Defender: Final Project Writeup

Srija Makkapati, Katelyn Yang, Beiqi Zou

Abstract

Our project, Defender, is a spin-off of the popular shoot 'em up game Space Invaders. Space Invaders (1979) is the classic arcade game in which a player must destroy all the enemies using a fixed shooter that only moves horizontally across the bottom of the screen. Defender features exciting new customizations, difficulty modes, and enemy movements implemented on top of the classic Space Invaders game.

1 Introduction

1.1 Related Work and Motivation

The goal of our project is to create a game similar to that of Space Invaders in which we have the user-controlled player attack all the enemies in the space while avoiding incoming bullets along the way. Unlike the traditional Space Invaders invented in 1978, we created a game that was more interactive. The original Space Invaders game features a block of enemies who are slowly moving towards the player while missiles fall from the sky. The player is then tasked to dodge the missiles as well as shoot missiles of their own and defeat all of the enemies ahead. In order to create a game which was more interactive, we decided to implement enemies who directly shoot bullets from their ship and slightly change the objective of the game. Instead of aiming to obtain the highest score, the player of Defenders strives to play each level to completion. This game is great for those who obtain more satisfaction from completion of a game as opposed to competition among other players.

Figure 1: The original Space Invaders game with a block of enemies approaching the shooter.

One of the works that our group was most inspired by when creating this project was "Shape Battles: The Final Frontier" by Derek Sawicki, Hector Solis, and Jimmy Wu from the COS426 Final Project Hall of Fame. Similar to their project, we utilized an array data structure in order to store elements like our enemies and our bullets. This allowed for us to efficiently access these items which is important because they were constantly being used throughout the game. One difference in the implementation approach that the creators of Shape Battles used was initializing their array with an implicit array while we used an array constructor for our objects. Because we only use the objects initialized at the beginning of the game, no extra space is needed as long as the game continues since the bullets are constantly being recycled from the same array.

While previous similar games stick to a simple interface and gameplay, we decided to take the route of creating our own version of Space Invaders with new customizations and features that we hope will make the game more entertaining for players.

1.2 Approach

In order to replicate a game like Space Invaders, we started by creating a minimum viable product (MVP), which consisted of creating a player, enemies, a space background theme, an introductory, pause, and ending screen and bullets. The functionality of our MVP was creating a spaceship which moved horizontally along the bottom of the player's screen, and the movement is controlled by the left and right arrow keys on the user's computer. The enemies were created so that they were not yet moving but spawned in random areas along the top half of the screen, within the borders. The space background theme was initialized as a static black background with spheres (dots) of varying colors and sizes. The final aspect of the MVP was the bullets. Every time a player presses the ‘f’ button, a bullet originating from the position of the player at the time when the button was pressed is launched. The bullets move linearly upward from the start position at every time frame. If any of the bullets collide with the enemies, the enemy and the bullet disappear from the scene. Enemies also shoot bullets straight down starting from their position. After all of the enemies are shot and disappear from the screen, the ending screen is displayed. The introductory screen includes the name of the game and an instruction ‘Press Space to Start’. When the space bar is pressed, the scene changes to the game scene. The pause scene appears after the space bar is pressed during the game.

After creating a working MVP, we proceeded to upgrade the game by upgrading all of the features as well as making the game more complex and enjoyable. In order to do so, we created a more dynamic background with sparkling stars, as well as creating more complex movements for the enemies. We added visual elements such as sprites as well as an option to select the player’s character in the beginning of the game. A health-bar was added in order to make the game more challenging and give the player an incentive to dodge the bullets. Difficulty levels were also added in which the movements of the enemies get more complex as the level increases.

By starting with a minimum viable product and gradually increasing elements to make the game more complex, we were able to maintain an organized order in operating as a team since the tasks were clear and well understood. Additionally, starting with an MVP prevented our code from becoming sloppy as it became clear how to organize the functions and methods making modifications of the code simpler since it was logical how to navigate through it.

2 Methodology

2.1 Tools

We decided to implement our game with the three.js framework, since we knew we needed to alter the player and enemy positions with the help of vectors. Also, we wanted to take advantage of various three.js geometries (e.g. RingGeometry, TextGeometry, BoxGeometry) to make the game aesthetically pleasing. 2D sprites and animations were found on itch.io.

2.2 Project Architecture

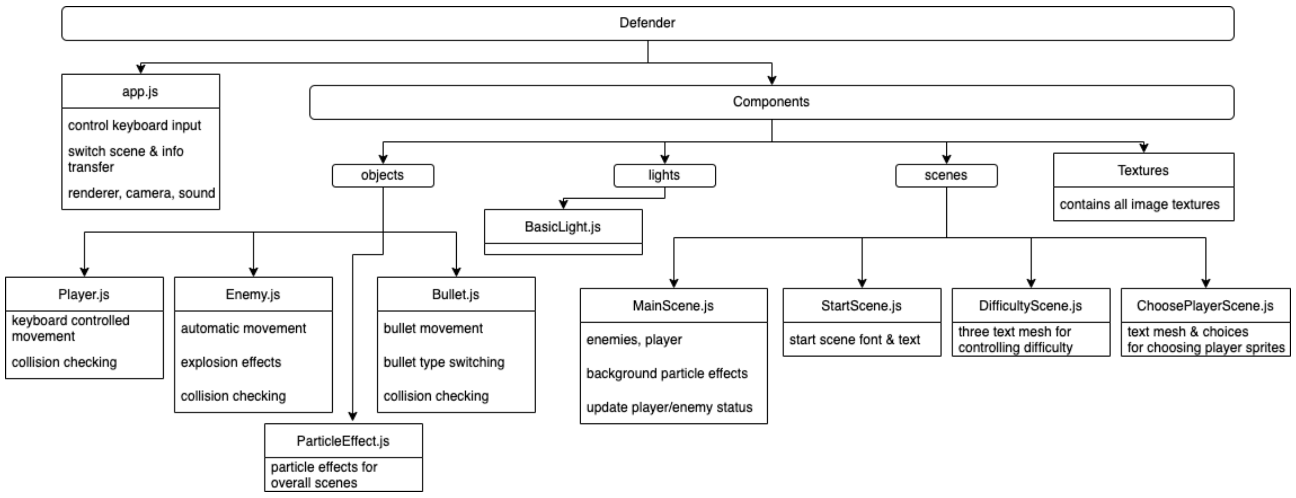

Figure 2: Overall project structure of Defender.

2.3 Player

We used sprites from itch.io as a representation of our main character. The player can use arrow keys to control horizontal movement of the character, and press “f” to shoot out bullets. The movement is done by using velocity and net forces, similar to the implementation in A5. Specifically, the character has a preset speed. Each time when the player presses arrow keys, a force gets added to the velocity, where force = speed * direction. Since we are constantly checking velocity and force updates at each timeStamp, we need to reset the force each time it’s getting checked, in order to prevent accumulative forces. In addition, we need to make sure the main character won’t go off bounds. Since we’re mimicking the Space Invader gameplay, the character can only move horizontally. Thus, we only need to handle the horizontal out-of-bound cases. Hence, we first preset boundary values. We then added a collision handler in the update function to check at each timestamp, whether the character position with some small offset is out of bound. If it is, then we set the character’s x axis value to the boundary that we preset.

2.4 Enemy

Similar to players, we also adopted sprites from itch.io as a representation of enemies.

2.4.1 Movements

Unlike player movements, which are controlled by keyboards, enemies have random/specific patterns of movements. Specifically, in easy level, the enemies are static; in medium level, the enemies move horizontally; whereas in hard level, the enemies have (pseudo) random motions.

Directions of “pseudo” random motions depend on the random number generated. It has 60% probability of having a completely random direction, which is calculated by $\text{dir} = \text{Vec3}(\text{random}()\times 2-1, 0, \text{random}()\times 2-1)$, where $(\text{random}()\times 2-1)$ is in range $[-1,1]$. It has a 40% probability of moving in the direction of the player, which can increase the difficulty of the gameplay. The direction change will only occur if the enemy either hits the wall or hasn't changed direction for a certain amount of time. In scenario 1 we want to make sure that the enemy will not be stuck in a certain position.

2.4.2 Random enemy generation

Random enemy spawning is based on several aspects: level difficulties, last enemy spawning time, spawn probability and number of enemies spawning at one time. Level difficulty determines the maximum number of enemies that can be generated in one screen. If the number of enemies in the scene reaches the maximum number of enemies, it will not spawn enemies anymore. Further, in order to prevent continuously spawning enemies, we set a spawning probability based on last enemy spawning time. Specifically, we want the spawning probability to increase as the time elapsed (timeStep $-$ lastSpawningTime) increases. Thus, we set $spawnProb = (elapsedTime/1000 \times 60 \times 60) \times 0.4$, where $elapsedTime = timeStep - lastSpawningTime$. If there's no enemy on the screen, then the spawning probability is set to 1. In order to make the game more random, we also randomly generated the number of spawning enemies, but with a fixed minimum which is preset based on the difficulty level (can be found as init_enemy_num).

2.4.3 Collision check and position update

Similar to collision detection in player class, we also used positional checking strategy to determine whether an enemy object is out of bound. Original approach was to directly change the moving direction to an opposite direction, i.e. if the enemy touches the upper bound, then the next moving direction of that enemy will be downwards. This change was implemented in the check-collision function. However, this approach lacks randomness and it was predictable for the player. Hence, instead of directly changing the direction, we used a boolean value to store whether it hits the wall or not. Then we included this scenario in the move function instead – if the enemy is hitting walls, then immediately changes the direction.

2.5 Bullets

In order to implement the player and enemy bullets, we created a fixed array for the player bullets as well as a fixed array for each of bullets for each of the enemies. In the beginning of the game, all of the arrays are initialized as a new bullet that starts at the position of either the enemy or the player. When initialized, all of the bullets have an opacity of 0 and only appear once the specified key is pressed. For the enemies, the bullets spawn automatically and their opacity is only set to 0 if it collides with the player or if it leaves the frame. Similarly, if there is a bullet and enemy collision, the opacity of the bullets are set to 0. The bullets with 0 opacity are considered 'dead bullets' which can be reused if a bullet press occurs. The bullet outputs are limited by a timer in order to prevent spamming the bullet keypress. The bullets of the enemies are updated so that the z value of the bullet decreases giving the effect that the bullets are shooting downward. On the other hand, the bullets of the player is updated so that the z value of the bullet increases which gives the effect that the bullets are shooting upward.

2.6 Scenes

We implemented a number of scenes in the game to make the user experience flow smoothly. In our game, users interact with and navigate through scenes with simple keypresses (Space key to move through scenes, arrow keys to toggle through difficulty modes, and Esc key to return home after the game ends). In this section, we detail all the different scenes featured in the game.

2.6.1 Start Scene

The user first sees a start screen, which we implemented by creating a basic scene (StartScene.js) with a couple of TextGeometry instances to show text on the screen. We found a couple of fun space-themed fonts to use for dramatic effect, and did some hard coding to set the text positions according to the camera position and orientation.

2.6.2 Difficulty Mode Scene

We then implemented a scene (DifficultyScene.js) to allow the user to toggle between Easy, Medium, and Hard difficulty modes. The keypress events are handled in app.js. We rendered text and selection boxes using instances of TextGeometry and BoxGeometry, and we implemented the illusion of the user selecting the difficulty mode by changing the color of the "currently selected" difficulty mode from black to white.

2.6.3 Choose Player Scene

At the time of submission, we do not have a choose player scene implemented, but we plan on allowing users to select their desired player sprite. We will format this scene similarly to the Difficulty Mode scene just to keep continuity and simplicity.

2.6.4 Main Scene

The bulk of our game logic lies in MainScene.js, where we implemented game features including but not limited to the player shooter, bullet movements, enemies, health bar, a starry background, game over/victory/pause views. We describe the various elements of MainScene throughout this paper except for the health bar, so we discuss it in this section. We considered a variety of potential health bar implementations, but we settled on a green health bar shaped like a ring that gets smaller and more red and the player loses health. We chose to implement it as a ring so that it obstructs less of the screen, and there is no chance that the health bar obstructs an enemy that might be spawned in that area of the screen. We implemented this feature with THREE.RingGeometry, adjusting the size and angle parameters of the geometry when the player loses health. The health ring is initialized to have 8 sections or segments, and since a full circle corresponds to 2π radians, we subtract the endAngle of the ring by $\frac{\pi}{4}$ every time the player loses health. FInally, to compute the color of the ring, we compute a linear combination of the pure colors green (hex: 0x00ff00) and red (hex: 0xff0000) using as coefficients remaining health and $10 -$ remaining health respectively.

2.6.5 Background Particle Effects

The inspiration for star particle effects comes from three.js examples, where they used THREE.Points to generate a set of particles, each of which has a certain geometry and shader material. In most of the cases, they used a 3d perspective camera to achieve moving/changing visual effects, since the perspective camera can capture the z-depth. However, in our case, we're using an orthographic camera, where changing particles' z value will not affect the visual effects of the particles. Hence, instead we found that changing the size of the particles may achieve similar effects. We changed the size of the particle based on a sine function, specifically, $1.8 \times (1 + \sin(0.1 \times idx + \text{time}))$, where idx is the index of a particle, time is timeStamp$/500$. 1.8 represents the scale of a particle.

2.7 Audio and Sound Effects

To make our game more entertaining and interactive for the user, we incorporated a number of simple audio elements. When the player navigates through the various scenes (Home screen, choose difficulty mode screen, pause screen, etc), they hear an appropriate clicking sound. Similarly, when they click through the difficulty modes, a different clicking sound is played. Shooting sounds are heard when the player shoots a bullet, and a destroy sound is played for bullet collision events.

3 Results

3.1 Evaluation Metrics

In order to evaluate the success of our project, we used three broad key metrics:

- The game is functional and works as intended.
- The game is visually appealing to the player and incorporates artistic components.
- The game is entertaining and challenging.

3.2 Evaluation

In order to test whether the game is functional, we made sure to play through the game and observed the effects of the bullet hitting the enemies, the bullet missing the enemies, playing until the player ran out of health, and playing until the prescribed amount of enemies disappear. We found that the game was functional when playing in the way that was intended and there were no scenarios or corner cases that returned any errors or crashed the game.

In order to make the game visually appealing, we chose to incorporate a space theme which includes a dynamic starry night background. We used sprites found online in order to replace the original triangle mesh used for the player and enemy and the rectangle mesh used for the bullets. Additionally, we added sound effects to bullet shots as well as for entering the game which makes the game more engaging.

We made the game challenging by adjusting the speed and movements of the enemies. In the easiest level, all of the enemies are stationary but because bullets from the enemies shoot directly below them, the player must be able to effectively time their movements to avoid the bullets and find the right time to go underneath to shoot the enemies above. Even in the easiest difficulty, there is still a challenge as the player can only get shot 4 times. As the levels increase, the movements in the enemies increase as well. In the medium level, enemies move horizontally so the player not only has to dodge the bullets coming their way but they also have to predict where to shoot the bullet as the spaceship will not be in the same position when it reaches the same horizontal position. Finally, in the hardest level, not only do the amount of enemies increase but the movements of the enemies are less predictable as they follow a more random pattern of movements. In addition to ensuring that the game was difficult, we also made sure it was winnable so we made sure to test the game enough to know that an advanced player can be victorious.

Because we were able to successfully achieve all of the objectives outlined above, we can conclude that the project was successful. While there are certainly more improvements we plan on implementing, our resulting game is satisfactory given the time restrictions we were under.

4 Discussion

4.1 Implementation Challenges

Throughout our development process, we faced a number of implementation challenges, which we will briefly discuss in this section.

4.1.1 Handling Out-of-Bounds Bullets

We implemented moving bullets but initially found that they wouldn't leave the screen once they went out of bounds (they would stay stuck at the top of the screen). Our initial workaround was to allow the bullets to continue moving for an indefinite amount of time off the screen, but we figured this would lead to the game eventually lagging. Our final fix was to make the bullets "invisible" once they hit the top of the screen, reset their positions to the tip of our fixed shooter, and then make them visible again once the player shoots.

4.1.2 Background Sparkling Effects

The usual approach for sparkling effects would be to adjust the depth or how far the particles are from the camera. However, this method only works for Perspective cameras. In our 2D top-down game settings, we use orthographic cameras in order to display correct player/enemy sprites.

Hence, we tried to resize the particles themselves based on a sine function, in order to achieve sparkling effects.

4.1.3 Standardizing Game Element Positions

An existing bug in our implementation currently is that the positions of various game elements vary depending on the user's screen size. For example, the health bar shows up in an appropriate location on Srija's screen, but Beiqi and Katelyn see the health bar in an oddly misplaced location. We suspect this might be because we hard-coded the game element locations, and we plan on standardizing the locations by resizing them according to screen/window size.

4.1.4 Death Animations

We found a gif for the death animation feature, and we initially tried implementing animations using THREE.VideoTexture and loading the textures to the dead enemy sprite, but we ran into issues with gif and mp4 files not being recognized when we loaded them. As a workaround, we used the 10 images that were initially put together to form the animation gif and essentially looped through them every time an enemy was destroyed. We wrote a function in the enemy class that only gets triggered when an enemy is destroyed, and tied it to the main render loop through the Enemy class update function. We used the timestamp passed to the animation frame handler and a global variable containing the time of the explode function's first call to compute which of the 10 images to load to the sprite texture, and then load the texture within the explode function.

4.2 Future Work

While we implemented a number of new features on top of the classic Space Invaders game, we plan on adding more customizations and features to further improve the game's interactivity and aesthetic/entertainment value. First, we plan on adding functionality to allow users to customize what they want their shooter and enemies to look like. Our first step for this feature would be to allow the user to use their arrow keys to cycle through a number of precreated shooter sprites and select the one they want to use. A further improvement may be to allow users to also upload images they want to use as sprites if they wish.

Secondly, we plan on implementing infinite levels, where the user will face faster and faster bullets and enemies as the levels progress. With this comes the potential of implementing a "record" feature or leaderboard feature, which enables users to track the record number of levels they reached and compare their performance against that of their friends.

Lastly, we would like to improve the game's aesthetic value by, for example, adding apt background music and player animations for when the player is hit. There is a lot of room for potential customizations in this area. For instance, we can allow the user to adjust the background music volume, adjust the game sound effects, change the color and background themes as the user wishes, etc. We might also consider adding fun particle effects, different types of bullets and different powerups for players to collect.

4.3 What We Learned

This project gave us the opportunity to learn what it takes to create an end-to-end game, as well as develop our resourcefulness in terms of debugging.

5 Conclusion

All in all, our approach has been effective and we achieved most of goals we set out at the beginning of the project. Throughout the implementation process, we learned fairly amount of knowledge in three.js, gameplay designs, collision handling and physics.

Moving forward, as we mentioned in Future Work section, we want to implement a few extra features, for example allowing players to select main characters or even upload their own images, as well as having infinite levels and more aesthetic visual effects.

One of the key issues that we need to revisit is how to make the proportions of the game more consistent even when the size of the browser changes. When minimizing/maximizing the browser, the enemy and player sprites look a bit distorted or stretched. Ideally, we want to ensure that when we change the size of the window, the shape of sprites will not be affected.

6 Contributions

Srija worked on creating the scenes (start scene, difficulty mode scene, pause/game over scenes), scene transitions, the enemy death animation, and health bar, as well as found the 2D player/enemy sprites and fonts used in the game. Beiqi worked on player and enemies (movements, controls, spawning, collisions), background particle effects, bullet sprites implementation. Katelyn worked on bullet movements coming from enemies and players, implemented bullet collision effects, and added falling power-ups to the game. All three of us worked on the writeup and presentation, and we collaborated on designing the new features we added to the game.

7 Works Cited

- https://gero3.github.io/facetype.js/
- https://mixkit.co/free-sound-effects/click/
- https://github.com/mrdoob/three.js/blob/master/examples/webgl_buffergeometry_points_interleaved.html
- https://soundbible.com/470-Laser-Blaster.html
- https://threejs.org/
- https://itch.io/game-assets/free/tag-2d/tag-space
- https://www.zapsplat.com/music/game-negative-tone-lose-life/