Abstract

Spaceship Trainer is a first person shooting game placed in a 3D scene simulating space where the user has to maximize the points they get by gunning down moving enemy ships. It was implemented with the 3-Dimensional Three.JS library in Javascript, along with HTML and CSS for styling. The main features the project implements are a 3D scene, lighting, user input and computer control over elements. Additional features that were included were texture mapping, on-screen control panel, collision detection, and sounds. The use of these features allow the user to experience a simulated game that was ideated as the 3D version of the vintage arcade games Space Invaders and Galaga. The game currently has three types of ships, all with different movement patterns, which spawn in with increased speed and numbers as the game goes on. All ships are destructible and the game ends after the player is hit by an enemy ship.

Introduction

Our project is Spaceship Trainer. The overall goal of our project is to imitate classic arcade games in a modern way that involves 3-dimensional space and movement. Therefore, we implement a first person game in which a user can shoot a ray from the current camera position towards an enemy ship and move and move not only left to right, but also down and up while perpetually going forward by having all the objects come towards the camera position. Think of the player as being a turret on the front of a small ship travelling at hyper-speed. We hope the game provides classic arcade game lovers the experience of playing a 3D version of a classic game and attempting to beat their own personal score.

Previous Works

We have come across various games and personal projects that implement a spaceship shooting game. One such example is found https://nickyvanurk.com/void/ . Some of the successes seen in these games have involved better performance and less lag on the gameplay. However the failures of these games are also that the gameplay is done on stationary objects rather than moving objects. The scenes that were implemented were objects such as stars, ships, and meteors in place with the user spaceship and lasers being the only moving factors of the scene. Although this does improve performance, the gameplay is not as fun as having to avoid the enemy spaceships and or trying to hit moving enemy ships which is much more difficult to do. The difficulty of a game is what makes a user enjoy the experience much more, which is something we hope to bring.

Methodology

The main requirements listed for the project specifications were all implemented for the project: 3D perspective viewing and objects, Lighting and smooth shading, User input, Computer control over some elements of the scene. There were several implementations which could have been taken instead of the ones we did. However, these would have resulted in a very different product. To include the 3D perspective viewing and objects, we combine the use of the Scene(), WebGLRenderer(), PerspectiveCamera() and other Three.JS functions to implement a 3D scene onto the browser screen that allows the user to view 3D represented objects and use their cursor to change perspective or the viewing angle from the scene that

allows for objects at a far distance to appear smaller than objects up close to the camera position essentially mimicking the view from a human eye onto the distance. There was another camera angle that was considered at first which was an orthographic camera, which would allow the user to view objects in the scene as the same size despite the distance from the camera position. However, we wanted to give the user a view that would simulate what being in space would realistically look like, so we decided to implement the PerspectiveCamera view. In the 3D Scene, we added lighting through the use of the AmbientLight function that illuminates objects in the scene equally from all directions. As the player is moving through a starfield where a star has equal probability of being in any given direction from the player, we felt that ambient lighting was a reasonable approximation for that effect. We allow the user to give input, through the use of the keyboard and the mouse. At the beginning of the game, the user has to click a button that starts the game. Through the use of these inputs, the user can control certain elements of the scene and manipulate how they view the scene. Throughout the game, they can press the mouse down that will perform a specific action like shooting a ray from the current camera direction. Similarly, the user can move the mouse around to change the perspective they get of the 3D scene essentially changing the camera direction. The user can also use the arrow keys, or the w, s,a and d keys to move the camera position up, down, left, and right, in its respective axis by a specific velocity on the 3D scene. Along these main requirements, we implement four additional requirements: texture mapping, on screen panel, collision detection, and sound. We implement two different types of textures for the objects in our scene. One of these are the star textures in which we use the TextureLoader function to render groups of sprites, which are planes that always face towards the camera position no matter the direction in which they are first rendered. The first implementation we had used for this texture was within the Points class. For some reason the Points class was not properly rendering alpha values from .png images, so we chose to switch to the sprite class which rendered alpha values properly.

Player Movement:

The player is able to move up/down/left/right within a limited square within the YZ plane. The player can look around within a limited angle, to ensure that they never look behind themselves. Think of the player as a turret on a small ship which is always moving forwards and never looking back. Movement was built on top of an existing Three example module known as ‘PointerLockControls.js’ which allows the player to lock their mouse to the screen and look around.

Star Spawning and Movement:

To create the illusion of movement through space, as though traveling through some hyperspeed portal, we used sprites of stars to simulate a moving field of stars. First we load the three textures of stars we have, for the blue, white, and red stars. We then randomize the x, y and z coordinates between -500 and 500. The sprite’s positions are then set to these coordinates and added to the group of stars. In 1 group of stars there are 1000 stars of each color and there are two groups that we keep track of in total. As the stars go past the player's camera position, we keep track of the difference in position between the starting group x and the camera position x. If the distance in x position gets too large, we set the group position back to be in front of the camera at a certain distance that is past the ending x position of the other star

group so it appears in the near future after the other star group gets too far. We ended up using sprites instead of geometry to render stars because we wanted to render a *lot* of stars to make the scene look interesting, and sprites were more efficient to render than geometry, even low-poly meshes.

Collision Checking:

There are two types of collision which are being checked in this game: collision between a blaster shot and an enemy ship, and collision between an enemy ship and the player. For blaster shot collisions, once per frame we iterate through all blaster shots. Each shot has its projectile's position stored, as well as the original ray which the projectile was cast from. As projectiles only travel along their ray's path, we were able to make an optimization on collision checking. We initially check the intersection of each projectile's original ray with each enemy ship. If there is a ray collision then we check for a projectile collision. As ray collisions are much faster to check than geometry collisions. Geometry collisions are checked using the three.js box3 object. Each piece of geometry getting checked for a collision has a bounding box drawn around it using Three's built in geometry to bounding box function, then those bounding boxes are checked against each other for collision. In the case of enemy ships colliding with the player, those collisions are only checked for enemy ships which are close to the player (their x position is less than 1 unit away from the player). Checking player collisions consists of drawing a bounding box around the nearby enemy ship and then checking if the position of the player camera is within that bounding box. We chose not to give the camera a bounding box as it would force the player to get a feel for a personal 'width' which they could not physically see, so we chose to only use the point collision of the player's position so that there is no arguing of whether or not a player death was valid.

On-screen display:

In the beginning of the game, we display a screen with the game's title, instructions on how to play the game, and a button that once clicked initiates the scene by calling all the necessary functions. Another on-screen display is then displayed on the top center of the screen. This was made with an orthographic camera and canvas that stays at the same position as you move around the scene.

Difficulty Increase:

Difficulty increases on a logarithmic scale with time. The longer a player plays, the larger the amount of ships which are spawned per spawn cycle, and the faster those ships go. We chose a logarithmic difficulty increase to allow a quick initial difficulty jump to a reasonable mid-level difficulty, then a reasonable leveling off of difficulty increase for the long term as survival becomes much more difficult.

Ship Modeling and Texturing:

We modeled and textured the ships in Blender. Loading in the models to Three was fairly simple using Three's GLTFLoader object, which allowed us to generate our loaded models and their textures from their exported GLTF formats.

Ship Spawning:
A new set of N ships, N scaling logarithmically with time, are spawned every second within a random but bounded point in the same YZ plane, which will then move towards the player on the x-axis at a set speed which also scales logarithmically with time. Each spawn is simply copying the initial model for a given ship which was loaded in. We chose to not load in each new ship using the loader function because load operations take a long time and happen asynchronously, making mistakes referencing an object that does not exist yet easier. There are three different ship types that can spawn: red ships, yellow ships, and green ships. Red ships are simple and only move forward. Yellow ships move left and right as they go forward. Green ships move in a spiral as they go forward. Each green and yellow ship spawns with a random offset factor to ensure it is not synchronized with other ships of its type which spawned in with it. Then, each green and yellow ship's z and/or y position is changed according to a sine function using both time and each ship's offset factor as arguments. We felt like adding unique movement patterns would make the game more interesting for the player by making certain ships harder to hit. Note that the difficulty increase in hitting the moving yellow/green ships is balanced out by their bulkier profiles which are easier to hit than the slender red ships.

Sounds:
We have 4 sounds which will play in our game: a laser sound, a destruction sound, and a "Game Over" and a scream sound. For every event in which the mouse is clicked down, a laser sound is added to an array of sounds. Then the array of laser sounds is iterated through and played. This is so you can hear them back to back giving the effect of one shooting after another without there being any delay. The destruction sound is played in the event that the laser shot from the camera's direction intersects with one of the ships, therefore "destroying" it. The last two sounds are played simultaneously to signify the game is over. This occurs when you collide/crash with an enemy ship.

Discussion and Conclusion

Success of the project was measured off of the amount of goals met based off of priority and how many of the features, required and additional, we were able to implement successfully. We met all of our expectations for the project that we initially thought of such as laser shooting from the current camera position based off of the camera direction, movement of the ship to go up, down, right and left, as well as a perpetual movement forward.
We are confident in the final product of our project as it contains many of the ideas we envisioned for the project's MVP, as well as including some stretch goals that we wanted such as having ships spawned instead of targets, as well as having them moving. The next steps would be to make the ending condition of the game when an enemy ship fires lasers back at you and it intersects with your ship. This would be implemented by keeping track of all the enemy ships' lasers shot and iterating through them to see if they intersect with a box around the camera position. Some reasons we were not able to implement this was because of not enough time, the difficulty of the task, and performance decrease that would happen from having to track these and iterate through them for every time step. Other next steps would be to make the scene more elegant through ship models, movement of stars that look like shooting stars (with visual ghosting trails), implementing other objects that exist in stars such as meteors,

comets, and planets. In its finished form now the project may have some unfound memory leak glitches from objects not being removed, although we have accounted for the most obvious cases. We haven't done enough stress testing to check other cases. Also, if the user clicks off the screen they can not click back on in this final iteration of our project.

- David's Contributions
  - Player Spaceship movement and look controls
  - Enemy models, textures, spawning, and pathing
    - Beautifying Green Ship
  - Collision checking
  - Difficulty increase over time
  - Initial creation and tweaking of dual-starfield movement to create continuous movement effect

- Kevin's Contributions
  - Sound effects for the spaceship, game over
  - Ray shooting
  - Star movement spawning
  - On mouse click events
  - On screen display
    - Making scene appear on click of button
    - Beginning screen code to begin game
    - Score counter using orthographic camera and canvas
    - Game over page
    - Resetting the scene

- Works Cited

The library from this example, pointerlock, was built upon to make ‘test.js’ (naming it anything else caused a bug) and movement controls were also from this article:
https://github.com/mrdoob/three.js/blob/master/examples/misc_controls_pointerlock.html

This was used as a general tutorial for adding objects and lights to a scene:
https://discoverthreejs.com/book/first-steps/first-scene/

This instruction guide was used to make the hud, with some code copied from it:
https://www.evermade.fi/story/pure-three-js-hud/

This is the crosshair asset:
https://www.nicepng.com/maxp/u2q8u2r5u2w7t4t4/

This was used as a guide on how to create sprites in Threejs:
https://en.threejs-university.com/2021/08/03/chapter-7-sprites-and-particles-in-three-js/

This was used to figure out which angle to use for camera raycasting:
https://stackoverflow.com/questions/31220969/three-js-raycast-from-camera-center

Code was copied from here to light the scene:
https://threejs.org/docs/#api/en/lights/AmbientLight

Initially we attempted to more formally create paths for ships. This was later scrapped, but here’s the guide we used before it was scrapped:
https://observablehq.com/@rveciana/three-js-object-moving-object-along-path

This is the death sound:
https://bigsoundbank.com/detail-0477-wilhelm-scream.html

This is the death game over sound.
https://pixabay.com/sound-effects/search/game-over/

This is the laser sound
https://pixabay.com/sound-effects/search/ray-gun/

This was to help remove objects from the scene.
https://stackoverflow.com/questions/18357529/threejs-remove-object-from-scene

This was to help ensure there were no unintended effects of restarting the game:

https://stackoverflow.com/questions/52040914/resetting-the-value-of-timestamp-in-requestanimationframe

This is the white star png
https://www.freeiconspng.com/images/stars-png

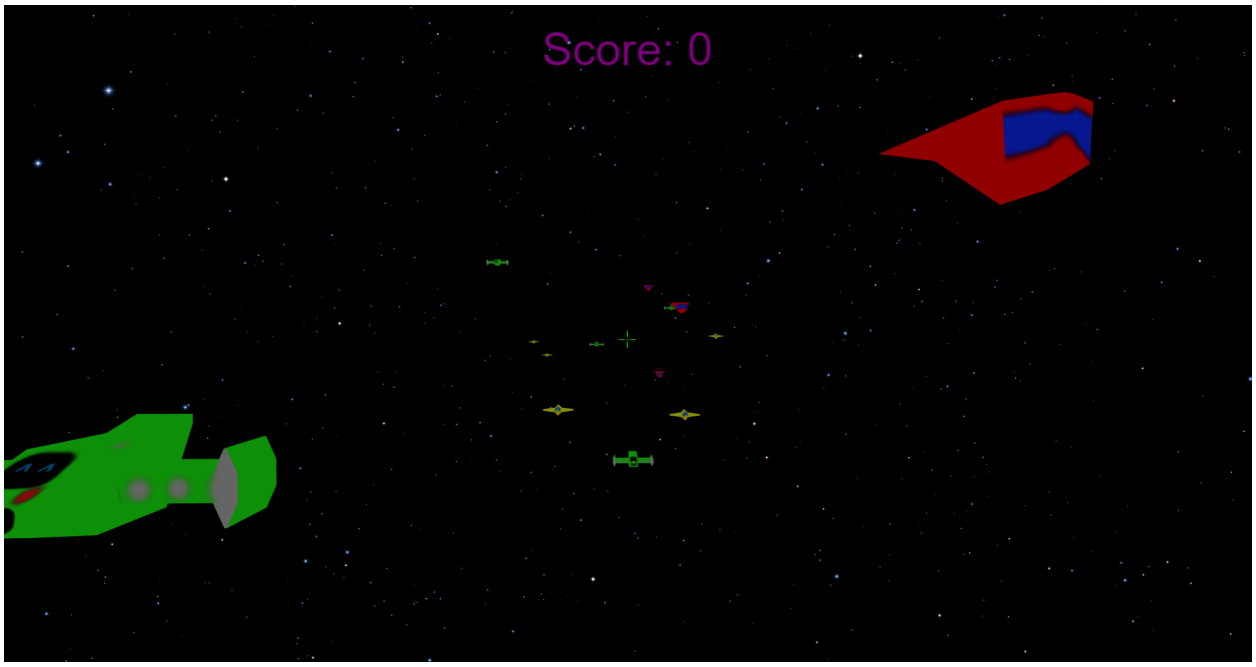
This is the red star png
https://www.pngkey.com/maxpic/u2q8a9a9u2a9t4i1/

This is the blue star png
https://www.pinterest.com/pin/613967361693856645/
This is the space background used for the html pages
https://wccftech.com/space-isnt-as-colorful-as-it-seems/

Pictures:


SpaceShip Trainer
The rules of the games are as follow:
Move up, down, right and left with the arrow keys on your keyboard or w,a,s,d keys
Move your mouse around to change your perspective. You CANNOT look behind you
Mouse click to shoot through the reticle. Get as many points as you can.
DON'T CRASH WITH AN ENEMY SPACESHIP or the game will end
Begin


Score: 0


Score: 0


GAME OVER