COS 426 Final Project Written Report: TIGER ROAD

Malinda Huang, Angelo Gonzalez, Brian Max Tchouambe
December 16, 2022

Figure 1. The starting page of the Tiger Road game.

Abstract

Since the widespread adoption of mobile gaming in 2008, there has been a surge in the research and development of lightweight mobile games. Although popularized as far back as the 1997's with simple Nokia games, due to the rising popularity of Apple and the centralization of Apps in the app store, mobile gaming has become a mainstream that is more than here to stay. Among the thousands of mobile games released per year, many have become popular in recent times as more and more idle moments. In this project, we chose to recreate a popular game Crossy Road.

This report details the planning, design, and implementation of the COS 426 project Tiger Road from start to completion. It examines the features implemented during the course of the project, the roles played by individuals in the creation of the game, and the challenges/obstacles encountered by this group during the development of the project. This report also provides a comprehensive insight into the methodologies chosen by our team to implement this game and our findings/ results.

The code of the game is available at https://github.com/MalindaH/cos426-project with a demo at https://malindah.github.io/cos426-project/.

1. Introduction

1.1. Goal

The goal of this project is to test the skills we learned in COS 426, by seeing how faithfully we can recreate the popular mobile game Crossy Road, using Javascript for a web browser. Furthermore, we want to push the limits of our creative capabilities by adding our own personal flavor to the design aesthetics of the game, to make it more familiar to us. Our goal is to see if we can redesign the game in a style that is more reminiscent of Princeton, by incorporating features such as the Tiger Transit bus, Psafe vehicles, rowing boats, etc, using the design techniques we learned over the course of the semester. We, as developers, have benefited tremendously from learning more about the inherent intricacies of developing games in Three.js. Moreover, anyone who is a fan of the popular mobile game Crossy Road can simply click on their browser and enjoy this game from the convenience of their home. It expands to a broader audience, one that does not have access to an expensive mobile phone. We aim to see if we can replicate the game with greater efficiency in a browser, and possibly revamp certain features in the original game into our interface.

1.2 Previous Work

Previous work has been concentrated on developing a game in which the user plays a character who navigates a seemingly infinite number of stationary and moving obstacles. Examples include Crossy Road, Temple Run, Subway Surfers, and many more. The challenge of pushing one's avatar to reach the highest goal is what primarily propels users to continue to play time after time. Previous approaches that have failed do not consider what might discourage users from seeking to push themselves. A rapidly scaling increase in game difficulty might deter newer users from playing as their score progresses very slowly. In addition, game incentives such as skins, ranking systems, or even in-game boosters give users a boost in motivation to reach certain smaller checkpoints within the larger aim of increasing their score. Other approaches also did not implement the full functionality of the original game including the river crossing and the timeout eagle that kills the character when it stays in one place for too long.

1.3 Approach

For this project, as explained above, our aim is to replicate the popular mobile game Crossy Road. The approach is to build up multiple rows of the map called "strips", which are combined together in a chain to form a grid map that grows longer and longer. New strips are populated and attached to the map as the player jumps forward, and old strips that are not visible in the camera view are removed to optimize the rendering. Each strip has its own obstacles depending on its type and it would be up to the player to navigate through as many strips as possible, monotonically increasing the score. We choose to limit the increase in game difficulty and instead offer several modes with different relative speeds in order to not deter newer users from playing the game. The game should work well under any and all circumstances as we have considered several aspects pertaining to speed, including randomization, ensuring the player can move at least as fast as the other vehicles. All functionalities including the river crossing attachment, …, have been tested thoroughly to make sure the game is hopefully bug-free.

2. Methodology

2.1 Game Features

The game is implemented in JavaScript using the three.js library. In order to fully realize the game design, the following main features are required:

- Setting up the world
- Character movement control
- Building an infinite landscape with which the players will be traversing.
- Collision detection
- Sound effects to bring more life to the animation
- Optimizing the game: view frustum culling and deleting unseen objects
- Multiple camera views
- Texts, score, and game-over animation

The additional features that are implemented in this game include collision detection, texture mapping, view frustum culling, sound effects, procedural modeling, vertex and fragment shaders, and multiple views.

Figure 2. A screenshot of the game. A first-person view is on the lower right corner.

2.2 Setting up the world

An orthographic camera from three.js is used. The lighting system includes a main directional light that shines at the ground from the top left direction of the character and a supplementary directional light from another direction, an ambient light, and a hemisphere light. Shadow is implemented using the main directional light's shadow camera. The shadow camera's size is kept just a bit bigger than the camera view to ensure fast rendering while preventing the shadows from being too pixelated.

2.3 Character movement control

A character object is implemented and added to the scene. Event handlers for the w/a/s/d keys are implemented to initiate character jumping in the front/left/back/right directions,

corresponding to +z, +x, -z, and -x directions in the world. The parabolic jumping animation is implemented by using a sine function to calculate the y position of the character based on how much it has moved in the x or z direction. The character lowers a bit while turning to the direction corresponding to the pressed key before possibly jumping to indicate that the key press has been detected even though a jump may not be allowed. Long presses of keys are ignored to prevent buggy jumping behavior. Multiple key presses are stored in a queue and a jump is only initiated when the previous jumping animation finishes.

The camera is programmed to move forward at a constant speed so that the player dies if the character lags outside the camera view. The camera smoothly moves with the character as it jumps around, and the main directional light moves with the camera

2.4 Building the landscape

2.4.1 The map

The map in the game is implemented as a 2D grid in the x-z direction. Each grid has a size of 2x2 and the character is only allowed to move 5 grids to the left and 5 grids to the right in the x direction. The landscape is built 5 grids bigger than the movable grid in the x direction to ensure that the camera view is filled with the landscape while keeping the rendered map as small as possible to keep a high rendering efficiency. This 2D grid is represented by a 2D floor type array to store the floor type of that grid which is needed for collision detection. Different flags are set when the player shares the same x and z coordinates as the floor: the player dies when it is on water and not on a log.

2.4.2 Types of strips

For the landscape, we have to implement three types of strips to simulate the various environments that the avatar has to navigate within the game: river, road, and forest. To make the game seemingly endless, a new strip is added to the world when the character moves forward, and old strips that go outside the camera view are removed.

For the river strip, the river is implemented as a long cube. Each river contains 3 boats that are spawned automatically in random positions along the river. The animation of the river moves its boats along itself. Each river is given a speed and a direction when it is instantiated. When a boat moves in a direction and it reaches the edge it is automatically moved to the opposite edge by translating its position, then the boats continue to move along the correct direction. Texture mapping is added to the water cube using a water texture image.

For the road strip, the floor is implemented as beige cubes. Three types of car objects are implemented: golf cart, TigerTransit bus, and Psafe patrol car, each corresponding to a specific speed. A random type of vehicle is spawned on each road strip at random x positions with a random direction (left or right). When a car reaches one end of the strip, it is then moved to the other end to simulate more cars coming continuously.

For the forest strip, the grass is implemented as green cubes. Trees are implemented as an object and upon initialization, they gain a random size within a range and a direction. Trees of random sizes and directions are populated to fill all the unmovable grids and some random positions of the movable grid. The existence of trees is indicated in the floor type array to disallow character from moving to grids that are occupied by trees.

2.4.3 glTF assets

glTF assets of the objects including the mallard character, the golf cart, the bus, the police car, and the tree are downloaded from SketchFab (https://sketchfab.com/). For the assets, although we have a huge selection to choose from, our aim is to make them as Princetonian as possible, only using assets that resemble Princeton symbols such as the TigerTransit bus and the Psafe patrol car. Bus and patrol car assets are downloaded from sketchfab and exported into Blender for editing the color and adding the TigerTransit and Psafe logos. The meshes are also decimated to reduce the number of faces and vertices so that they can be rendered faster in the game. They are then exported as glTF files and loaded onto the corresponding objects in the game. These assets give the game a Princeton feel, as we edit the assets to be as close to the Princeton symbols as possible. Assets like trees and golf carts are meticulously chosen to be as close as possible to features from the surrounding Princeton neighborhood.

Figure 3. Modifying a bus asset obtained from SketchFab (https://sketchfab.com/) in Blender to make it look like TigerTransit.

2.4 Collision detection

Collision detection is implemented by generating a bounding box for the character and the obstacles, including the different vehicles and the logs, using Box3 and Box3Helper from three.js. Different flags are set when the player collides with the different types of objects: the player dies when it collides with a car, moves together with the log when it collides with a log, and dies when it is on water but not colliding with a log.

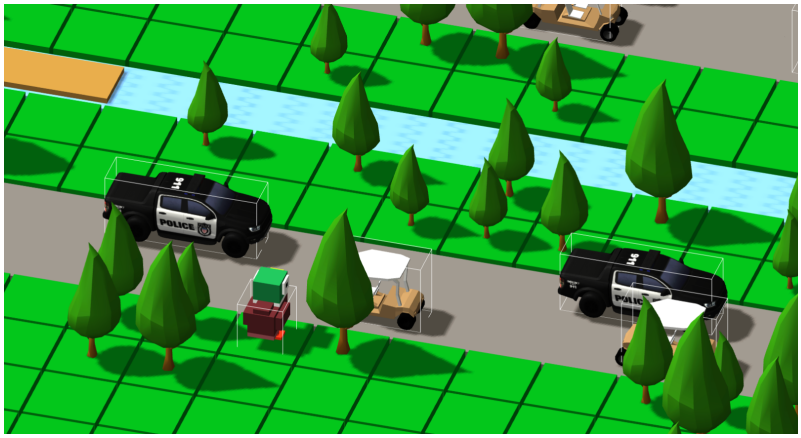

Figure 4. The collision detection system using bounding boxes.

2.5 Sound effects

Sound effects are implemented using the three.js AudioListener and AudioLoader to load and play the sound. The sound effects are obtained from the FreeSound website (https://freesound.org/) and edited in Adobe Audition to get the desired effect. Currently, we only have implemented the jumping sound effect, but background music, sound effects when different types of cars pass by, and sound effects when the character falls into the water have already been obtained and edited and will be added to the game.

2.6 Optimizing the game

To make the game render at a higher FPS, view frustum culling is implemented by checking whether each object is within the camera's view frustum in each frame. The camera's view frustum is first obtained from its projection matrix and matrix world inverse. For all objects with bounding boxes, the intersection of their bounding boxes with the view frustum is checked; for objects without bounding boxes, which are simple cubes indicating the grass and road, the intersection of themselves with the view frustum is checked. The objects that don't intersect with the view frustum are not rendered in this frame.

2.7 Multiple camera views

A second first-person camera view is implemented by adding a second orthographic camera that looks at the character very closely from its -z direction and rendering both cameras in the render loop of the app. However, rendering a second view requires rendering twice as many objects as a single-view game, which causes the FPS to significantly drop, so we decide to keep this game as a single view to ensure the performance.

2.8 Texts, score, and game-over animation

Game start and game over texts are implemented as HTML elements with CSS styles and they are animated to go away and appear when the game starts or when the character dies. A score at the upper left corner starts to count as soon as the player starts to play the game. One score is added when the character moves forward to a new row.

The death animation is implemented so that the character is programmed to get squashed in the x or z direction depending on which side it gets hit by the car. A particle system appears

when the character dies, which uses vertex and fragment shaders to render each particle. The particle system consists of square particles of random sizes and colors that are generated in random x and z positions near the dead character. The particles have a speed in the +y direction to float upwards and disappear after a set lifetime.

After the game is over, the game returns to the start screen when any key is pressed.

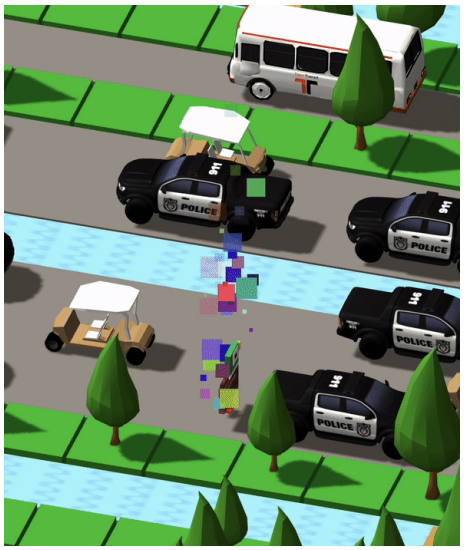

Figure 5. The particle system used in the death animation.

Results

The goal of this project is to see how well we can recreate the original Crossy Road game. By this standard, we choose to measure the success of the game by closeness in similarity and function to the original game, the performance of the game in the web browser, and by how well we can manipulate the speed and player statistics to ensure a balanced game. By this we mean, a sense of fairness in the game is achieved so as to not necessarily leave the player at a disadvantage, i.e cars not being too fast as compared to the player's movement or the player not glitching out unnecessarily. In this project, we experiment how well-paced the game feels by having multiple peers test the game. We gauged their level of satisfaction by asking questions about their experience while playing the game. The majority of feedback was typically positive with most players citing how nostalgic they felt during the game. This indicates to us that we did a successful job of recreating the original game structure. However, there were some players who commented on the fact that certain aspects of the game were not balanced, such as the pacing of the river boards compared to the speed of the vehicles on the road and the player. This tells us that there is more work to be done in finding a balance between all of the moving parts of the game in order to provide a better experience. We also test the FPS in order to gauge the performance of the project. While removing a second view window helps the performance greatly, unfortunately, some users find the character object too slow based on the computers they are using. As they progress forward and new strips are mapped they also say the performance of the game begins to fall. This indicates to us that a more efficient method of spawning and recycling objects in the scene is needed.

Discussion and Conclusion

Our group accomplished our objective of recreating the original Crossy Road game as not only did we implement all the core features of the original game, but we were able to add our own elements to the game. The approach showed great promise overall and aside from minor improvements we achieved most if not all of our goals. In order to further improve our game we plan to tweak the game statistics to “balance our game”. We also aim to add extra features such as implementing a real physics engine, a better texture and mesh for the boat logs, and to remove jitters caused by event handling. We also want to work on improving our FPS. This is a crucial feature that is required to provide the best user experience. In order to improve this, we aim to work on the recycling of scene objects and the spawning of strips. We believe spawning many smaller objects individually is less effective than spawning a large object with many children objects. This is a possible cause for the reduced FPS in our game. We also want to implement extra features that we did not have time to. In the original Crossy Road game, there is a bird that comes and gets you if you stay in a spot too long. This would be great to implement. Better drowning animation could also be implemented to bring the game to life.

Contributions

All three teammates brainstormed together to come up with the game design. All three teammates worked on preparing the presentations. Malinda implemented setting up the camera and lights, character movement control and animation, view frustum culling, the forest strip, editing and loading glTF assets, editing sound effect files, texts and score tracker, death animation, and the particle system. Angelo implemented collision detection, keeping track of floor types, vehicle spawning and culling, and loading the sound effects. Max implemented the water floor type and spawning the boat log objects. Max and Malinda worked on writing the report.

Works Cited

The original game: Crossy Road
Three.js
Blender
OpenGL Shading Language (GLSL)
glTF assets from SketchFab: https://sketchfab.com/
Sound effects from https://freesound.org/