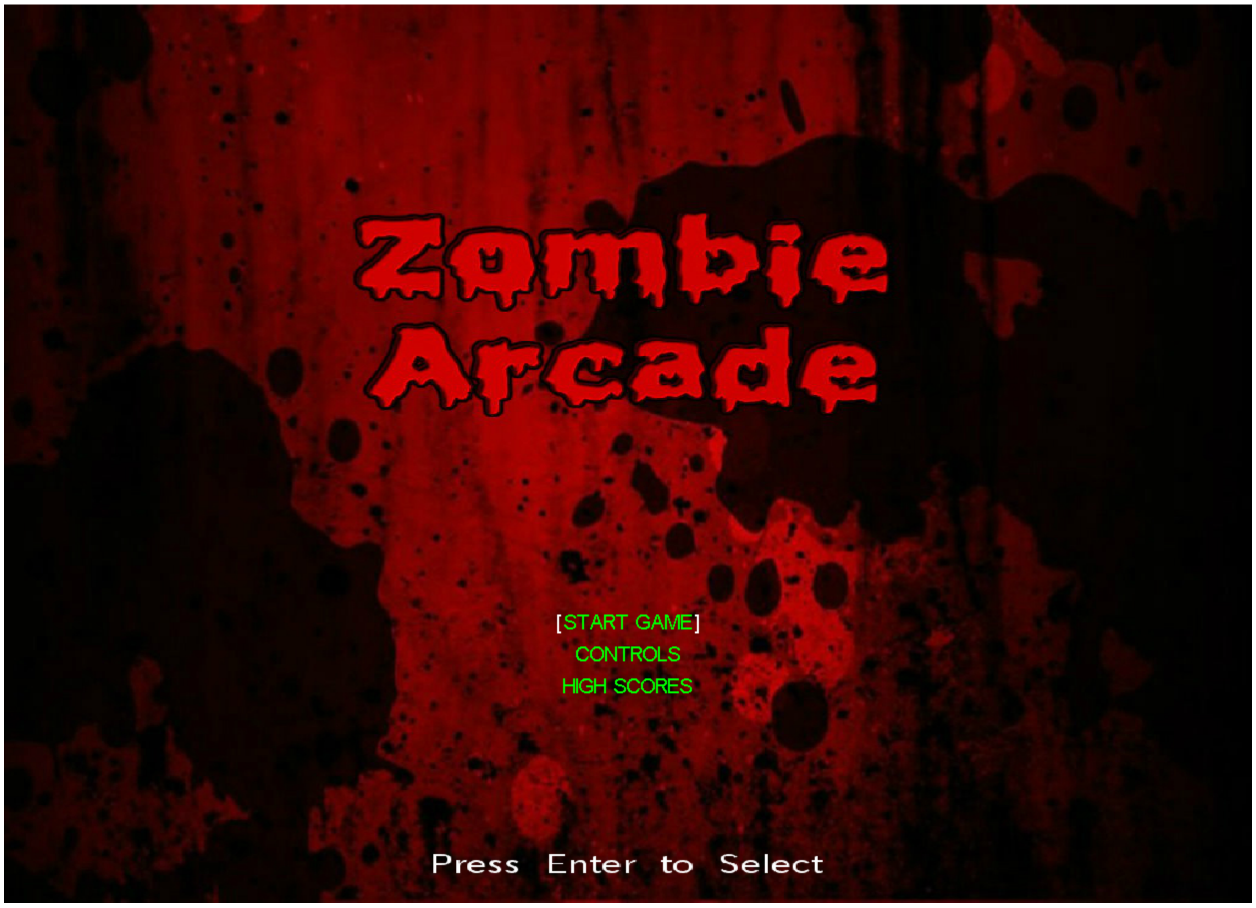
Zombie Arcade

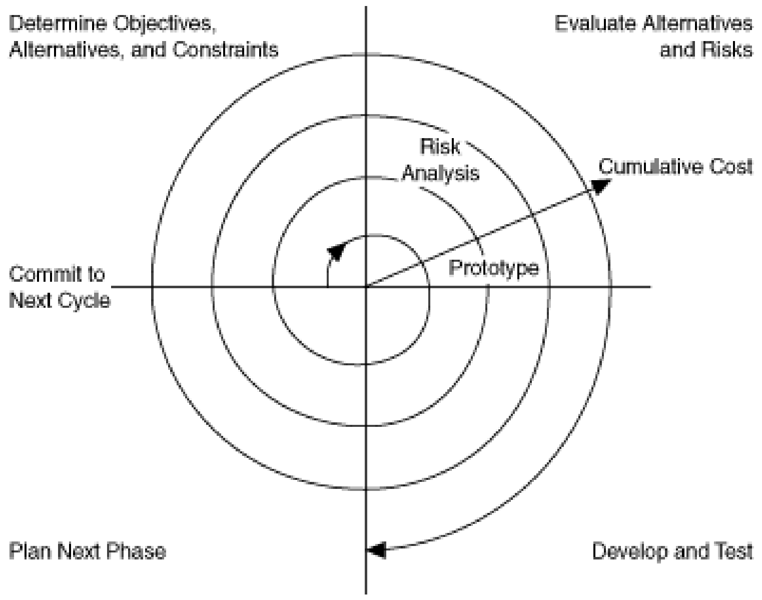
Team 7 Technical Report



Jonathan Halbrook
Alvaro Juban Jr.
Brandon Ware

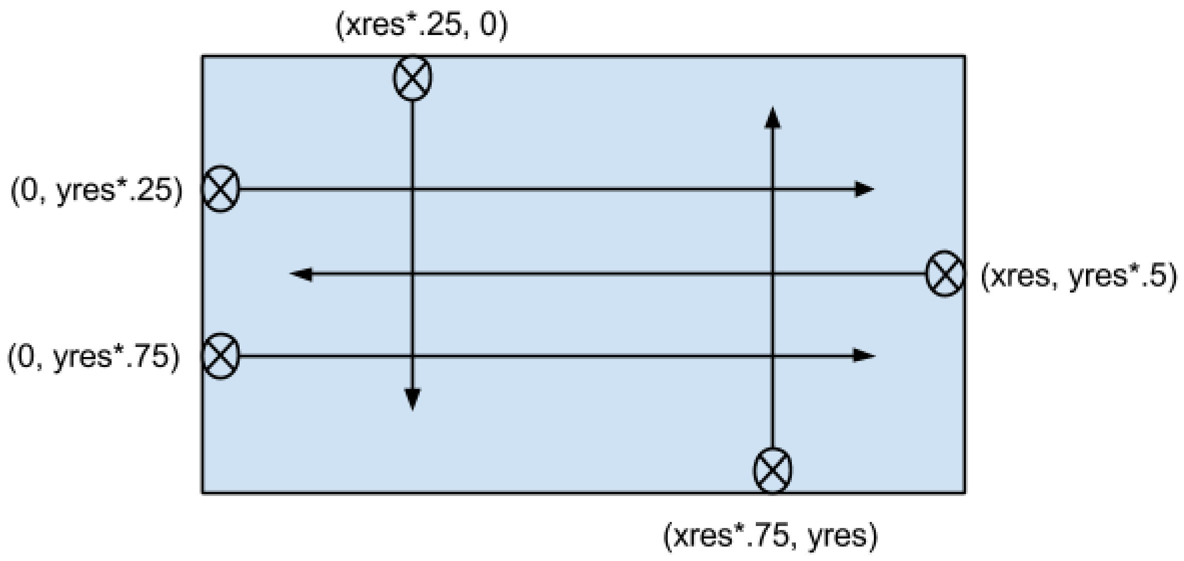
Lifecycle

For the lifecycle of the project we decided it was best to go with the spiral model. We started out deciding that what would be best would be to create a basic working game with all the necessities of what a game is, a hero, an enemy or AI, an objective, functionality of the player and enemy/AI, and a score. And then once we had a working a game we would go back through the same process and implement new ideas or extras that would make the game better and more enjoyable to the player as long as time permitted. This allowed us to be certain that once the deadline was reached we would have at the very least a working product.



<u>Determining how and where to spawn the zombies</u>

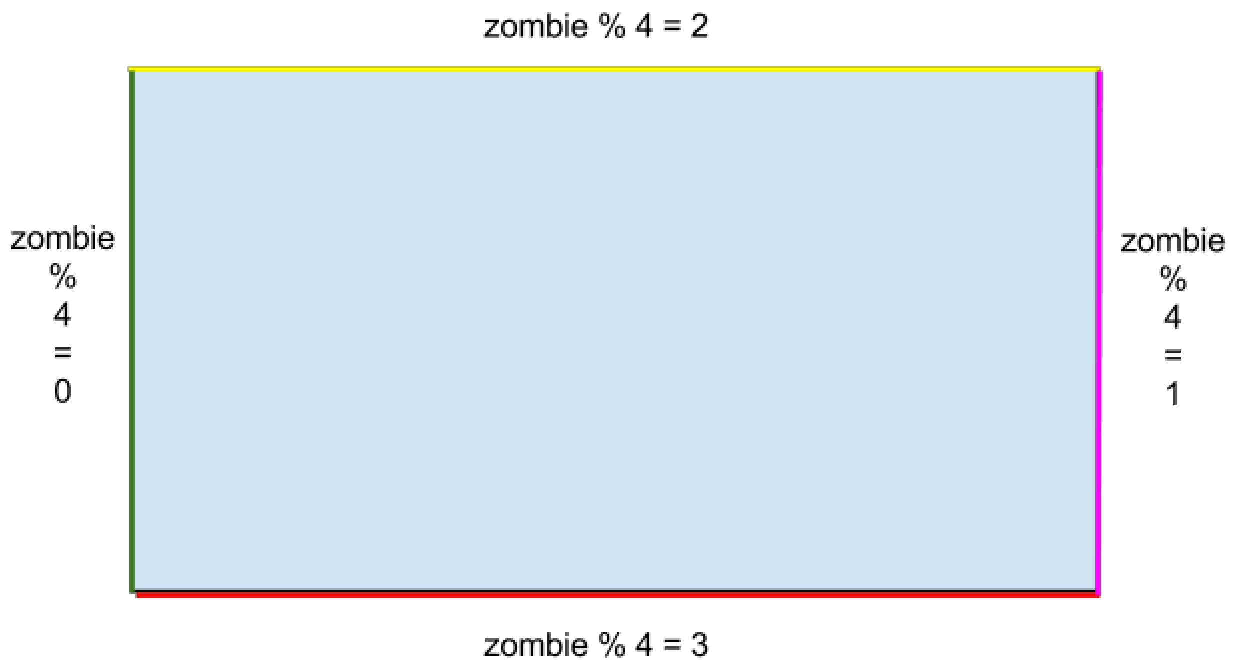
At first we took the functionality of the asteroids program to determine how to spawn the zombies. Once we understood the code and functionality of code we implemented into our own program. We started with just five spawn points alongside the windows edge.



Also causing them to go at a constant speed straight ahead. Once the functionality was finished the challenge was to make the Zombie go straight towards the player.

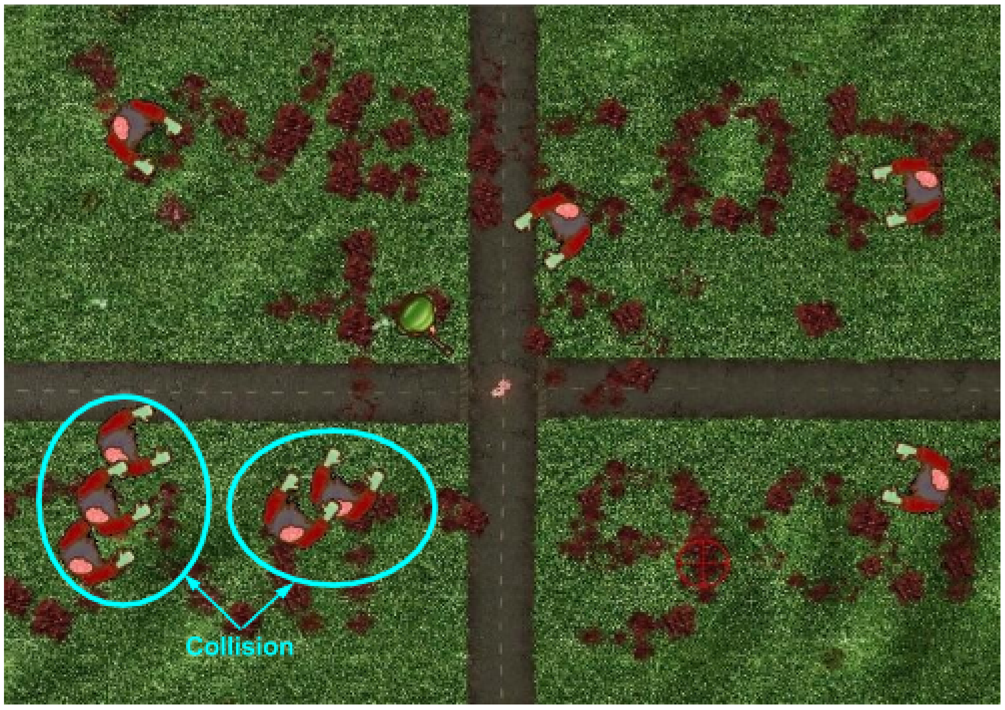
In order for the Zombie to go straight towards the player the hypotenuse and the distance between the Zombie and the Player had to be found. Once found it was only a matter of checking to see once the player moved at what angle he moved and in what direction. This rotates the

Zombie and we made the Zombie maintain a constant speed that was just slower than the player. Once the Zombie was appropriately moving towards the player we decided it would be best and a lot more challenging if the zombie spawn points were randomized along the left, top, right, and bottom of the screen. In order to accomplish this we made the zombie spawn mod four since the screen has four sides. So all the zombies that once modded by four equal zero would go on the left side of the screen, the ones that equaled one would go on the right side, two would go on the top, and three on the bottom.



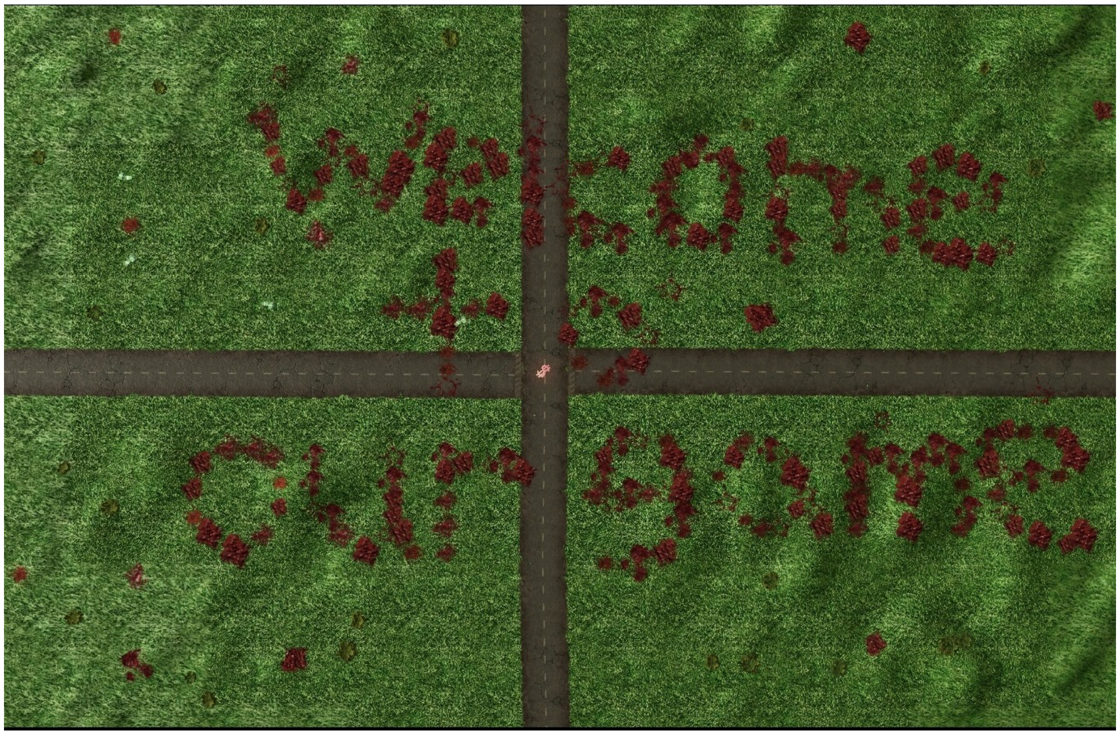
Zombie Collision

In the structure we give the Zombies a radius. So for collision to work all that was needed to be done was to check the distance of the other Zombies. Once the distance was less than the radius we had the option of either giving it some sort of penalty or to push the other Zombie it was colliding with forward or sideways. We decided that it would be better if it made it seem like the Zombies were actually pushing each other once they collided to get to the Player.



Backgrounds

For the backgrounds we made all decided that we would only use five different backgrounds for the player to kill zombies on. The first background being the welcoming of the player to threshold of our game.

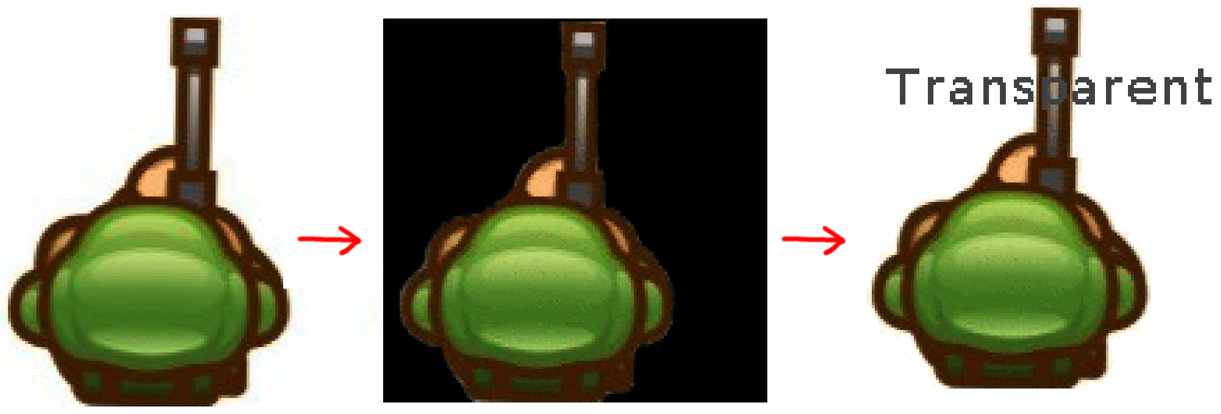


All the backgrounds were made on the Starcraft II map editor, screenshotted, converted to a ppm image, then inserted into the game. Once you go past zone five or the fifth background it will return you to the second background using a modulus operator. After each zone not only does the background change but the zombie count is also multiplied by two.

Textures

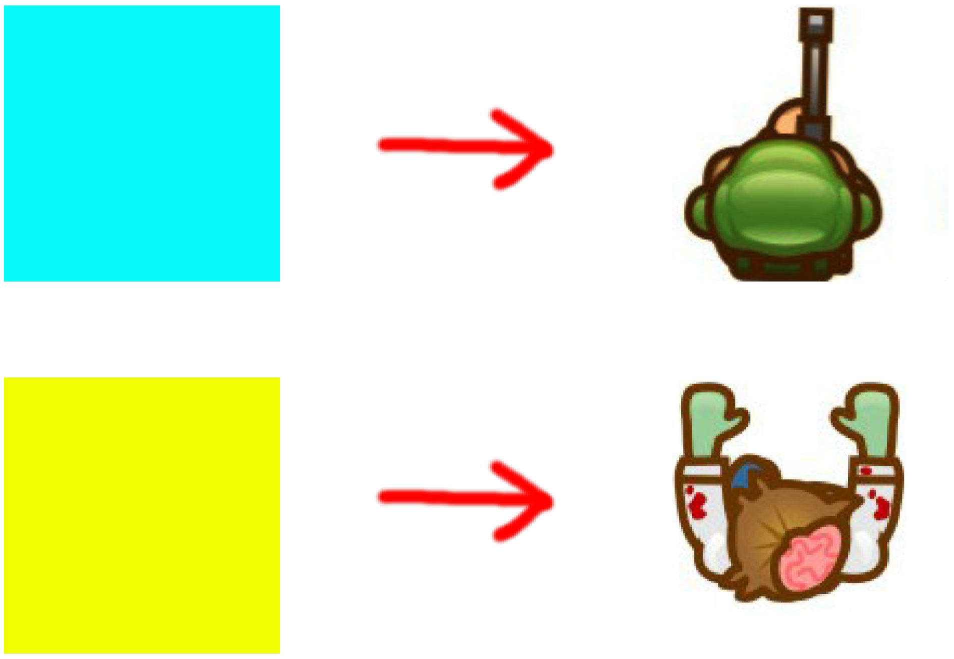
As a group we decided that we wanted to create a zombie game using the framework of an asteroid game. Based off the asteroid model we came to the conclusion that using “top-down” textures would be the most natural and easier. Normally most games use sprite-sheets that include a large image with multiple perspectives of said sprite. To create the illusion of animation this method requires coordinates of a certain sprite and timing for when you would like to change. From a top-down perspective we can simply rotate the image to create the animation of the object changing its view.

Though a bunch of square pictures moving around doesn't look as appealing as actual textures. In order to make parts of an image we didn't want to be seen disappear we had to convert the image. This was done by coloring undesired parts of our image black. We then use OpenGL functions to create a silhouette of the original image. This method takes the color black in our image and doesn't display it to the screen. Once this is done the undesired parts of our image appear to be transparent.



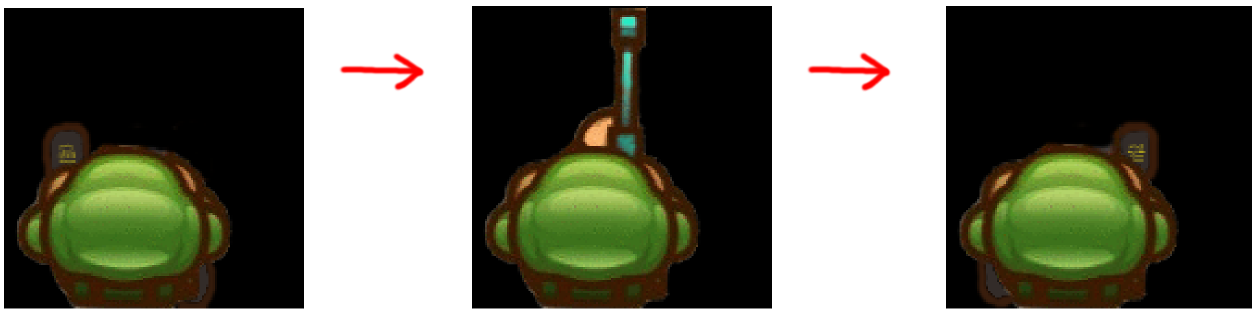
Texture Binding

Once we decided what type of textures we wanted to use we had to figure out how to attach these textures to both the player object and zombie objects. For this we used OpenGL, which was very convenient because not only did it offer the function of binding textures it also had the function for rotating the image. Without going into to much detail binding a texture to an object is pretty simple. For example with our player object (soldier) we first find where the player is located by using glTranslate() with the position of the object . Once found we simply use glBindTexture() and call forth the image to be displayed onto the screen. We then resize the image to fit into the designated object so that correct collision properties still apply to our texture.



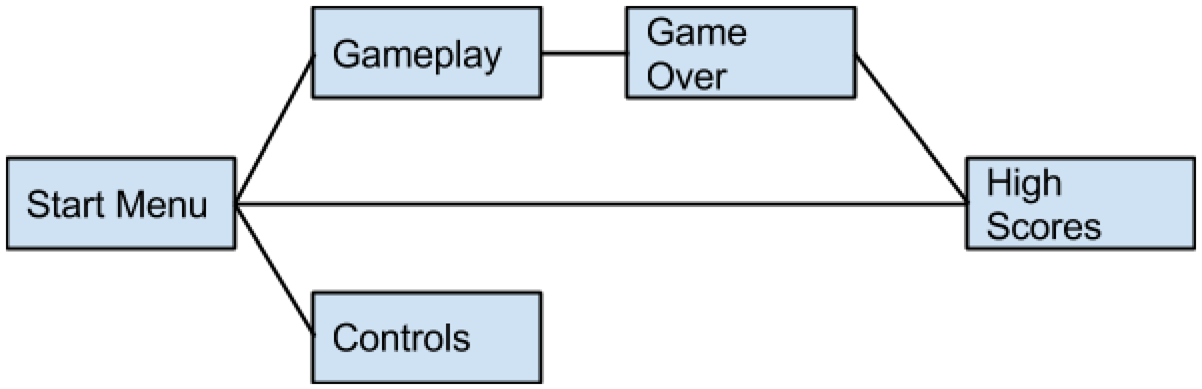
Animation

Now that the textures were binded to the textures we wanted to add little more detail to our player. This little detail was the implementation of animation. We aimed to make our soldier texture appear to take steps on screen. As stated early we used the glBindTexture() but instead of attaching the same image automatically to the player object, we placed conditions to decide which image would be displayed. When the player isn't moving the standard soldier is attached, but once the player begins to move a walking animation initialized. This is done by seeing if the user is pressing any of the designated keys that moves the player. In this case it is “w, a, s, d”. Simply put whenever any of these keys are pressed 3 images begin to loop and continue to loop just so as long the player keeps moving.



Screen Control / Program Flow

In order to implement several of the requirements for our game, it was necessary to create a menu screen. The start menu acts as a gateway to all of the other parts of our program; a starting point for the user.



This was done by creating a set of ‘states’, or variables, set to 0 or 1 depending on which screen was currently active. Each state corresponds to its own function, and its own loop nested within the main program loop. These variables work similarly to how semaphores function, except in regards to program flow in place of user access. These variables, for the most part, are mutually exclusive; two screens can not, and should not be displayed at the same time for obvious reasons.

Loot

A loot system was implemented by creating this simple structure:

```
struct Loot {
        Vec pos;                        // Set this to the position of the ‘dead’ zombie

        Ppmimage *lootbg;

        GLuint lootTex;                 // Each loot object has it’s own texture
        int type;                       // type determines what the powerup() will be
        struct Loot *next;              // List stuff...
        struct Loot *prev;              // ...
        struct timespec lootTimer;      // Each loot object has it’s own timer
        Loot() {
                //constructor stuff...
        }
};
```

Each loot object houses an image to be bound and displayed, a type which corresponds to a specific powerup, and a timer which lets us know when to delete the loot object.

After every zombie kill, a value is randomized and given a 5% chance to succeed. If the randomized value succeeds, we roll again to determine the type of the loot. After the type is determined, we call the powerup() function to accomplish our goal. Each loot object is deleted when either picked up, or left on the ground for more than 5 seconds, as determined by the timer check within render().

Sound

Contrary to what several other groups experienced, there were very few problems with the addition of sound to our game. We used the FMOD library provided in some of the other frameworks used throughout the course.

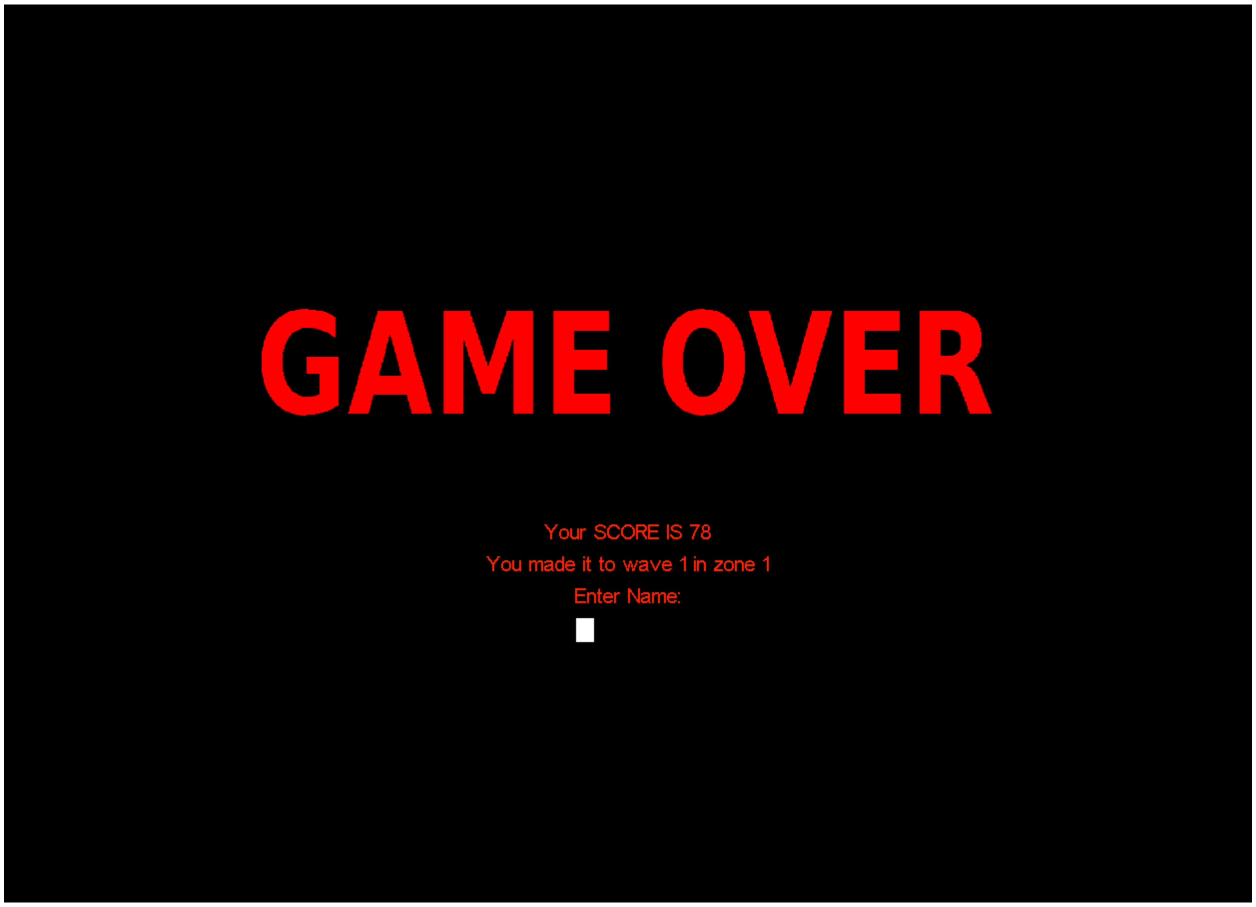
In addition to the standard functionality, we also implemented a way of accessing and setting the volume of specific sounds within the game, so that certain shots or powerups did not ring more loudly than others. The functionality existed within FMOD.hpp, but was not accessed within any of the files included with other frameworks.

```
int fmod_volume(float i)
{
        FMOD_RESULT result;
        result = FMOD_Channel_SetVolume(channel,i);
        if (ERRCHECK(result)) {
                printf("error fmod_volume()\n");
                return 1;
        }
        result = FMOD_Channel_SetPaused(channel,false);
        if (ERRCHECK(result)) {
                printf("error channel->setPaused()\n");
                return 1;
        }
        return 0;
}
```

We created this function and called it within the playsound() function whenever any sound was to be played.

Conclusion

We very much enjoyed working on this project throughout the quarter. We learned a great deal about the software engineering process, and gained some valuable coding skills in the process (especially within the realm of program modularization!). There are so many additional features, concepts, and ideas that could have been implemented given more time, but we are quite happy with what we managed to accomplish in such a seemingly short period of time, and with only three people involved.



```
GAME OVER

Your SCORE IS 78
You made it to wave 1 in zone 1
Enter Name:
```