Kareem Amin ID: 922543151

Github: ka-reem CSC317 Web Development

Assignment 3 - Javascript Individual Game Assignment

Description:
This assignment is to create a game in JavaScript. I created a remake of the game lightbike 1, which is inspired by the movie Tron. This is a 3d single-player game where you try to avoid the walls your tron bike creates while living as long as possible.

Rules:
Use “a/d” keys or left and right keys to avoid colliding with your own tron bikes’ walls or hitting the edge walls. There was supposed to be a npc to play against but due to time constraints, I wasn’t able to implement it.

Approach / What I Did:

I started this assignment without a strong approach; all I knew was that I wanted to create a 3D game. This was my first attempt at creating something in 3d, and I was uncertain if I could even accomplish it. During my research, I discovered a library called Three.js. Subsequently, a YouTube tutorial guided me through the process of creating a 3D plane.

Interestingly, this 3D plane reminded me of a game called "Lightbike," which I used to play on my iPod in elementary school. The game controls left and right movements and a boost for speed seemed relatively simple. I decided to implement the boost feature only if time allowed. Lacking confidence and an organized plan, the project progressed from creating the player, and defining movements, to establishing a pursuing wall and handling collisions. Though I had a general idea of my next steps, the majority of the project unfolded naturally as I navigated it step by step.

In hindsight, the absence of a structured approach became evident due to the interdependence of various project components. This realization dawned on me mid-project, making it more complicated as I hesitated to modify existing code. Consequently, I had to explore different methods to address this, resulting in numerous bugs and a less enjoyable experience.

Finally, due to inadequate planning, I faced challenges creating an NPC that could inherit methods from the normal player. This issue arose because I coded all the methods by passing in the player directly, instead of calling the method and passing in the player as a parameter. Due to this, I was not able to create a NPC.

Issues and Resolutions:

My initial challenge revolved around getting the 3D model to load properly in the game, and it turned into a bit of a time-consuming ordeal. I spent more time than expected on troubleshooting. The primary issues stemmed from the model not being in the right folder and possibly due to an incorrect file path. Another complication was the chosen file type not showing up as intended, leading me to re-download the files in a different format. After a couple of hours of grappling with these issues, I finally managed to get the render to show up in the game.

Another challenge arose when working on the camera turning. I aimed to implement a third-person camera that would smoothly follow the player's movements. Initially, it seemed straightforward to create a camera that turned at a 90º angle with the player. However, this approach presented a gameplay discrepancy, as the camera turned instantly and it became disorienting.

To address this, I delved into finding a solution for the camera to pan smoothly. Despite the various functions available for this effect, they ended up adding more confusion than clarity. After experimenting with various methods, I settled on one that eventually worked. This solution involved an involved set of mathematical operations, some of which I admit I may not fully grasp even now.

The process was divided into two rotations. The first rotation handled a simple left or right rotation. The second rotation, however, required copying the player's position using a vector—a representing the player's location with x-y-z coordinates. To position the camera slightly behind the player, I had to manipulate values within this vector. I opted for a somewhat straightforward approach by hard-coding positions for each of the four directions the player could face. While there might be more efficient ways to handle this, I found this method easy to comprehend, leading me to reuse it in various parts of the code.

Finally, I employed a "lerp" method, short for linear interpolation, to achieve a smooth transition during turning. Linear interpolation basically tries to find a center point between the two camera positions and smooths it out. Despite these efforts, a persistent bug remains: when facing 180º, the lerp method executes a 420º turn in the opposite direction, posing a challenge to gameplay. I identified the root cause of this and its a problem with the formula and I have yet to find a way to get around it. I presumed the issue is when it is turning from -pi/2 to pi since we are representing the angels in radians it was trying to turn 420º the other way because thats how you would find the center of a negative and positive number, however, as writing this I realized I could do 3pi/2 which is a positive number which should theoretically get rid of the issue but it does not mean the problem could be something else more deeply routed in the formula.

A significant portion of this project was consumed by grappling with painstaking bugs that, in hindsight, should have caught my attention earlier. One notable instance occurred during my pursuit of collision detection. While exploring various methods, I recalled that three.js had its

own built-in mechanisms for checking intersections. Although unsure if I would ultimately employ them, as they required casting objects into specific types.

The night before, I had started implementing a method named checkCollisions as the wall types I was using didn’t have their own collision methods without casting into different datatypes which I did not want to do. Unfortunately, I hadn't completed its implementation and forgot about it. In the course of my work, I encountered this method in my code, and copied and pasted it, assuming it was a built-in three.js function as recalling three.js has its own collision methods. Despite its obvious incompleteness, I attributed the lack of functionality to my code and not to the method itself.

This misjudgment led me to spend a considerable amount of time scrutinizing the method, attempting to print all the coordinates it was comparing, desperately trying to understand why it failed to detect a collision between the player and the wall. Faced with frustration and on the verge of giving up, I resorted to a final attempt to salvage the situation. I decided to look up the method to understand its workings, only to discover that it never existed in three.js.

This scenario was one of many numerous instances throughout the project where similar oversights led me into intricate debugging rabbit holes. In retrospect, debugging constituted a substantial portion of my project endeavor.

Analysis: (Reflect on what you have learned and how decisions made could impact users.)

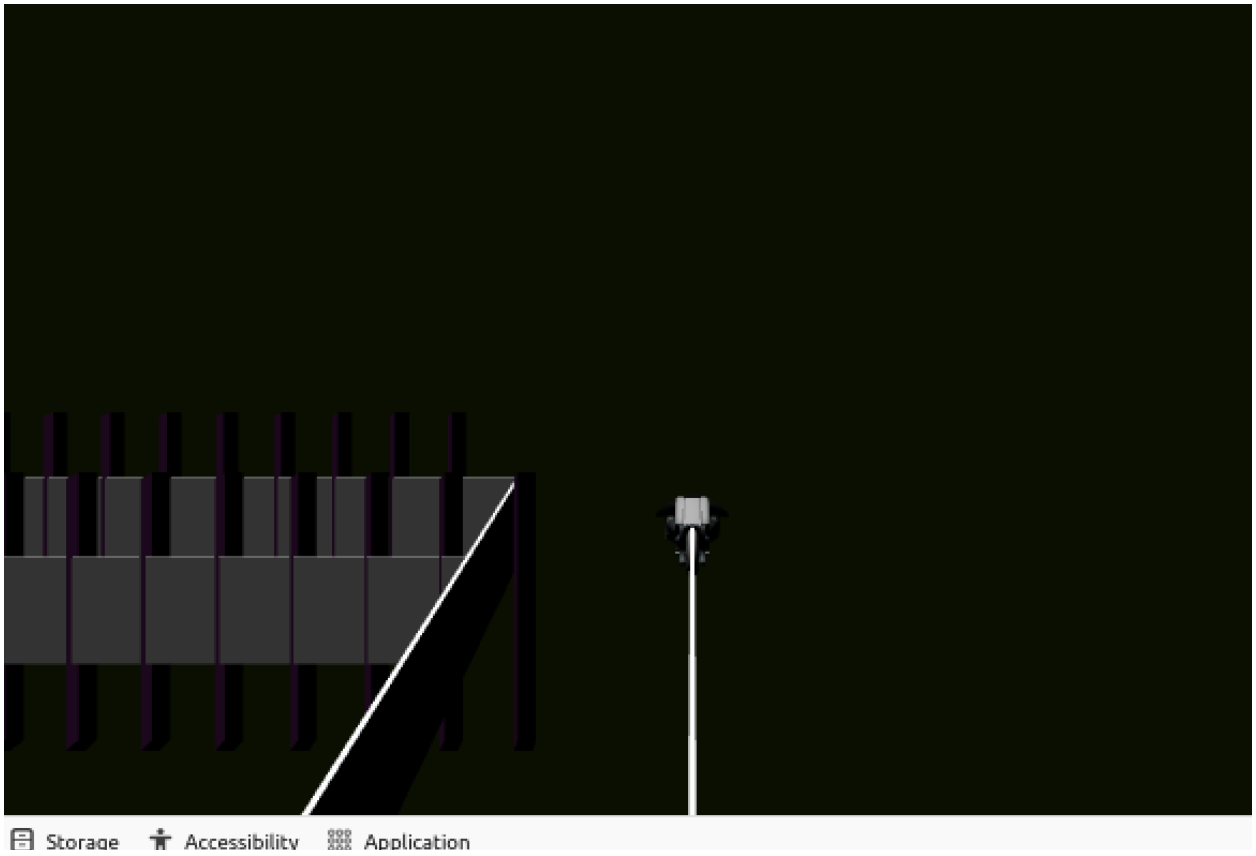
I refined my debugging approach by strategically leveraging console logs to identify issues within my code. In contrast to my previous method of commenting out sections until pinpointing the error, utilizing the console and logging mechanisms has proven instrumental in swiftly understanding and rectifying code discrepancies.

Venturing into game development, a domain unfamiliar to me beyond simpler projects like tic-tac-toe, brought forth new insights. I realized the necessity of anticipating and accommodating user actions comprehensively. In my specific game, players navigate through walls generated behind their bikes. Determining the optimal length for these walls became a critical decision, as too short would result in unsightly gaps, while excessively long walls posed the risk of game slowdown due to increased wall spawns.

Initially, finding the right wall length involved trial and error, adjusting incrementally to eliminate gaps. However, the introduction of a speed adjustment for the player revealed an unforeseen challenge the reemergence of wall gaps. This revelation underscored the game's dependency on frames per second (fps). Each player might experience the game at different fps rates, demanding an optimal wall length to cater to various computing capacities.

As a solo player at this stage, I determined an optimal wall length for my computer. Yet, this experience unveiled the necessity for game developers to address this variability for a broader audience. Balancing the game's performance across a spectrum of computers, from the fastest to the slowest, becomes a pivotal consideration. This might involve restricting access for users with slower computers, potentially sacrificing player base, or devising diverse solutions tailored to different computing capabilities. While the latter approach may demand more time and effort, it paves the way for a more inclusive gaming experience and a wider player community.

Screenshots:



One of the issues I mentioned earlier with collisions: The purple lines represent walls that spawn wherever the player is, symbolizing the player's hitbox. Each grey/white wall behind the player is stored in an array and checked against each purple wall when it spawns to detect collisions. The previous purple wall is deleted before the next one spawns. During debugging, I retained all purple walls to troubleshoot how to offset them accurately to follow the player. In the photo above, you can observe that the last purple wall extends too far. I had to make

adjustments to prevent collisions with the wall before reaching it or when positioned side by side.

GamePlay:
https://drive.google.com/file/d/1K7dqgUwWkjN4gi_53CgRTYkMs2UcZg2m/view?usp=drive_link