COS 426 Final Project Report

Friday, December 16, 2022

Kyu Han and Brendan Wang

Abstract

Our group created a game which we titled “Ambulance” which was based off of Crazy Taxi, an educational math game. Ambulance is an infinite runner game where a user drives an ambulance against incoming traffic for as long as possible without colliding into a car. Prior to implementation, we aspired to successfully implement a working MVP, namely to have an infinite runner, a main “character” and obstacles, and collision detection. During development, we used box-to-box collisions, sounds, texture mapping, and an on-screen control panel. Eventually, we not only successfully reached our MVP objectives, but also were able to finish some additional features, including procedural modeling.

Introduction

For our COS426 Final Project, our group aspired to create a game that was both interesting in concept and practical to complete within two weeks. Inspired by projects we saw in the Hall of Fame from previous years and examples, we were compelled by the idea of an infinite runner game. While the game concept was simple, there was a lot of potential for creativity and exploration. During this brainstorming process, Kyu proposed that we implement our own spin-off version of Crazy Taxi, an infinite runner game on Cool Math games, shown in **Figure 1** below. As the figure depicts, the user controls a car and avoids obstacles, which include other cars and other obstacles not shown in the figure (e.g., cones, rocks, etc.).

Figure 1: Crazy Taxi from Cool Math Games (image credit)

We adopt the general idea of a vehicle driving on a three-lane highway; the user being able to move between the lanes and jump; and the objective of avoiding obstacles. However, our game differs from Crazy Taxi in many ways. First, we insert our own flavor of the game by introducing a rather interesting story. The story is as follows.

Suppose it is the year 3020. Robots have taken over the Earth. The layers of the Earth are melting away due to a planetary explosion and whole towns and cities are floating above the lava. And there's only one robot, Melon Musk, that can provide a way to rescue the planet's inhabitants since Musk owns all the planet's spaceships. While other cars are fleeing to the nearest space shuttle to escape, you are notified that Melon Musk is malfunctioning and must get repaired immediately so that it can authorize the launching of spaceships for evacuation. While the road is infinite and a crash is inevitable, Musk will eventually be rescued. But as for how that happens, one must wait for the sequel: Ambulance 2…

In addition to the story, our game differs from Crazy Taxi in the following characteristics:

1. Instead of a taxi, we have an *ambulance*.
2. Instead of cars that are stationary, we have incoming traffic that is rushing against us at a *progressively faster* rate.
3. Instead of the surroundings being land spawned by cacti, our scene features dynamically generated pieces of land floating above a moving sea of lava.
4. Unlike Crazy Taxi, we allow the user to customize the difficulty or visual settings.

In terms of goals, we aspired to implement four advanced features: sound, collisions, texture mapping, and an on-screen control panel. Importantly, we implemented this game knowing that video game enthusiasts that enjoy games similar to Crazy Taxi would benefit from increased user customization. In addition, Crazy Taxi is fun but in many ways falls short in that there is no way for the user to customize the difficulty or visual settings. As such, in Ambulance, we wished for users to be able to customize the color of the sky, fog, floor, and terrain, as well as change the difficulty and the acceleration rate in the case that they wanted an easier or harder challenge and/or simply wanted to change the visuals of the game.

Methodology

We implemented Ambulance using THREE.js, external GLTF and image files, and sound resources. We start with high level details. To implement Ambulance, we needed to handle the creation of three main components: a scene, objects, and user interactions. We briefly overview each component as follows:

1. **Scene**: the scene setup was implemented in the Scene folder. Here, we implemented a function that allowed us to create a finite 2D plane that was orthogonal to the vertical axis. In addition, in the main constructor, we also populate the scene with the objects of our game (detailed in the Objects section below). The scene also includes fog.
2. **Objects**: the objects setup was implemented in the objects folder. Each with their own separate subfolder, the objects include Ambulance, Car, Lines, and Road which are for the ambulance, cars, road lines, and the road respectively.

3. **User Interactions**: the user interactions were implemented in App.js in the form of event listeners and handlers. In particular, when a user presses the appropriate key (e.g., space or left/right), we trigger the appropriate action as a response (e.g., jump, move).

With the high-level overview of our project components defined, we now highlight in detail the steps we took to implement the five advanced features we implemented. They include Collision Detection, Textures, Sound, On-Screen Control Panel, and Procedural Modeling.

Collision Detection: Collision detection in our game involved checking whether or not two vehicles touched each other (see example in **Figure 2**). In our game, a box-to-box collision check sufficed. Thus, we implemented a function that took in two car objects, constructed a bounding box around each, and performed a series of checks. More formally, let C_1 and C_2 be Car 1 and Car 2 and let $C_n(a)$ be the a^{th} coordinate of Car n. Then, the function checks the following cases:

- If $C_1(x)$!= $C_2(x)$, the cars do not have the same x position and they are not in the same lane. Thus, they cannot collide and we return false.
- If the $\min(C_1(z)) <= \min(C_2(z)) <= \max(C_1(z))$ and $\min(C_1(y)) <= \max(C_2(y))$, this means the cars are in the same lane and overlap both in the direction of the road and the direction of the sky. In particular, we have a front-to-front collision and return true.
- If the $\min(C_1(z)) <= \max(C_2(z)) <= \max(C_1(z))$ and $\min(C_1(y)) <= \max(C_2(y))$, this means the cars are in the same lane and overlap both in the direction of the road and the direction of the sky. In particular, we have a back-to-front collision and return true.

At each frame, we use collision detection to see if the ambulance has collided with any of the other cars. If so, the game terminates.

Figure 2: Collision between Ambulance and Car

Textures: A concrete and ocean-like image were added as textures to the road and the floor respectively. We added these to the road and floor meshes using the texture loader from the THREE.js library. Both textures are shown in **Figure 3**.

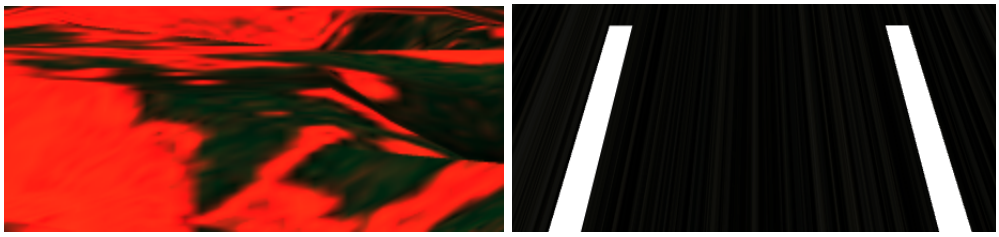

Figure 3: Lava texture (left) and Road texture (right)

Sound: We attached several sounds to particular events. They are as follows:

1. **Jump**: whenever a user jumps by clicking the space button (see below for implementation details on jumping), we emit a jump sound.
2. **Whoosh**: whenever a user moves left or right by clicking the left or right arrow keys, we emit a whoosh sound.
3. **Collision**: a boom noise is generated whenever a collision is detected.
4. **Background**: background music and ambulance sound effect is played continuously until a collision causes the game to terminate.

On-Screen Control Panel: To create the GUI, we created a state object which stored the default values for the control features. The state variables include the difficulty level, the rate at which acceleration for jumping increases, and colors for the sky, fog, and road and are categorized as either Gameplay or Color features using the GUI folder property.

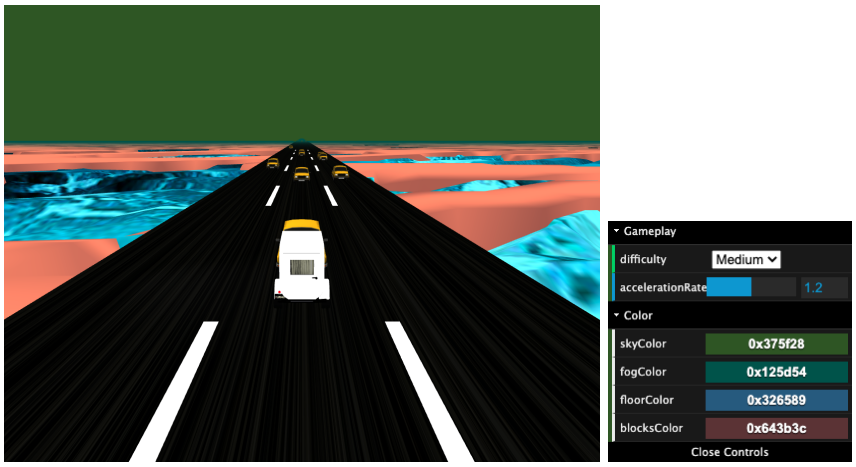

Figure 4: On-Screen Control Panel (right) and corresponding changes (left). Note that the colors chosen are not the default colors.

After defining the state, the state of the GUI is then checked every frame to see if the user changed the state. If the state changes, the feature is then immediately reflected since the state variables are directly attached to the variable or property they modify. We show an example in **Figure 4** above in which we change every color option to a color that is *not* the default.

Procedural Modeling: Procedural modeling was crucial for generating infinite terrain that was both random and dynamic. While we thought having a static background terrain would suffice, we wished to create a more compelling and engaging experience for the user and saw procedurally generating terrain as a great way to achieve this goal. To implement procedural generation, we used a Perlin noise function that took two dimensions as input and generated a value between -1 and 1 as output. We chose Perlin noise over a random number generator because the former achieves a more smooth aesthetic effect than the latter and we hoped for the terrain to look somewhat realistic. An example is shown in **Figure 5**.

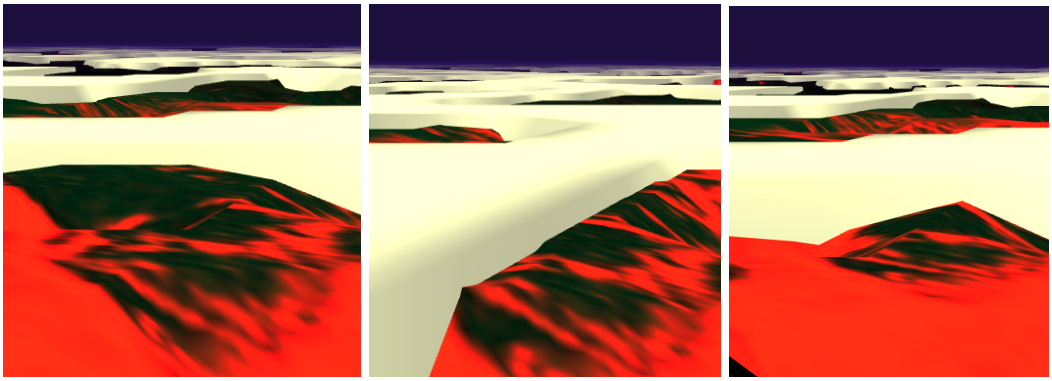

Figure 5: The terrain features procedurally generated blocks of land floating over moving lava. The panels are ordered consecutively and taken at different frames.

Our approach was as follows. We have two terrains, one to the left and one to the right of the main road, both of which are constructed as a grid of triangular meshes. First, for each (x, z) on either terrain, we input the x and z values into the Perlin noise function to get the y value, namely the height of the terrain at the (x, z) point. This has the effect of generating terrain with different heights at different spots in the (x, z) plane. Since the range of the Perlin noise is between -1 and 1, we took the maximum of 0 and the Perlin noise. To generate the y-values for each of the (x, z) points, we increment by some x-offset and z-offset when we move in the x and z directions, respectively. To create the illusion that the terrain was moving past us as we are driving past it, we introduced a variable *flying* that would determine where in the z-direction we started when generating a new terrain again. Once every ten frames, we would increase the value of *flying* to make the next iteration of the terrain seem some distance closer. After the player crashes, we stop updating this variable in order to prevent the terrain from "moving" any further.

In addition to these core advanced features, proper *movement* of objects was also essential to our game. First, we needed to render the scene such that the road appeared to be moving towards the user playing the game. To accomplish this, we made two sets of lines on the road, one "leading" and the other "lagging". The former would appear in view before the latter. Both sets of lines would move towards the user until they reach past a certain z-value threshold. As soon as this threshold is reached, the set of lines is repositioned beyond the horizon. By alternating the set of lines in view, this allowed us to achieve the infinite runner effect that we aimed for.

Beyond the infinite runner, we also needed for the car to jump smoothly in the air. To accomplish this, we defined a velocity and acceleration variable. As soon as we detect that the car is in the air, we set the car's velocity in the y-direction as some positive value (e.g., 50) and perform the following steps for each frame:

1. Add this y-velocity to the current position of the ambulance.
2. Subtract the y-velocity by some acceleration factor.
3. Multiply the acceleration factor by some acceleration rate.
4. Move to the next frame and repeat steps 1-3.

Once the y-position of the ambulance reaches the road's y-value, we set the velocity to 0. Importantly, this acceleration rate above is a parameter that the user can adjust in the GUI. Lower values of the acceleration rate will allow you to jump higher and longer relative to higher values.

Finally, we needed the cars to appear continuously on the horizon so they would be coming towards us *and* make the cars progressively faster to make the game fairly challenging. To implement the movement towards the user, we respawned the *same* car objects on the horizon once they reach a certain z-threshold and reposition them on the horizon. Respawning the same objects instead of generating new ones was crucial for good memory performance. This was repeated until we crashed into a car, ending the game. In addition, for the increase in speed, we created a speed variable *s*, moved the cars at a rate proportional to *s*, and increased *s* by some value each frame.

For other possible implementations, we were considering allowing the ambulance to move forward and/or moving the camera position forward. While these changes are worth considering, they also come with many caveats, including the need for more tedious calculations without necessarily adding too much benefit to the user experience. Additionally, we did not implement the feature of spawning new coins that served a "boost" to the score nor did we keep track of score. In the current version we have, we felt that the objective of the game was to save the robot in the ambulance and felt that the existence of coins might take away from that mission. More details are discussed in the Discussion and Conclusion section.

Results

For the initial MVP of Ambulance, success meant that the user could play on the infinite runner until they crash into some obstacles. The aesthetics in this version were not of high priority. In fact, instead of loading the GLTF model, we started by implementing cubes for the cars.

However, in our second version after we implemented our MVP, our definition of success expanded. In particular, we needed the game to be performant, adequate, and aesthetically pleasing. To test performance, we made sure that we didn't create unnecessary meshes or print

statements which would slow down our game. For adequacy, we made sure that our game was not too easy nor too difficult, and that it was possible for users to move in the way they expect. In addition, we ensured that the movement of the ambulance was responsive and the collisions were accurate (i.e., we did not want to penalize the player and rule that there was a collision when there shouldn't have been). Finally, to address having good design in our game, we added moving terrain, made the background more colorful, and inserted cars and an ambulance.

After playing the game and testing it in as many different scenarios and edge cases as possible, we determined that our game is overall fairly fun and challenging. We also determined that our collisions are fair and accurate. Lastly, we observed that the performance of the game was great.

Discussion and Conclusion

Overall, this project was very enriching and enjoyable. We were able to directly put many of the concepts we learned over the course of this semester into practice, including collision detection and procedural generation. We also learned more about Perlin noise and how games involve continuously updating and rendering a scene. We believe the approach we took of creating the MVP first before adding additional embellishments and features was also successful since it allowed us to initially focus on the essentials. In addition, we avoided the need for excessive calculation and were able to create the illusion of driving forward by moving the objects towards the ambulance instead of moving the ambulance forwards. Finally, we were able to not only meet all four of the advanced features we intended to implement but also implemented procedural terrain generation.

That said, we have a few additional items that if given more time, we would have considered implementing as follow up work. They include:

1. Making the left/right movements continuous and to be able to occupy any x position in the highway rather than the three different x-positions. That said, we believed that this could possibly make the game less fun unless there were other components to the game, such as possibly having a race with multiplayer.
2. Having some scoreboard that kept track of the users score (e.g., how much time they have currently "survived in the game").

Contributions

Kyu contributed to the detection of box-to-box collisions, imported GLTF files and helped make the procedural generation of the terrain using the Perlin noise update.

Brendan implemented the jumping logic, road movement, and sounds. In addition, he created the on-screen control panel. He also helped create the procedurally generated terrain.

Works Cited

Ambulance.js

Background for jumping logic in Ambulance.js was taken from:
https://discourse.threejs.org/t/three-js-simple-jump/40411

The code for loading GLTF files was adapted from the following (we also got the idea to use the links to our own raw Github files in the GLTF files to load textures from this):
https://github.com/harveyw24/Glider

GLTF model for Ambulance:
https://sketchfab.com/3d-models/model-a-ambulance-a1c2373a25654117a731a1861403f372

Car.js

GLTF model for Car:
https://sketchfab.com/3d-models/lowpoly-car-pack-94fcef58d8d04af2b6bf42f2949227eb

Road.js

Texture for the road was taken from:
https://wallpaperaccess.com/full/4951395.jpg

SeedScene.js

Texture for the floor was taken from:
https://img.besthqwallpapers.com/Uploads/19-3-2020/125338/water-background-waves-ocean-water-texture-ocean-aero-view.jpg

Perlin.js

Code in this file is **entirely** credited to banksean (Github):
https://gist.github.com/banksean/304522#file-perlin-noise-simplex-js

The Perlin noise code above was used by the following project (we are citing to show that this is how we found out about the source):
http://www.stephanbaker.com/post/perlinterrain/
https://github.com/stephanbaker/PerlinTerrain

App.js

Code for updateGround and updateWater was copied/adapted from:
https://www.youtube.com/watch?v=IKB1hWWedMk
https://woodenraft.games/blog/generating-terrain-plane-geometry-three-js

We adapt various parts of the code below for the moveRoadLine function:
https://jsfiddle.net/prisoner849/hg90shov/

The code for the jumping logic in moveCarInAir and the handleMoveAmbulance was adapted from:
https://discourse.threejs.org/t/three-js-simple-jump/40411

Utils.js

The code for generating a random integer between a min and max value was directly copied from: https://stackoverflow.com/questions/18921134/math-random-numbers-between-50-and-80

Setup.js

Context on how to insert sounds and code for inserting sounds was adapted from:
https://www.youtube.com/watch?v=91sjdKmqxdE

Sound sources:

1. jump: https://pixabay.com/sound-effects/cartoon-jump-6462/
2. ambulance: https://pixabay.com/sound-effects/search/ambulance/
3. collision: https://pixabay.com/sound-effects/clank-car-crash-collision-6206/
4. whoosh: https://pixabay.com/sound-effects/whoosh-6316/
5. background2: https://pixabay.com/music/synthwave-neon-gaming-128925/

Code for setting up the GUI state in createGUIState was adapted from:
https://codepen.io/justgooddesign/pen/ngKJQx

General

The logic for rotating the plane about the x-axis was adapted from scene.js in Assignment 5.