Stay On Track

COS 426 Final Project by Giao Vu Dinh, Yuanqiao Lin, and Yasmine Calvo

---

Abstract

Inspired by mobile games such as ZigZag and 跳一跳 (Jump), Stay on Track is an endless runner game where the user controls a ball that must stay on a continuously generated zig-zag pattern track. Our game incorporates features from both of these platformer games, notably changing directions, jumping, and picking up collectibles. Users can upload their own .mp3 audio files that will be used by the game to procedurally generate the track, and players must jump, turn, and collect gems to earn the highest score possible.

Introduction

Goal

Our primary goal for the project was to create an endless runner game that would provide an enjoyable and relaxing experience for the player. We were also inspired by rhythm games and past projects that have incorporated or visualized audio to make the tracks more individualized for the player. Our MVP plan was to implement a game that had all of the basic features we wanted: gameplay logic, player movement and physics, track and object collision, and incorporating user input. Additionally, we wanted to incorporate audio into our game by allowing users to upload a file (or providing a default track) and interpreting these tracks to procedurally generate the path for the player to move on. At the time of writing, we have successfully achieved all of these MVP core features. Stretch goals for our project included adding more visual elements, such as a audio visualizer or custom models for the track, player, and background.

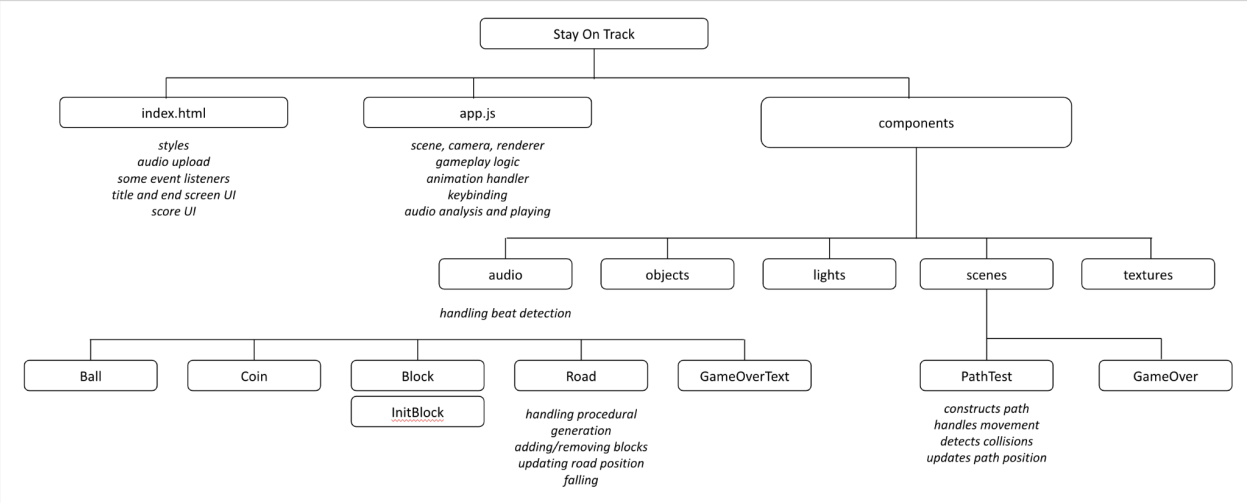
Previous Work

Our inspiration for the path and endless runner mechanic was from existing mobile games such as ZigZag and Jump. Inspiration for the audio detection and visualization part of the project came from past projects developed for this source such as mpThree.js and VSKeys.

The two main features that we wanted to incorporate and blend in our project were the endless runner mechanic and audio visualization. We felt that the endless runner type of game was a well established and effective gameplay mode that would keep players engaged, and the audio visualization in real time mechanic would add to the replayability and uniqueness of our project. While we have seen other games generate visual effects in real time based on audio, or have predetermined levels set to a particular track like in rhythm games, we wanted to merge these two concepts to create a game where the track was

generated entirely in real time using audio tracks. The game requires both personal skill and repeated attempts to get used to the mechanics, which encourages the player to try again and again.

Architecture



Methodology

Audio Incorporation

Multiple methods for audio incorporation and beat detection were tested out, but ultimately the audio in this project was implemented using the WebAudio API in a similar approach to the COS426 2020 assignment “mpThree.js”.

The first portion of audio incorporation, namely the user upload feature, was handled by the index.html file using an audio element and event handlers within the <script> tags of the html file.

The actual audio playing and analysis was handled by different components of the WebAudio API. The typical flow of the WebAudio API works as such: first, an AudioContext and an audio element linked to the context are created, then effect nodes are appended to the audio context, and finally they are linked to the destination to output the sound. In particular, this project made use of the AnalyserNode, which specifically is designed to calculate audio frequency analysis. The delayNode was also experimented with for track generation, but ultimately was not needed for the final product. The sound file linked to the audio element would update upon user file upload.

The primary goal of this project was to have the turns and jumps in the path line up with beats in the audio. Therefore, we needed to 1) implement a beat detection algorithm, 2) parse it into something the

rest of our code could decipher, and 3) generate the path based on that sequence. The rest of this section will discuss 1) and 2).

The beat detection algorithm for this project was referenced from the mp3js writeup. For every instance, we calculated the immediate energy of the audio using the frequency analysis returned by the analyser node, and then averaged this energy across a buffer. If the instant energy was higher than the average energy multiplied by a constant, then the instance would report a beat. Certain songs have higher constants than others, so various constants were tested with different audio tracks to find a value that worked the best across different samples. In order to make sure that we were not constantly reporting beats, there was a minimum time threshold set for beats - only after this amount of time would another beat be reported. We also passed the raw frequency data information, stored as a Uint8Array, into our scene to handle and generate the road.

Procedural Generation

The road is made up of many individual blocks that are placed in a certain alignment to resemble a continuous road. The individual blocks were created in Block.js, which used GLTFLoader to import a glb file for the model. As the block was created, a bounding box, stored in this.bb, was created for it as well: the bounding box dimensions were calculated according to the box model, and were stored as a mesh. Other methods for the block included remove(), which would remove the block element from the page completely, and updatePosition(), which would move the block closer to the player as they progressed forward. A secondary block, InitBlock, was created as the starting platform for the player.

The Coin model, representing collectibles, was created very similarly, with a model imported using GLTFLoader and a collision box created around the model. Coin() also stored information on whether the coin instance had been collected or not.

The Road.js file handled most of the procedural generation logic and frequency parsing. addBlock() handled the block generation logic: every time it was called, it would determine the current directionality of the blocks (left or right) and whether or not a beat had just occurred. If so, it would either change the direction of the path or add a space for jumping based on random chance. If the path was about to move out of bounds of the screen, it would be forced back into the other direction. Creating the block involved storing both the Block() object and its bounding box in arrays for easier access. Finally, coins had a certain probability of spawning on top of random road blocks, and they were added to the scene in the same way blocks were.

As the player progressed down the path, the blocks would gradually move closer to the camera and eventually offscreen. Any blocks and coins that went offscreen were immediately destroyed to reduce lag and processing power for the application. After a certain amount of time between each block's spawning, the update loop called addBlock() to make sure the path continued generating endlessly.

Game Logic

Overall, the instructions for the game are very simple, the user just has to stay on the track by turning left and right and jumping. The user should also try to collide with as many coins as possible to increase their score. Colliding with a coin increases their score by one, this doesn't currently change at any point in the game. Falling off the track at any point during the game results in game over, there are no lives or any amount of second chances, the player just has to start over again.

Most of the game logic controls were handled in the app.js file. There was a click event handler on the "Play" button on the landing screen, which would cause the app.js file to start the scene. This included rendering the PathTest.js scene and road, beginning audio playback, and updating the control variables. When the ball fell off the track, app.js called the lose() function, which would reset all of the game control variables and transport the player to the Game Over state.

On the Game Over screen, we overlaid a UI button over the scene that had an event listener attached to it. When the player clicks it, the page will reload, allowing them to play the game again.

Player Controls

The player can control the ball using the keyboard. In the PathTest.js scene file, we added a function move() which reacts to the input from the keyboard. When the player presses left/right arrows, the ball turns left/right respectively. When there's a gap in the map, the player could press space or the up arrow to jump.

The Ball.js file contains the model of the ball and its movements. At first we tried to use SphereGeometry from ThreeJS; however, due to the lighting conditions in the scene, the ball looked like it was 2D. Then we loaded a model online with a poke ball texture.

The ball could turn left, right, jump and fall off the track (in which case the game would end), and has states isLeft, isRight, isFall and isFallen. When the ball is turning left, the x element of the position vector of the ball is subtracted by the movement velocity, and when the ball is turning right, the x element of the position vector of the ball is added with the movement velocity. In order to display the turning effect of the ball, a rotation velocity is added to/subtracted from the z element of the position vector. In the update() loop, the moving left/moving right operation is implemented continuously. For jump and fall off the track, they are implemented using TweenJS. For jump, the fall down operation is followed up with jump up operation. We considered the falling off movement of the ball as a flat throw movement. For falling off track, only fall down operation is implemented, and we adjust the time to complete the operation to make the motion more real. Once the y element of the ball position vector falls below a certain threshold, the state isFallen becomes true, and the ball is removed from the scene.

Collisions

Each of the individual elements that could collide (the ball, the individual blocks in the path, and the coin) had a bounding box mesh stored in its instance. The handling of collisions itself was within the update loop of PathTest.js.

For each instance, the current list of collision meshes for the roads and coins present on screen were calculated, as well as the bounding box of the ball based on its bounding mesh at that current instance. We calculated whether or not the ball was colliding with any of the road blocks present using intersectsBox(), and if not, we classified that as a “fall.” which would end the game by invoking the ball’s falling mechanic and stopping the game loop.

A similar collision detection was implemented with coins; upon detecting collision with a particular coin, we removed it from the environment and incremented the player’s score.

Results

We measured success by playing that game and making sure that everything we had attempted to implement worked, and that the experience of the user was what we intended it to be, which is engaging yet relaxing. For example, testing the gameplay logic and the physics involved falling off the edge of the track on purpose to see if the ball would fall like it was supposed to, and that the game over screen would appear. These experiments guided our debugging and helped us make important design decisions, like when we decided to try and make sure that the next block after a gap is in the same direction and how we noticed that the collision detection was buggy at one point. Our results indicate that the game has reached its intended goal and that any bugs do not prevent the user from enjoying the game.

Discussion

Overall, the approach we took is very promising because it directly aligns with our intended user experience. We knew from the beginning that the most important feature would be the procedurally generated track because the biggest draw for our game is how customizable it is, and that for the user the experience of playing the game would be more important than achieving the highest score. Because of this we started with the track and this feature was finished before we started the motion of the ball, and the game logic, which were completed in that order. A different approach would be to emphasize the game aspect and create levels and to incentivize higher scores by adding different features when a score is achieved but this was not in line with our original vision for the project. The next steps would be to sort out any minor bugs to make gameplay as smooth as possible, and to improve the aesthetic of the game by adding more visual elements that are not necessarily part of playing the game. We learned a lot doing this project - some of the biggest things that we gained were familiarity with Threejs, and how all

the elements (ex. Scene, Object, Camera) come together to make an interactive virtual environment. We also developed our teamwork and communication skills, and managed to find a way to collaborate very smoothly and support each other, to the end of creating as awesome of a game as we could.

Conclusion

In this project, we successfully implemented all of our target goals, creating a game that analyzed real-time audio to procedurally generate a path and allowed players to turn and jump in order to avoid obstacles and stay on track. We also implemented collectibles and scoring, and multiply UI layouts to provide the user with a full game experience. We have incorporated many key topics from the COS426 curriculum, such as 3d mesh objects, mesh editing operations, event listeners, physics and collisions, and scene hierarchies and architecture.

Given more time, we would like to expand upon the aesthetic vision of the project, adding more backgrounds, roads, player meshes, and potentially different modes. Additionally, it might be interesting to experiment with different beat detection constants and have the speed of the track reflect the tempo of the song. We would also like to further improve our collision detection of the game and potentially create a more modularized method for collision detection.

Contributions

Giao

- webAudio API implementation
- Frontend development (menus, scores, UI)
- Importing models
- Procedural generation of the road
- Collision detection
- Collectibles and scoring

Yuanqiao

- Player control
- Ball motion

Yasmine

- Game logic
- Game over scene
- User interaction
- Text meshes

References

Assets

- https://sketchfab.com/3d-models/minecraft-grass-block-84938a8f3f8d4a0aa64aaa9c4e4d27d3
- https://sketchfab.com/3d-models/gem-618236229c4c4e4a97ff0fe87f0671a2
- https://sketchfab.com/3d-models/pokemon-poke-ball-9f63803d93714eadaa869c23cee0581d

Open Source Code

- https://github.com/mayoyoyo/mp3js/blob/master/COS_426_Final_Project_Report_mpthreejs.pdf