how to use threads. In our research, we also came across some amazing games such as Hollow Knight and Limbo**,** which ended up serving as an inspiration for the art style, the game mechanics and the GUI for SKARTA.

After learning the basics however, we felt confident that we had learned enough from the series and our university lectures to build the rest of the game on our own. Moreover, we felt that it was necessary for us to try to do everything by ourselves to properly understand the depth and complexity of all the challenges that can occur while working on such a project. And since we were trying to learn from this experience as much as possible, we chose to potentially compromise the efficiency of the game in exchange for a richer learning experience. Fortunately, the risk paid off and we feel extremely proud of what we've managed to accomplish.

II. Method

The method goes over the flow of the entire program in different sections. The first section (section A) explains how everything is initialized when the game is first launch. Section B goes over how the game runs using multi-threading and multiple game loops. Section C elaborates on the functionality of the **Character** class and its sub-class, the **Player** class along with a basic idea of how collision detection works. Section D focuses on what happens when the game is either exited or the player has won and finally, section E discusses all the different Object Oriented Programming concepts we used in the making of the game.

A. How the Game Starts

At its core, the program exists in multiple states. The state of the game refers to the condition the game is currently in. A possible state could be that the main menu is running, or the game is being played or the game is paused and so on. These states are represented by integer values and they help the game determine and run the code that is necessary for the state the game is in and to avoid doing any unnecessary calculations.

The current state of the game is stored in the static variable ***gameState*** which is available in the **Key** class. Other than ***gameState***, the **Key** class contains all the standardized values such as the dimensions of a single platform and the layout of a single level, the coordinates at which the edges of the screen are located, the time delay before the screen is updated, the gravity constant that determines the downward acceleration of objects and many more. This was done to allow us a place where we can adjust and test these values before standardizing them without having to search for them throughout the entire code every time a change was needed to be made.

When the game is initially launched, the "main()" function in the **Game** class creates an object of type **Handler** which, in turn, starts the entire game.

The **Handler** class is the backbone of the entire program. It is, in some way, connected to everything that happens in the game. The constructor creates a **JFrame *window*** where the game is to be run. It also creates the

pauseMenu, mainMenu, musicPlayer, background and ***camera*** (later discussed in detail). It contains a variable ***currentLevel*** that stores the current level being played and the ***enemies*** and ***bosses*** arrays along with a 2D integer array called ***foreground*** that contains the layout of the current level where each integer entry represents either a platform or an enemy/boss. Additionally, it starts the three threads that run the game and it calls for all the images (sprites) of all the different types of platforms and spikes to be loaded in the static arrays contained in the **Platform** class.

The arrays are static as for platforms and spikes, the same sprites are used throughout the game regardless of what level it is, thus they only need to be loaded once and not reset whenever levels are changed.

The ***camera*** object inherits the Java class **Canvas,** to which a Key Listener is added when the object is initialized, which allows the user to interact with the game through their keyboard. At this point, the game has the default state 1 which means the main menu is running. The ***camera*** renders the main menu screen and the object of the class **KeyboardListener** allows the user to select from the multiple options presented in the main menu.

Once either "New Game" or "Continue" are selected by the player, the ***gameState*** is changed to 5 which prompts the Loading Screen to be rendered and the "levelSetup()" method in **Handler** is called which either loads the "save.txt" file or the "level0.txt" file in correspondence with the user's input.

Each level file including the save file has information such as the path for the image of the background for the particular level, the starting coordinates of the player on the screen and of the ***camera*** and ***background*** in relation to the ***foreground*** array along with the entire ***foreground*** array for that level.

After loading the file, everything is reset according to the information in the file and after a designated amount of time, the loading screen stops automatically and ***gameState*** is changed to 0 which means the game is being played. Thus, the user can now start playing the game.

B. How the Game Runs

Using ***gameState***, the program runs three parallel threads, namely the ***logicThread*** which exists in the **Handler** class, the ***graphicsThread*** in the **Camera** class and the ***musicThread*** in the ***MusicPlayer*** class, all of which implement the Runnable interface.

The ***logicThread*** performs some of the most important tasks of the game. It checks if the player is interacting ("colliding") with any other objects on the screen and if some of those objects are colliding with each other. It also checks if the camera needs to be moved in order to follow the player throughout the level and moves it accordingly. It also updates the animation cycles of the player for example from moving to jumping or jumping to landing based on what's happening in the game. In addition, it also checks if the player has died, in which case the level is reset, or has reached the end of the level, in which case, the next level is loaded or the ending credits are played based on the current level.

After **Handler**, the **Camera** class is the second most important class in the program as it controls the entire display. It inherits the **Java.awt** class "Canvas" which allows it to render objects on the screen using a ***bufferStrategy*** and a **Graphics** object ***g***. It's also heavily connected with the ***gameState*** variable through its ***graphicsThread***. The thread essentially runs a single method "render()" in a while loop and then "sleeps" for a certain amount of time calculated in milli-seconds so that the total time taken for each iteration of the loop is constant. This is done so the screen refreshes at a constant rate and to make the animations run smoothly.

Considering the value stored in the ***gameState*** variable, the "render()" method renders objects on the screen such as the Main Menu, the pause menu or the loading screen by calling the method also named "render()" that exists in the object's respective class. However when, the gameplay is being rendered, the method "renderGame()" is called which exists inside the **Camera** class. This method uses the variable ***xcor*** (x coordinate) that keeps track of the position of the camera relative to the foreground array and reads the area of the foreground array that needs to be rendered on the screen. It then calls all relevant "render()" functions of the objects that need to be rendered. It first renders the background and the player, followed by all the objects whose reference it found on the foreground array, followed by the Heads up Display which shows all the player stats such as how much life the player has, the character that the player is switched to and if they can use that character's special ability.

For all the objects that require some form of animation or movement, in addition to their "render()" method, a method called "tick()" is also called which updates the current frame of the animation. The objects that meet this criterion, implement an Interface called **Animated** that only contains these two methods. The interface allowed us to organize and determine which objects would need to be animated quite effectively in the designing process. Thus, with the combination of the

"render()" and "tick()" methods, we were able to add animations to the game.

Although the **MusicPlayer** class adds a whole another dimension to the gameplay experience, it performs a relatively simpler tasks compared to **Camera** and **Handler**. The ***musicThread*** plays background music on loop based on the ***gameState*** and ***bossFight*** variables.

Additionally, the class has a static method "playSound()" which allows any object to play a sound clip at any volume by passing the path of that clip and the volume as arguments. It also takes the position in micro-seconds within the clip from where the object wants the clip to start playing as a parameter. This was especially helpful as we made some of the students in our class record audio for their respective character in the game and the clips had around half a second of pause before the intended sound was made. Since the clips were mostly a second or two in duration, they couldn't be trimmed properly by any online audio trimmers, this functionality of the built-in class **Clip** proved to be a great asset.

The main purpose of separating the game into three threads was so that the workload can be effectively divided. Initially, we only had the ***logicThread*** and the ***graphicsThread***. The idea behind separating the two was that in order to make the animations run at the same speed regardless of the processing speed of the computer running the game, the ***graphicsThread*** needed a constant delay between each iteration of it's while loop. This however, made the thread slow. Thus, calculations regarding collisions and object interactions could not take place in this array because of two reasons. Firstly, the time taken for calculations was longer than what the delay allowed, it would cause the entire game to lag and slow down and secondly, even if the calculations were carried out in time, the program would need to wait till the next iteration of the while loop to do them again. Since multiple things need to be calculated and accounted for irrespective of the movements seen on screen, we decided to move all the logical calculations into a separate thread which could run multiple times in the time it took the ***graphicsThread*** to complete a single iteration. In fact, this strategy was so effective, we had to add a minor delay to the ***logicThread*** as it was running around 25 times faster than the ***graphicsThread,*** which was highly unnecessary.

When sound was later added to the game, we decided to create a third thread that dealt with the background music for two main reasons. Firstly, because the "start()" and "stop()" methods of the different audio clips needed to be called often and calling too many functions in the same thread seemed to cause synchronization errors and glitches and secondly, because it would've crowded the ***logicThread*** significantly and would've made it difficult to program with. Thus, we ended up with a system where the three threads run in tandem to make the game run as smoothly and efficiently as possible.

C. Characters and Collision

As is apparent in the UML diagram below (also submitted separately to make it more readable), the abstract class **Character** is a major class that provides the groundwork for every character in the game. There are eight people with their own class, namely **Abdullah**, **Ahsan, Jahangir, Mutahar, Sarosh, Talha, Tayyab, Usman** all of whom are people from BSCS 9A. The **Character** class provides all its sub classes with a display coordinates, the character's position on the foreground array in terms of row and column, a speed, number of animation cycles, methods essential to movement and collision and finally, a sprite sheet.

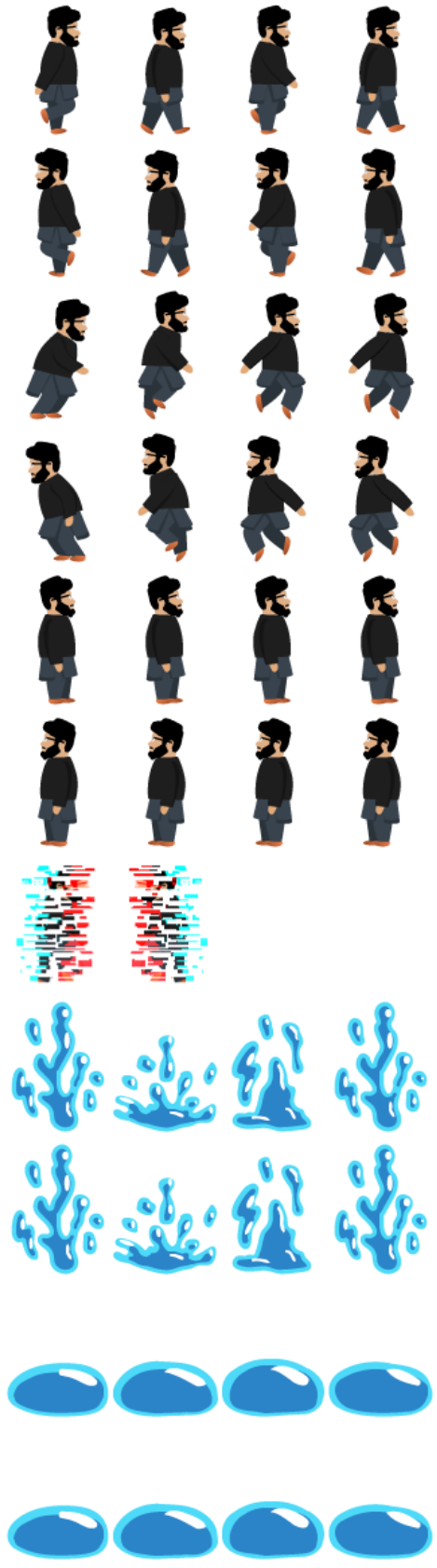

Fig. 1. Sprite Sheet of Ahsan

A sprite sheet is a big image composed of smaller images (sprites) arranged in rows and columns. Since animation requires every moving object to have multiple images (frames) playing back to back, this would've caused us to store and load multiple image files for every character, and then organize them carefully for the animation to work. In order to make the task less taxing for us, we used a relatively popular technique called sprite sheets. We designed our own format where the different animation cycles such as walking or jumping were separated by rows and the different frames of the cycles arranged in columns. In the sheet, each cycle could have a maximum of only four frames as we only had one graphic designer, Tayyab Ahmed and it is a time intensive task. The cycles were also separated by the direction the character was facing and were ordered as the right facing cycles were on even numbered rows and the left facing cycles were on odd numbered rows starting from zeros with some exceptions in the sprite sheets of **Player** characters, one such example is given on the left. Thus, all we had to do was load one image file, crop it and turn it into a 2D array of images for each character using the static method "makeSpriteSheet()" in the class **HelperFunctions.**

Although the **Character** class acts as the main super class for all characters in the game, the **Player** class is the biggest and by far the most complex class in the entire program. The **Player** class is an abstract class that has all static variables. This is done for two main reasons. First and foremost, although the three characters ***Ahsan, Sarosh and Tayyab*** are three separate objects, they are still extensions of the same **Player**, which means they all share the same variables.

For example, if the player is standing at one end of the screen as ***Sarosh,*** then switches to ***Tayyab*** and travels to another end of the screen and switches back to ***Sarosh.*** In this case, if ***Sarosh*** and ***Tayyab*** are to be treated as two separate **Player** objects with their own separate display coordinates, then upon switching, the player will suddenly teleport back to the other edge of the screen where they were last in the form of ***Sarosh.*** Thus, to rectify this, we constantly always need to update both their display coordinates so that they're at the same location. Moreover, we also need to constantly update Ahsan's coordinates so that all three characters exist in the same location.

This gets more confusing when variables such as hitbox coordinates and ***life*** and Boolean variables such as ***isColliding*** are taken into consideration. Essentially, there is only one player on the screen at any given time. The player can, however, take three forms. But at any given time, the player

only exists in one of those forms, so at any given time, only the Player and its current form need to be considered.

Since this was the opposite of polymorphism, the only way we knew how to achieve this behavior was by making all the variables in the **Player** class static, so all different forms of the player can share the same fundamental variables, as those remain the same, irrespective of the current form.

As for collision detection, it is the idea that we need to check if two objects are interacting ("colliding") and to determine what actions should be performed when a collision occurs. In the game, the "collide()" method of individual objects (Characters or projectiles) is called in the "collision()" function in the **Handler** through the ***logicThread.***

For the Player's collision detection, the 20 x 60 foreground array is used. The collision detection is called after the player moves. Depending on the direction of the Player's movement, specific sub-functions of the collision detection are called. For example, if the Player has moved upwards, only the upward collision detection is called. In the upward collision detection algorithm, it is then checked whether the Upper Limit of the Character's hit Box(A dynamic box that covers the portion of an object that can be collided with - usually a little smaller than the visible part) has a Platform intersecting with it (courtesy of the foreground Array). If there is a platform there, then the Upper Limit of the Character's hit Box is moved to the Bottom Limit of that Platform, the vertical speed is reset, and if the platform type is spikes, the player also takes damage.

The Player's collision with bosses, or projectiles is much simpler as we don't have to worry about which specific parts of the Objects are colliding, nor what the previous directions were. There are arrays for the path that the two objects have taken compared to the last frame. For each element of the arrays we check whether the two objects collide, and if they do we called the (getting) hit function of one of the objects depending on their states.

Other than this we had to work on a regular Enemy's collision with Platforms. For this we made sure that the Enemy walked around on their Assigned Platform. We simply inverted their direction when they collided with another platform horizontally, or the bottom limit of their hit box was not the upper limit of a platform.

D. How the Game Ends

There are two ways the game can end. First, if the "exit" option is selected in the pause menu, the static method "saveGame()" is called from the **HelperFunctions** class which saves the current progress in the "save.txt" file and the main menu is opened. From the main menu, the "exit" option exits the entire game. If Alt+F4 is pressed during the game, the game also exits but without saving any progress.

If you beat the game, the credits roll up, stay for a certain amount of time, and you are taken back to the main menu, where you can choose to start a new game or replay from your last save point before you beat the game in continue.

E. Object Oriented Programming (OOP) Concepts Used

Since SKARTA is created in Java, a completely object-oriented level, the game ends up using almost all the OOP concepts that we learned this semester.

Firstly, the **Animated** interface allowed us to organize every object that would require an animation effectively. We've also used the built in **Runnable** Interface of Java in classes that make use of threads.

Secondly, the abstract class **Character** and its multiple branches of sub classes are a perfect example of inheritance used in an effective manner.

Additionally, how the **Boss, Enemy and Player** classes contain most of the collision detection functionality that is used by their sub classes during collision detection highlights the use of polymorphism.

Moreover, the combination of the **JFrame *window,*** the **Canvas** sub class **Camera,** and the **KeyAdapter** sub class **KeyboardListener** create the GUI that is used for user interaction.

All in all, the general use of encapsulation by nearly every class allows all the data fields and methods to be easily organized and bundled together which divides the code into smaller chunks, making it easier to both understand and manipulate.

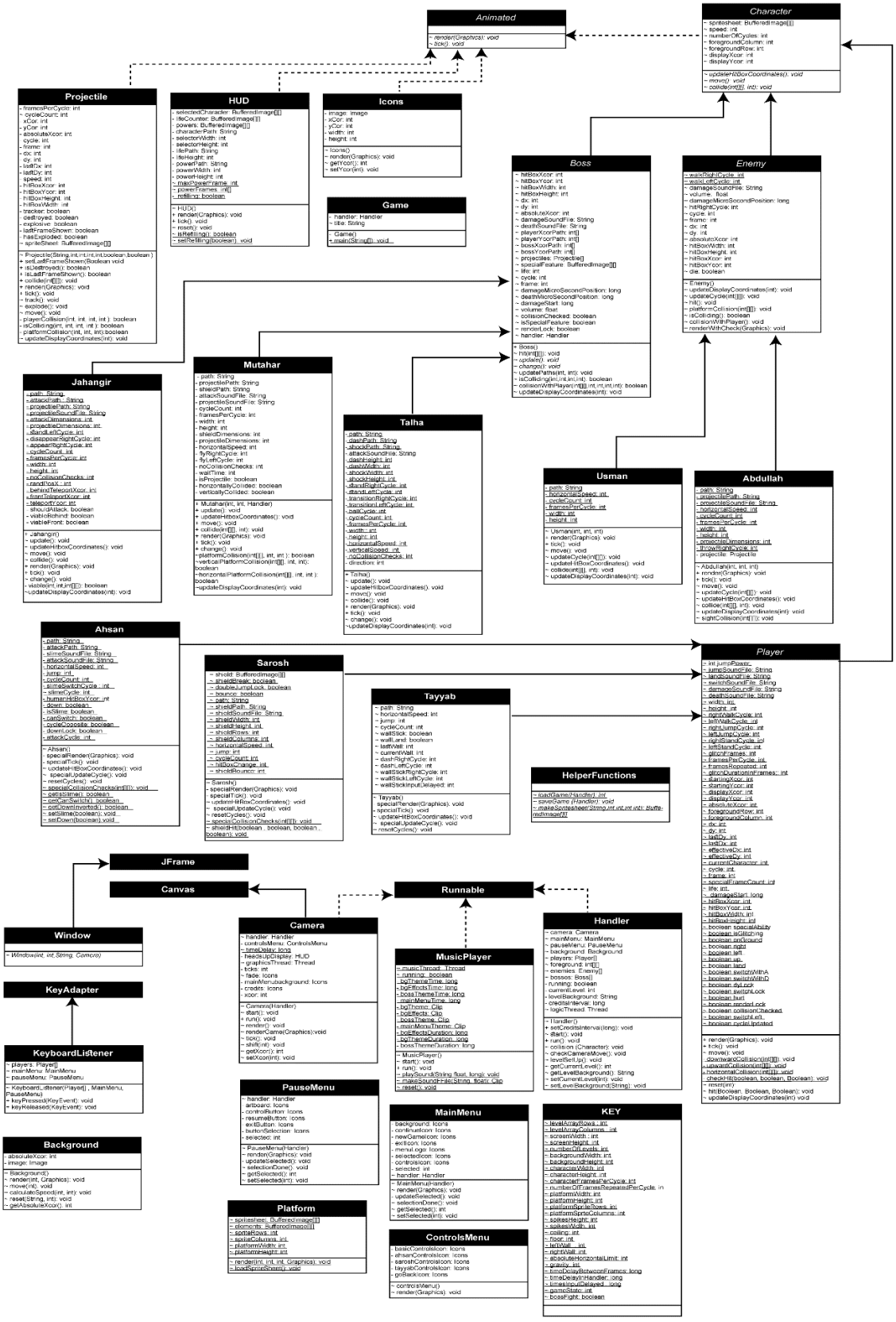

Fig. 2. The class UML diagram

III. Results

During our time on this project, we learned a great number of things.

We realized how time intensive and exhausting graphic designing can be. Every single image displayed in the game was hand made by Tayyab Ahmed along with the design for all the levels. We have witnessed firsthand the amount of painstaking amount of effort he put into each design. Tayyab also edited the demo and created the gigantic UML diagram presented in this report and included with the project submission.

Ahsan and Sarosh worked mostly on the code. Sarosh created the basic skeleton of the game and the implementation of all the mechanics while Ahsan worked on collision detection and polishing and finalizing the code.

Finally, after two months of coding, brainstorming, designing and coding, we managed to execute our vision. The game runs smoothly with nearly no bugs or glitches. After completion, we tested it on an intel gen i5 2013 processor with integrated graphics to asses the limitations of the program, but to our surprise, it ran perfectly fine.

IV. Discussion

Although the game is finished, there are still things we feel we could've done to improve the program's functionality if we had more time.

Firstly, since all three of us working on the project had displays with a resolution of 1920 x 1080, that ended up being the only available size of the screen, meaning the game cannot run properly on a display with a lower resolution. In the future, we might try to add a feature where the game window adjusts its size based on the display it's being viewed on.

Secondly, although we've only used static variables in classes that exist as one single entity in the game and at places where we feel it's necessary to improve the efficiency of our code, we would still like to reduce the number of static variables to as few as we can.

Thirdly, since the project was really time consuming in addition to our other academic duties, we never found the time to properly learn about all the different features GitHub provided that would've undoubtedly helped us work significantly better together especially since we couldn't meet in person and discuss things in person. In the future, we will try to take time before starting the project, to educate ourselves on GitHub and all its functionalities.

Furthermore, our original idea was to have four bosses in total, but due to time constraints we could only make three and had to make the player fight all of them in the last level instead of having a final boss. If possible, we would like to create the final boss in the future.

Finally, since the sound was added near the end of the project too, all the levels and boss fights have the same two background themes. In the future, we would like to add more sounds, with each level and boss fight preferably having its own theme

V. Conclusion

In conclusion, we ended up learning a lot about the process of creating and all the different skills required to make a video game. We ended up respecting each other's talents and all the different skillsets and ideas we brought to the project. We loved every minute of it and if luck permits, we might work together on an even bigger project and try to make it just as entertaining and educational for each one of us, if not more.