Space Defender

Tuan Dinh, Laura Reyes, James Richardson

Abstract

An 80s arcade style space-themed game where the player is tasked with saving the earth. The player tries to dodge the many space obstacles such as asteroids and not so friendly enemies with space weapons. Players must try to keep up with the increasing pace of the revolutions proportional to the number of revolutions they have successfully survived. Players must try to last as long as they can without losing all of their health.

Goal

We tried to create a game that had a nostalgic design that people can recognize but using the modern techniques for graphics we have learned throughout the course of the semester. We felt that we missed the simple task and pixelated design that came with older games. We wanted to make a game for those who equally missed these mini retro games.

Previous Work

We were inspired by a game called Light Bike Runner(1). This is a game that is tasked with customizing your own cyberpunk inspired light bike and trying to dodge any obstacles and trying to not fall off the platform. You are also able to collect coins in exchange for other items to customize your bike with. We also took some initial inspiration from the movie “Tron” but ultimately we leaned more towards the space themed concept because it was easier to find models that communicate the setting easily and in an immersive way to the player. We thought that Lite Bike Runner is repetitive in the sense that the trail the bike rides along does not really let you see anything else in the landscape. It mostly is just a black screen. We wanted to create a more immersive experience that actually felt like you were in space. Though we did think that Lite Bike Runner successfully added a lot of ways for the player to completely customize how their bike looks. We also added a different type of movement that the user can control. Rather than just trying to balance the bike along the track, we have different lanes so the goal of the game is not relying on falling off the track but rather being able to keep up with the speed and dodging attacks from enemies and asteroids.

Approach

We knew that our game would have an infinite track based on our prior knowledge and experience with games. We decided to go with a circular loop as opposed to a straight track as it seemed to compliment a player-centered camera orientation. Our inspiration from Lite Bikes and a "racer" concept also influenced our choice for arena shape. This was solidified even further as we agreed to adopt a space and sci-fi aesthetic for the game.

Aside from the track and player components, we add difficulty to our game with obstacles. Likewise, we add a motive with coins. Together, these aspects form the base gameplay loop. To add even more difficulty and move our game away from a simple endless runner, we add an obstacle variation that can shoot at players.

Our MVP was to have at the very least a working game where players can control a vehicle, move it laterally on a track and dodge obstacles with no projectiles. To accomplish our final result, we relied on the Three.js library for making our game with Cannon.js as the 3D physics engine. We chose this over other engines due to its conveniently implemented primitives and helpful documentation. Particularly, object collision detection was easily implemented with Cannon.js.

Methodology

Game Aesthetics

To achieve a "retro" art-style reminiscent of early 90s shooters, we levied ThreeJS's built-in post-processing functionalities (namely: pixelation at render-time, as well as a final Unreal Engine bloom pass over the scene. In addition, almost all of our 3D models are overlaid with a special holographic material (courtesy of @ektogamat on GitHub) designed specifically for application onto a THREE.Mesh in ThreeJS.

Game Objects

The ObstacleManager class handles all spawn, model rendering and removal logic for all game objects. The spawnObstacle() method creates each obstacle and adds its mesh and body into scene and world, respectively. To do this, the method first checks if the maximum number of obstacles in the world has been reached. If it has, then skip all spawn code. If it hasn't, the method calculates the object's spawn position. One constraint that we implemented was a spawn "window" for objects, with a minimum and maximum angle position (relative to the center of the track object) to ensure that objects don't spawn too close or too far away from the player.

Objects are removed either by triggering a Cannon.js collision event or falling too far behind the player. Both are handled in the update() function in the ObstacleManager class. update() performs constant checks for boolean flags indicating if an object should be removed as well as calculating the relative angle between objects and the player. Once an object's relative angle passes a threshold, it is removed. The object spawn and removal windows are illustrated as a rudimentary whiteboard sketch.

We handle all FBX 3D model rendering in ObstacleManager as well. Originally, we attempted to load models asynchronously once and just set the mesh property of each obstacle class to the corresponding FBX mesh. However, this resulted in buggy behavior where objects were spawned but did not display their associated mesh. Upon empirical testing runs with varied maximum models, we determined that the mesh was only displaying for the most recently generated instance of each object. Thus, the FBX was not being properly cloned. To handle this, we cache loaded FBX models and initialize object meshes accordingly in the spawn logic.

We introduce the Obstacle class as static game elements for the player to avoid. Obstacles are initially created with a standard Three.js box geometry and phong material. Obstacle collisions are handled by the Cannon.js Body class. To ensure that obstacles do not collide more than once, the Obstacle class has a handleCollisions() to mark obstacles for removal.

Enemy is a class that extends the core functionality of Obstacle. Enemy objects have the ability to move laterally and shoot projectiles at the player. The shoot() function creates Projectile objects and stores the projectile collision handler. The first implementation of this collision attempted to remove the projectile object from the scene and world completely, but caused issues stemming from the code trying to reference these removed projectiles. Thus, projectiles just become invisible upon collision with the player until they are removed.

Projectile removal is handled by updateProjectiles() (similarly implemented with update() in ObstacleManager). This method is called in Enemy's main update() function, which calculates and updates lateral movement and checks if the enemy is allowed to shoot at the player. This condition depends on if the enemy is in front of the player, the shooting range of the enemy and whether the enemy has waited long enough between shots. This ensures that players cannot be shot from behind, too far ahead, or be "spammed" with projectiles.

Coin is a simple class that extends Obstacle and serves to help quantify how well a player did.

Perspective

Our game has two main camera perspectives: a static camera oriented towards the main game arena (for use in the startup and game-over splash screens) that is appropriate for moments where the user is not actively playing the game; and a dynamic camera perspective that hovers behind the user's player model. Notably, the dynamic camera perspective is responsive – even during and after player movement between lanes.

Player Shooting

Originally, we planned on allowing the player to defend themselves by shooting projectiles at enemies and static obstacles; we wanted to replicate the enemy-to-player projectile shooting logic using Cannon, allowing for player-to-enemy projectile collisions, with enemy/obstacle kills yielding a weighted addition to the player's score.

However, the lane-based nature of our player movement mechanic – contrasted with the more free-flowing movements of enemies and obstacles – forced us to choose either (a) seek-based shooting mechanics, such that projectiles shot by the player automatically seek the closest enemy or obstacle; *or* (b) a similar shooting style seen in that of the enemies.

Firstly, neither option (a) nor option (b) would have been entirely ideal: option (a) would have removed most of the difficulty of our game, since players could just keep shooting without much thought (since their projectiles will always hit the closest enemy/obstacle). Option (b) is certainly better in terms of maintaining our game's difficulty, however, the lack of a player-influenced rotation of the player model mesh complicates figuring out what direction the player's shot projectiles would travel in.

In an attempt to not do away with player-to-enemy shooting entirely, Tuan constructed a working implementation of option (b), in which projectiles shot by the player traveled in the direction of the player mesh's forward vector. Eager to work these changes into the current branch, James and Laura pulled Tuan's changes, only to discover that the projectiles shot by the player travel in a different direction than on Tuan's machine.

After verifying that our struggle wasn't the result of mere Git ineptitude, we noticed that when building Tuan's version of the player-to-enemy shooting mechanics on James and Laura's machines (M3 MacBook Pros), several OS/architecture-based conflicts arose (namely, MacOS security conflicts with processes inherited from Tuan's version of the node_modules folder), suggesting that this difference in projectile direction may stem from differences in the built versions of our ThreeJS and CannonES dependencies, since the CPU architectures of Tuan's machine (an M1 MacBook Pro) differ from that of James' and Laura's machines, such that different versions of the dependencies were built to accommodate two different CPU architectures.

Not surprisingly, we decided thereafter to scrap the player shooting mechanic entirely, in favor of staying true to the "endless runner" game genre (since most endless runner games don't necessarily allow the player to destroy/kill enemies or obstacles anyway).

End Conditions

Staying true to the endless-runner video game genre, we implemented a player health system, in which our player has a base HP of 3, where if the player runs into an obstacle/enemy or is hit by an enemy projectile, they lose a single point of HP, indicated by the change in the player model color. When the player loses all health, they are redirected to a game-over splash screen, with a dynamic rendering of the "taken-over" game world.

Ethical Considerations

An ethical consideration we had is that there are a lot of "Endless Runner" games. It is a very common concept and generally simple. However the question can be raised if it is ethical to use someone's idea for a school project, such as mentioned on the Lecture slides about ethics. We try to mitigate these issues by aiming to respect previous renditions of this idea and focus on having a different concept for our map. Instead of using an infinitely long track, we have one arena that the player keeps revolving around. We also wanted to use a space themed setting rather than focusing specifically on the themes we saw in the game we gathered inspiration from 'Lite Bike Runner'. We also make sure to include any sources that we consulted in the making of this project. Another ethical guideline we thought about is taking precautions to respect user privacy. We do not ask for authorization to any part of users data, unlike for the common Endless Runner game format. Many of these games would traditionally require the user to authorize access to parts of their personal data that do not directly have anything to do with game functionality. To uphold respectful consideration, we used the WebAudioAPI which implicitly requires the user to trigger an action that allows for music to play. This feature used to not require a user to trigger an event for the music to play, but because of privacy concerns related to possibilities of code injection, this principle was removed. We took precautions that would guarantee the safety and respect of users privacy.

Conclusion

Overall, we are very proud of what we have been able to accomplish for this project. Although there were definitely tough moments where we couldn't figure out how to fix a bug for what felt like days, it was an enriching experience creating a 3D game with Three.js and learning how to use Cannon.js. Our final project surpasses our vision of the MVP by a longshot, with additional features that could be implemented in the future.

Follow-up steps

One of the additions we wanted to add was to have a set of different possible environments that the user could choose. Each of these environments can have their own specific obstacles and enemies that match the environment.We also think that adding more chances to customize the player model that the user chooses in terms of colors would also make it more entertaining for the user.

Team Member Contributions

Tuan Dinh: I implemented the logic to handle all objects, including static obstacles and enemies. Furthermore, I implemented collision handling with Cannon.js and projectile movement. I had also implemented a working player shooting feature which worked on my end but unfortunately is not ready for deployment (due to issues mentioned earlier). Finally, I found the 3D models for the player, enemies, and obstacles.

James Richardson: I created a very bare initial scaffolding with which we worked our way towards our final result, starting only with 2D mesh movement along circular paths (to hammer out how our automatic player movement mechanics could be structured). I also implemented the ThreeJS post-processing filters, integrated the holographic material throughout our models to maintain a uniform art style, handled our sound handling, and worked to maintain a (relatively) consistent code-base in our project throughout.

Laura Reyes: I helped begin our process in finding which 3D models would be best suited for our game. I looked into using Blender with ThreeJS to use a more complex vehicle for our game. Later on, we decided on choosing different 3D models that worked better with our game logic. I largely worked with James to look into how to implement our splash page and end page. I on the visual aspects of our game and attempted to create a cohesive theme. I also worked on fixing the third person camera angle to prevent it from rotating around the player to a camera view that followed the player model in closer proximity and felt more akin to a real video game camera angle.

References

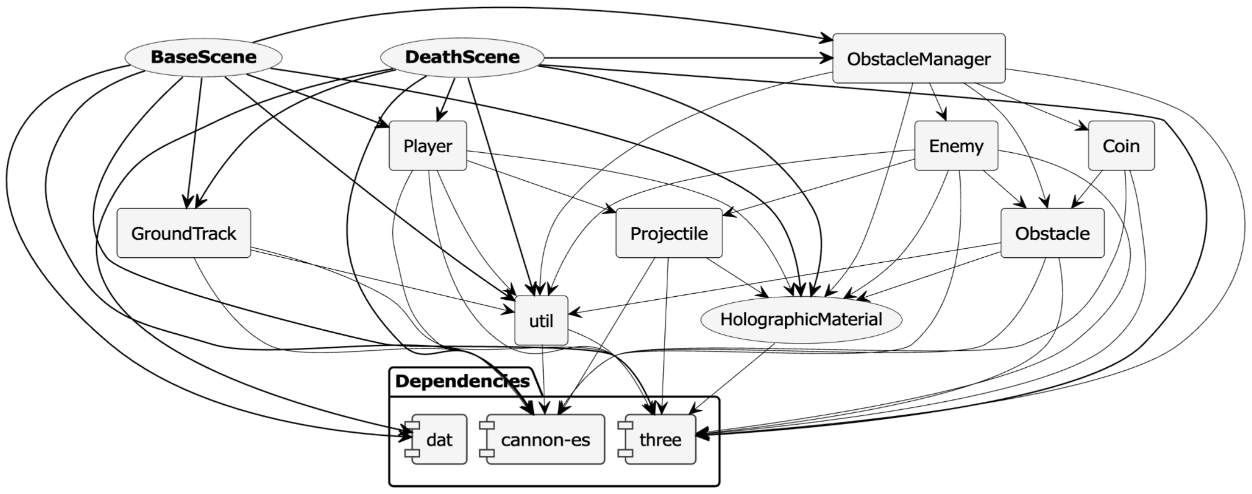
(1) Light Bike Runner: https://toucharcade.com/games/lightbike-runner
(2) ThreeJS Holographic Material: https://github.com/ektogamat/threejs-vanilla-holographic-material

Creative attributions

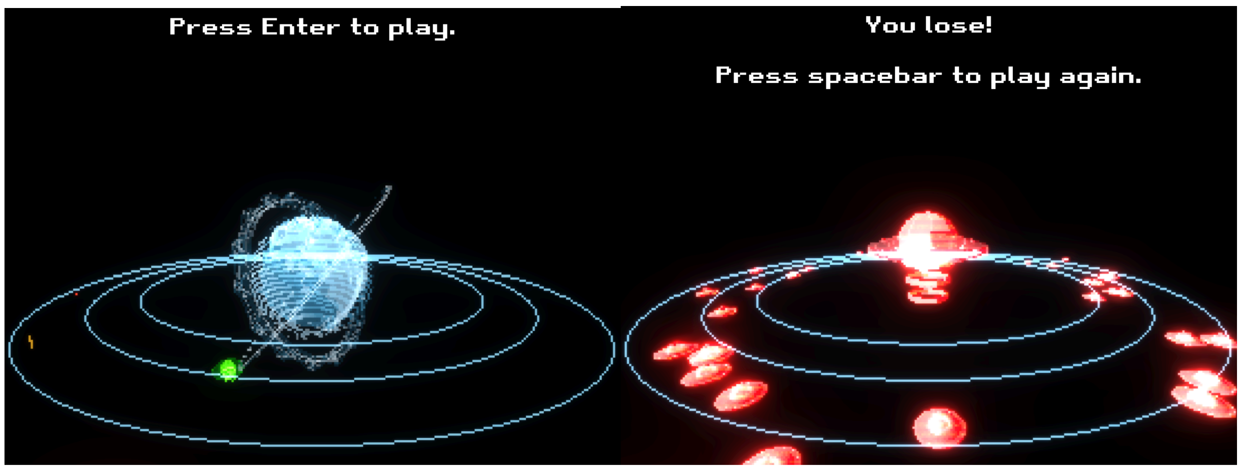
(1) All 3D models used were freely available and sourced from Sketchfab.
(2) Sound effects – excluding our background soundtrack – were all sourced from a freely-available asset pack.
(3) Our one background soundtrack – *Cyborg Ninjas* (Alex Macleod) — was sourced from uppbeat.io.

Appendix

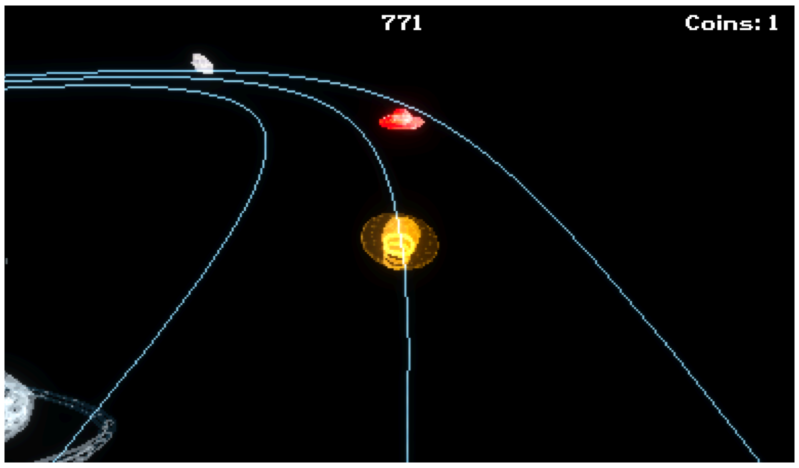
Game Object UML Hierarchy



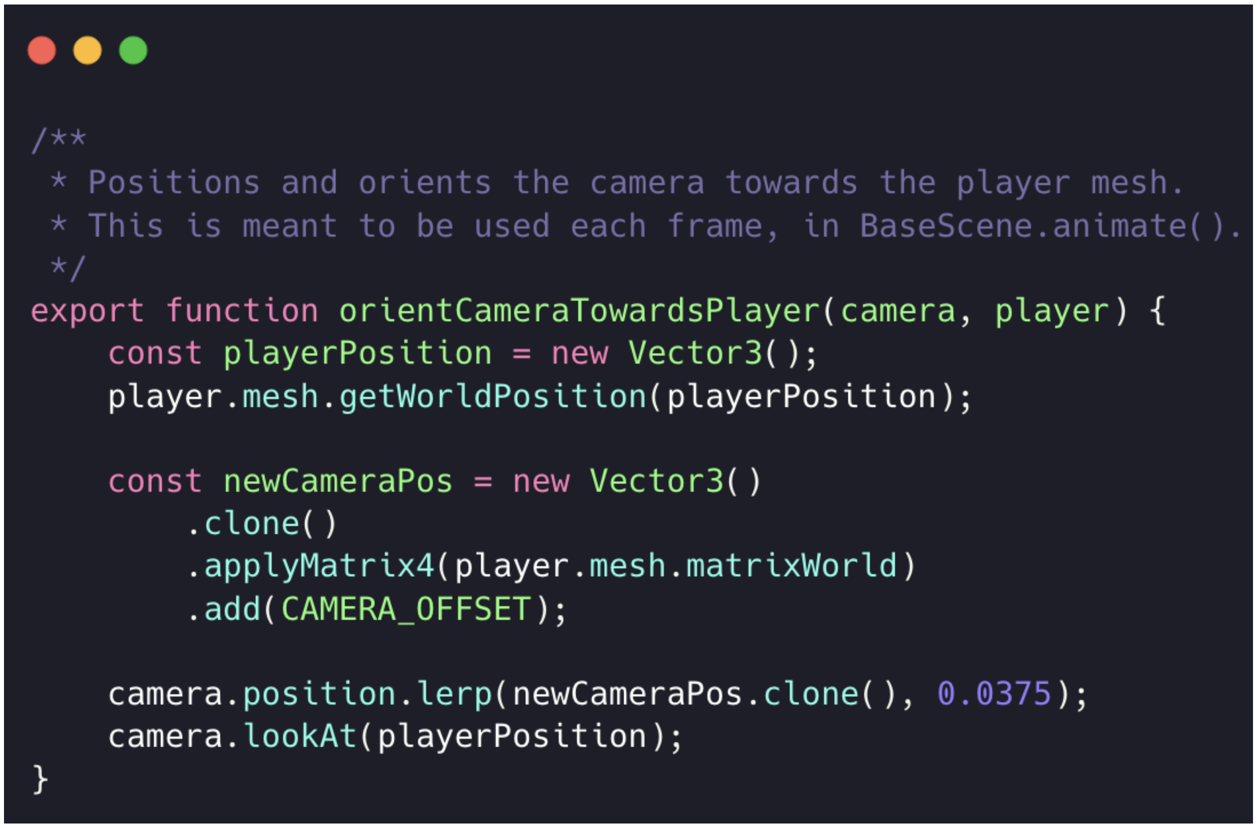
Static perspectives



Dynamic perspective

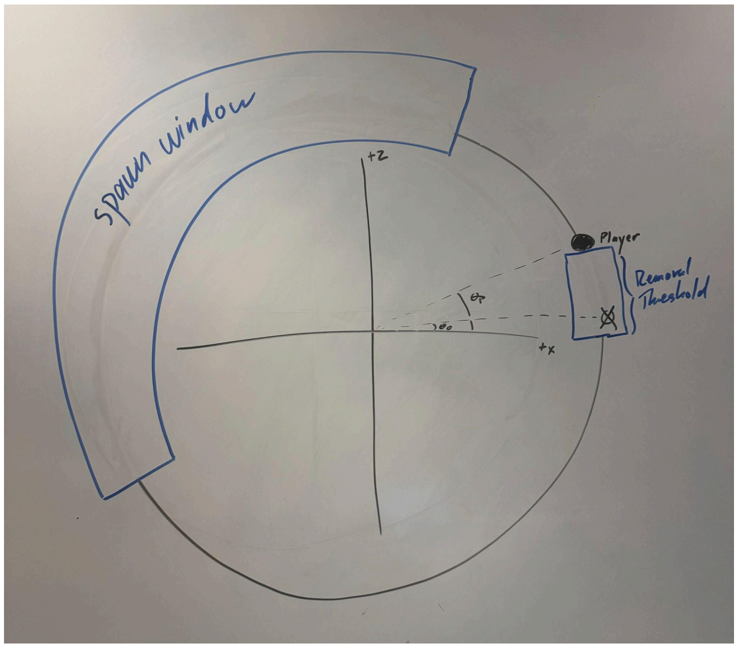


Dynamic perspective; implementation.

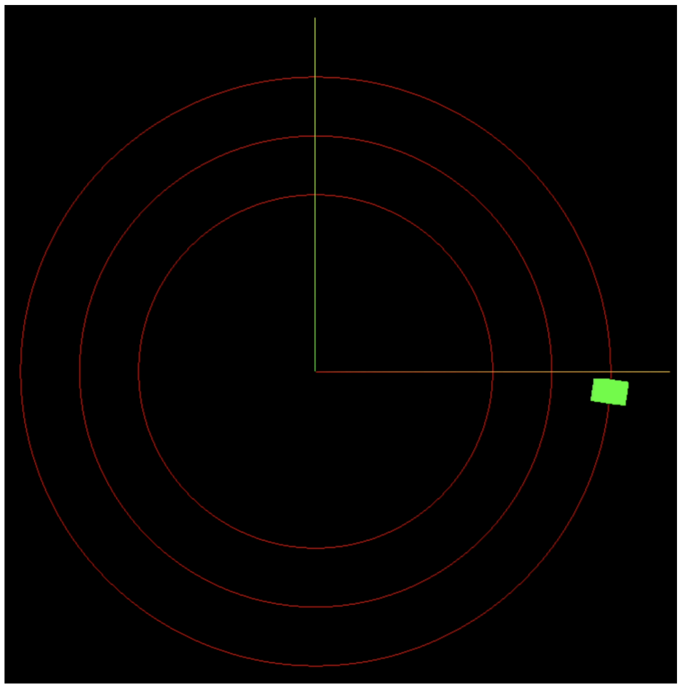


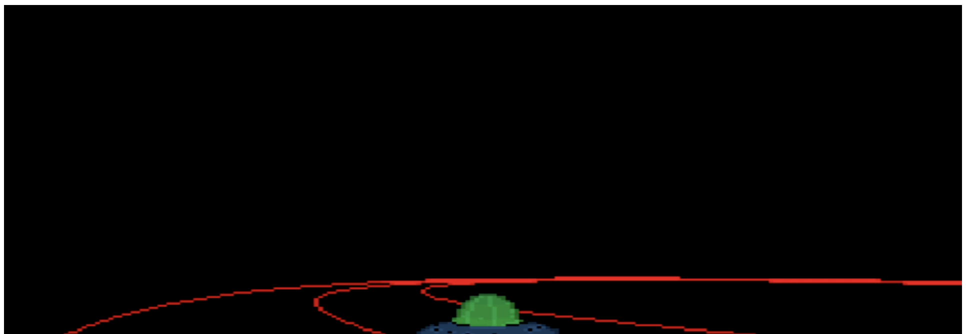
```
/**
 * Positions and orients the camera towards the player mesh.
 * This is meant to be used each frame, in BaseScene.animate().
 */
export function orientCameraTowardsPlayer(camera, player) {
    const playerPosition = new Vector3();
    player.mesh.getWorldPosition(playerPosition);

    const newCameraPos = new Vector3()
        .clone()
        .applyMatrix4(player.mesh.matrixWorld)
        .add(CAMERA_OFFSET);

    camera.position.lerp(newCameraPos.clone(), 0.0375);
    camera.lookAt(playerPosition);
}
```

Window Whiteboard Sketch



Prototyping





Shooting Demo Screenshot