Ty Morrell
Ryan Frapwell
John Henry Buenaventura
David Hernandez

**Campus Run Final Report**

For our Software Engineering project, we have decided to initiate a game that is somewhat a combination of the games Temple Run and Super Mario resulting to a 2D version of the Temple Run.
Campus Run is written all in C++ and consists of multiple PPM images that are used for rendering the background, the ground floor, and the character actions. The character actions are made up of three PPM images and each image is made of at least five frames of images. For example, the default image is made up of different pictures per frame and they are cycled through in its sprite sheet in order to implement a running animation. Throughout the creation of the Campus Run, we have gone through a few phases in order to get to where we are now.

As the first stage of our project, we have divided a few ideas throughout the group. At the beginning Ty wanted to work on the Running Image, John wanted to work on the Sliding image, Ryan wanted to work on the Jumping Image, and David wanted to work on organizing the code within the appropriate .cpp files.

During the first phase, outside the planned ideas of the group, everyone got together during class in order to get the ground rendered and test a moving object to go across the screen. That was the kick start of the Campus Run Game.

The green bar from the image above was the first graphics that the group was able to render at the beginning of the project. Ty was able to derive it from the Waterfall homework for its collision detection algorithm.

As the group members had some time off apart from each other, John was able to work on a temporary algorithm that took the concept of the Temple Run game and able to render a very “cheap” version of the collision detection game. However, the code he wrote was written to render an overhead 2D version of the game where the character moves left and right in order to dodge the oncoming obstacles. Also, another problem that John had was that his .cpp file was coded to run on a windows OS system only. His code contained conio.h and graphics.h that is
not found in any library within a Linux terminal. He tried coding both files from scratch in order to be able to run from a terminal, but the graphics.h was not successful.

```c
for(i=3; i<49; i=i+13)
    { A[i][1]='*';
    A[i][5]='*';
    A[i][8]='*';
}
```

His design consisted of a character array in order to store one character and that is the “*” symbol. John used the asterisk symbol to create the path of the game and when a character is in the same location as the controllable character, then it is game over. The concept didn’t work successfully due to the fact that John’s code is for an overhead perspective.

During the group time off, Ty was able to render the background into the window. He created a sprite sheet that is focused mainly for the background that continuously scrolls through on a loop. It became the first rendered PPM image that was inputted onto the game. His algorithm became the source of many other rendered images in the future such as the sprite sheets for the character actions.
As it can be seen from the image above, Ty was also able to figure out how to render another image with its background being rendered transparent. That also became the source of many other rendered image in the future so that everything ran smoothly against one another.

After the rendering of the background image and rendering another image on top, the idea of cycling through multiple image on one sprite sheet became another idea from Ty that later on led to the first running image of the character.

Derived from this idea came Ryan’s jump function. Ryan’s jump function consisted of multiple images in one PPM image that is cycled through while the action is executed. He was also able to render the image where the sprite sheet position is changed as the action is executed and show as if the character is jumping. The same concept was fed into the sliding function that was worked on by John.
John worked on getting the slide function working where when the user presses on the down key, the runner image is disable and the sliding image is enabled for a small duration. He didn’t have to be concerned about the change of position of the sprite sheet because the character was meant to stay on the ground, but he was able to manipulate it where is seems as if the character slides forward quickly for a second just before the Running image is enabled again.

Now as the fifth phase of the project, Ty and Ryan was able to generate and add a few additional images that represents boosts, obstacles, etc. With the Boost image, Ty was able to add the collision detection as if the power up was consumed by the character.

Being able to generate this image into the game fed the idea towards Ty’s accomplishment of rendering other images such as a spear. Multiple sprite sheets are being generated throughout the process of making the game.

Also in this phase, Ryan changed the background from a single image to multiple images that were layered on top of each other, that were all moving at differing speeds. He took a free image from the Internet and split it into 4 parts to create the parallax scrolling that is now in the game. He had some issues with getting parts of the images to be transparent at first, but he eventually figured out what was wrong and addressed it.
The next phase of the game was to include the audio effects for the runner and every object in the game. David spearheaded this task using the OpenAL development kit, which included the ALUT library. OpenAL is a cross-platform audio application programming interface (API) that is designed for efficient rendering of multi channel 3D positional audio. This environmental 3D audio library aims to provide realism to a game by simulating attenuation, such as the degradation of sound over distances, and the various densities of objects and materials within the game.
The general functionality of OpenAL involves source objects, audio buffers, and a single listener. A source object contains a pointer to a sound buffer and the velocity, position, direction, and intensity of that sound. While the listener object contains the position and direction of sounds for the user, it allows the sounds created by the sound buffer and source object to project towards the user by setting up the connection between the program and computer, thus providing a dynamic environment for the video game. The ALUT library allowed us the ability to use the functions for OpenAL and provide easier transitional functionality into our game. Hence, both libraries allowed us to establish a connection to the audio device the computer used, modify the position, velocity, gain, and pitch of each sound source, and play any source we want the game to emit when any instance occurred within the scope of the game.

![Image of a game interface](image)

The initial idea with the sounds of the game was to create a soundboard that would load the sound files into an array of sounds and play a specific sound when an event occurred in the game. Every sound source would have specific attributes within a class in order to play sounds while encapsulating the data that the source code would manipulate. However, one of the limitations of OpenAL and the ALUT libraries is that they do not take into account sound delays or queues without either executing an alSourcePlay() at a specific instance in time or adding in that functionality separately. Because of this issue, the sounds would continuously play with
every render frame until the source code would manually stop the play function. This either resulted in sources playing in an infinite loop or not playing at all.

```c
    case 4:
        x = obstacleEffect(monsterMovement, x, y, z, monsterTexture, 
                            dead, image_counter, obstacle, sprite x, 
                            sprite y, booster, xdiff, monstersheetx);
        //Need sound for monster here
        if (x == 1203) {
            play_monster();
        }
        if (x == 501) {
            play_monster();
        }
        break;
```

One method that solved this issue was executing a sound to play at a specific render frame when an object entered the screen. By setting a condition of when the object entered the screen, we were able to play the sounds only once and improve the cohesiveness between the objects and the sounds of the game.

```c
    //this makes sure the player can't double jump
    if ((jump || stuff) &amp; &amp; !dead) {
        showRunner = 0;
        jump = Jumping(jumpsheetx, (wid), jump, point_y, jumpTexture, 
                        stuff, xdiff);
        jumpcount++;
        play_jumpsound();
        if (jumpcount == 4) {
            jumpsheetx += .1;
            jumpcount = 0;
        }
    } else {
        showRunner = 1;
        sprite_y = standardy*xdiff;
        jumpcount = 0;
        jumpsheetx = 0;
        play = !play;
    }
```
Boolean logic also played a crucial role in setting up sounds as well, especially for the runner. By establishing conditions when a command was given to the runner to either slide or jump, we were able to play the sounds relative to those actions without triggering any other unnecessary sound. This helped immensely while we were going through the debugging process for the sounds because it allowed us to see which sounds played at any specific time or when sounds would either be stuck in those infinite loops or not play at all. With these methods, we were able to establish the array of sounds and play them whenever any instance occurred without interfering or manipulating the core functionality of the game.

The final phase of the game was to implement the effects that each obstacle or boost would have on the runner. This was done by implementing a switch statement along with some other ancillary helper functions that would ultimately affect the runner, score, and screen animation. There were nine obstacles or boosts that the runner could encounter at the time of game completion. Currently the obstacles are a low spear, high spear, faster low spear, faster high spear, speed boost, money bag, life increment, alien ship, alien monster. All of these obstacles and boosts have their own speed variable that is passed to the effects function where based on the number of obstacles passed the boosts and obstacles begin to move faster across the game screen. Some of the effects from colliding with one of the obstacles or boosts are runner death, runner speed increase for a temporary period, score increases, incrementing the number of
lives the runner has, and all of the effects are tied to a specific audio file that is played once they are spawned.

The helper functions that were referenced above were primarily used to ensure that the obstacle being spawned was random and that the effects of the obstacles did not repeat too often. The use of rand() to implement random obstacles does sometimes create the same obstacle a little more frequently than can be desired but the ultimate effect is still functional. Some other helper functions that were created for the game are a score modifier that increases the score increment the further the runner makes it through the game. This modifier is based solely off of the score which is based on the distance the runner has traveled.

The final function that was worked on was to create the ability to save the scores in a database where they can be viewed from a website so that users can track their high score. Unfortunately, there is currently no method for inputting the users name to go along with the score, but this aspect is still being worked on. This function works by communicating with the server where a function that creates a c++ instance of the PHP get method is used to store data in a mysql database. On the web server there are PHP files that communicate with the c++ document to save the data in the database. Currently the list is not sorted by high score but is instead listed chronologically from oldest to newest. This function is called by the main file when the user hits escape or the maximum numbers of lives has been reached.

This game was trying at times and the group aspect of implementation made it all the more difficult. However, our group was able to learn on the fly and implement a functional working game despite our different coding styles and methods for implementation. The amount of work put into this project is not purely represented by the number of lines of code but also by the fact that we were able to set aside our different ideologies and implement a project that we
are all proud of. Each group member contributed code to the main file as well as to their personal files that is of the highest quality based on each coder's abilities. We all look forward to continuing to contribute to this game on our own in the future as well as using this as a project to show to prospective employers.