COS 426 Final Project Write-Up

Henry Peters
Mensheng Romano

Introduction

Our project, entitled Red Green Bullets (RGB), was our attempt to create a simple but fun game based on the common genre of video games known as bullet hells. In a bullet hell game, the player is normally given the ability to fire some sort of projectile, or bullet, and then tasked with killing numerous enemies and bosses. The enemies also have the ability to fire projectiles, and usually they are able to fire so many that dodging them becomes a real challenge. RGB is our take on this genre, given the allotted time.

The main goal of our project was to create a game that was fun to play. In our minds, a number of aspects contribute to how fun a game is. We wanted the controls to be smooth and responsive. A game based on dodging without responsive controls would most likely be frustrating and therefore not fun to play. We also wanted the game to be sufficiently challenging. Everyone has their own ideal difficulty for a game, but no game is very fun if it presents no challenge to the player. We wanted to make sure that players felt like they were accomplishing something by beating it, not just randomly clicking the screen. Finally, we wanted the game to look nice. Part of this was because this is a graphics class, so we felt the game should have sufficiently good graphics, but also a visually pleasing game is more likely to draw in players. This project wouldn't benefit anyone in particular, but it will hopefully bring a couple minutes of fun into some people's lives during these difficult times.

There are plenty of bullet hell games currently available for anyone to play. It is a fairly common genre, but due to its simplicity it was one we thought we could emulate in the short amount of time given to us. Two games in particular inspired us to do this project. The first of those being a game called Realm of the Mad God, a pixel, roguelike bullet hell. The players explore a procedurally generated world and dungeons, dodging bullets and killing enemies. The other game that we took inspiration from is called Crypt of the Necrodancer. This game uses sound and rhythm as an extra mechanic, but we liked the aesthetic of the crypt, which is how we chose the textures we used in our game.

Both of the games mentioned above are two dimensional, but we decided to make ours in three dimensions. We have become familiar with the three.js library throughout this semester, and we felt that it would make the most sense in the restricted time frame to use a library we already knew, rather than learn a new one. Three.js does have support for sprites, but we felt that the game would look better if we instead used polygonal meshes for the player and enemy models. This approach works well when the game being made is relatively simple, as drawing meshes in three dimensions is more computationally expensive than just using two dimensional images. We believed that our approach would work well for our circumstances, though, since we were only given a couple weeks to complete this project so we didn't anticipate the game getting overly complex.

Methodology

To obtain all the models for our game, we used the Google Poly project's website. This website allows users to browse through polygonal meshes and download them for use in their own projects. All the meshes we used are cited at the end of this write-up. To import the models into our project we used the three.js OBJLoader and MTLLoader classes. These allowed us to easily load both the geometry and materials of all the meshes we wanted to use. To create a scene we just used the three.js Scene object. The scene for our game was relatively simple, so creating it was not very difficult. We essentially just created geometries for the floor and walls and then added them to the scene object. We used the three.js class MeshPhongMaterial as the material for all the objects in our scene so that they could interact with the lighting more accurately than MeshBasicMaterial.

The lighting for the scene was also fairly simple. We utilized the three.js class point light for our lighting needs. We placed a couple lights around the scene just so players could actually see what was happening, but then we added lighting to some of the special attacks in the game. We initially wanted the game to be darker, but then have every projectile light up the screen, but due to the way three.js does lighting calculations, adding this many lights to a scene and then asking for the renderer to draw shadows lagged the game too much for this to be feasible. As a solution to this problem we just added lights to the special attacks of both the boss and the player. To do this we just added a point light to the projectile class we created and then moved the position of this light at the same rate and in the same direction we moved the position of the projectile.

After we had a model loaded for the player, we needed to decide how to best make it interact with other objects in the scene. Before we added projectiles we just wanted to get basic movement working. To implement player movement, we used JavaScript event handlers. We maintained a boolean variable that would signal whether a direction was pressed and then set this variable to true on a key press and set it to false on the release of that key. Then in the render loop for our program we checked if the variable for a given direction was true, checked for collisions in that direction, and moved the model in the specified direction if there were no collisions with objects in the scene. To calculate collisions between the player and the scene we used bounding boxes. There are a number of ways to calculate collisions, but using bounding boxes is both quick and easy to implement, so we decided it was the best implementation for our project. To check if the player would collide with a wall, we first computed the bounding box of the player model using built in three.js functions. An important thing to note is that after computing the bounding box of the player we expanded the box in the direction the player was attempting to move in by a small margin to get rid of errors caused by floating point imprecisions. We then looped over each child of the scene that wasn't the player themselves, a projectile, or a light and computed the bounding box for each object. We then just checked if the two bounding boxes intersected to check for a collision between the player and each object.

Projectiles were implemented by creating a class that held a mesh model, velocity vector, damage value, and a light for special projectiles. Each character on the screen could create one by calling the constructor and giving a start position, direction vector, whether the projectile was supposed to be special for special attacks, and optionally a light color. It was possible to create a class that extended the three. Group class which would have allowed us to directly add the projectile to the scene, but we instead just chose to create a mesh object inside the class instead because that's how we started the implementation and it didn't offer enough benefit to retroactively change it.

Player attacks were implemented by projecting the mouse position to the X-Y plane and creating a direction vector from the player's position to the projected mouse position. That way the player aims by moving the mouse and shoots by clicking. To add more complexity to the player's attack choices, a special attack was implemented. This was done by calculating a vector in front of the player and rotating it by set angles across a quarter arc. A projectile was created at each rotation creating a quarter arc of 10 projectiles. These projectiles were deemed special which meant that they generated light and dealt double damage. Since this attack was special and so much stronger, a cooldown of 500 frames was put into place. This was tracked by a global variable that incremented every frame. If the spacebar was pressed before the first 500 frames or during the 500 frames after using the attack nothing would happen. If the cooldown was fully elapsed, the special attack would be launched.

Enemy attacks were implemented in a similar, slightly simpler for most of the enemies, way as the player. Instead of calculating a direction vector, for the first and second wave of enemies, a static vector was chosen. For the first wave this was rotated four times around a circle to create an 'X' pattern of projectiles. The second wave simply shot their projectiles down the y-axis. The boss had a more complicated attack pattern. The base of it was made up of a single projectile that was aimed by calculating the vector between the boss' position and the player's position. The second attack used this same vector, but first rotated it by 0.25 * PI. Then five vectors were calculated along a quarter circle and a projectile was launched using each, creating an arc of projectiles aimed towards the player. The boss also had a special attack that launched a massive projectile at the player amidst these waves of bullets that would do tremendous damage if it collided. This was on a similar cooldown to the player's special attack, but it would be used every time it was off cooldown. We needed to implement a delay between enemy attacks because if they fired every frame, it would create a solid stream of projectiles that were undodgeable and almost instantly kill the player if they ever intersected. To work around this, a forty-five frame delay between attacks for each enemy was implemented using modular arithmetic and a frame counter. We thought about using asynchronous function calls to achieve this delay, but a frame counter was a simple, effective, and less involved way of solving the problem.

Projectile collisions were handled in a similar way to player collisions. Projectiles generated from the player were stored in one array while projectiles generated from enemies were stored in another. Each projectile had a velocity associated with it, and at each timestep, both arrays were iterated through and the projectile positions updated. The bounding box for each projectile was then calculated and the children of the scene were iterated through to check for collisions. If the projectile collided with an entity, the projectile was removed from its associated array and the health was removed from the entity. If the projectile collided with the wall, the projectile was simply removed from its array. The purpose of storing the projectiles in two separate arrays was to prevent collisions with allies or the projectiles creator on firing, as the projectile originated from inside of the creator's mesh. A field could have been used to track the creator, replacing the separate arrays, but using separate arrays was easier and allowed for a more efficient code structure.

In order to make hitting the enemy a legitimate challenge, the enemies needed to be able to move in a slightly unpredictable pattern. This was achieved by calculating a random position on one of the enemy's positional axes. A velocity vector was calculated towards that position and added to the enemy's current position at each timestep. Once the enemy reached the calculated position, a new random position was calculated and the process repeated itself. The enemy was coded to only move on one axis to help with code complexity due to the time constraints presented by the project. Ideally, a more complex movement pattern would have been implemented, but in combination with the player's low health and the number of projectiles that need to be dodged, a simpler movement pattern still results in the addition of enough challenge to keep it interesting.

Waves were implemented by adding boolean variables to track which wave the player was currently on. At the start of each wave, enemies were spawned in and added to the enemies array. As they died, they were removed from the array, and once the array was empty, the next wave would commence.

Results

We measured the success of our project by us being able to beat it and enjoy playing it. There weren't really any experiments we could run on our project, as it was just a video game, but we did playtest to make sure all our features were working as intended. Before we implemented projectiles, we moved the character around and ran them into walls to make sure we were not able to clip through them. We were not, so we considered this a success. We were likewise not able to clip through enemies. We then added projectiles into the game and made sure that every time the player was hit by them their health would go down. We also made this check for the player's projectiles hitting enemies. We also made sure that the projectiles disappeared when they ran into a wall or a part of the scene that was not the player or a light. Finally, our game was structured in waves, so we had to make sure that each wave spawned correctly, and that it did not spawn until the previous wave was defeated.

Discussion

The overall approach we took was very promising. We were able to architect the code in a way that allowed for the easy addition of more enemies and separate waves after achieving the base functionality of movement, collisions, and attacks. The game could be continued to be expanded upon given more time. If the game were to be expanded, a larger map, more complex enemy movement patterns, and more enemies would be key. Additionally, items or powerups to change player attacks and stats would help add more depth to the game. It would also lend more purpose to the player rather than just "kill the boss". The only approach that could have been better was extending three.group in the projectile class. This didn't have a large impact on the game overall, but would have resulted in slightly cleaner code. This project taught us how to build our own scenes from the ground up, how to create a strong development plan and timeline, and the importance of good architecture from the very beginning. We began working without much of a plan in mind and floundered a bit after we added a few features and had to keep going back and changing the structure to fit more features. After we sat down and planned it out a bit more before continuing to code, it made it significantly easier to add more features. By coding with the future in mind, it made going from just a single enemy to adding waves and multiple enemies at once much simpler.

Conclusion

Our goal was successfully obtained. Red Green Bullets is genuinely fun and a challenge which is exactly what we were going for. Given the amount of time we had to work on it, there is a solid amount of depth and complexity to it. The next steps would be to continue adding features and expanding the scope of the game. This game works as a sort of proof of concept for a three-dimensional bullet hell, of which there are few. Items, a class system, more complex enemies, and a larger map to explore are all different avenues that could be implemented and expanded upon in the future. The enemy class might need to be revisited in the future if the game continues to be expanded to account for a wider variety of enemies, but as of right now, it supports our features beautifully. Overall, this project can be considered a resounding success.

Contributions

Mensheng:

- Projectile design
- Special and normal attacks
- Boss design
- Boss movement
- Enemy design
- Enemy wave implementation
- Projectile collisions
- Enemy models

Henry:

- Scene design and creation

- Textures and lighting
- Player movement
- Player collisions
- Player model
- Health bars and indicators
- Projectile collisions
- HTML/Front end

Sources

- Player and boss model: https://poly.google.com/view/bN9DnRavONC
- Eyeball model: https://poly.google.com/view/24EHzvxK0qr
- Devil model: https://poly.google.com/view/f5YYKuZ9qs2
- Wall and floor textures: http://www.plaintextures.com