COS 426: Computer Graphics **(Due: 5/12/20)**

mpThree.js - beats to relax/procrastinate to

Names: Hanson Kang (jaehok), Johan Ospina (jospina), Greg Umali (gumali)

Abstract

mpThree.js is a music-adapting retro racer that is powered by YOUR music. Users upload their own .mp3 files, and the game elements interact with the music to create a unique experience. Over the course of the song, players avoid red 'death spheres' and try to collect gems and power-ups to rack up as many points as they can.

Introduction and Inspiration

Given the scope and time constraints for this project, we decided that creating a take on an 'endless runner' game would be relatively simple to implement and would naturally integrate a variety of concepts covered in the course, from shaders to collisions to physics simulation. However, while brainstorming for our project, our group also gravitated towards the idea of visualizing audio for our project. Inspired by games such as BeatRacer that synchronize an infinite runner game with the accompanying backing track, we combined these two ideas to envision a game in which music could be parsed in order to create the elements of our infinite runner game - whether for obstacles, scoring orbs, or other environmental elements. Special thanks to Sehor Young '19 for suggesting the game name.

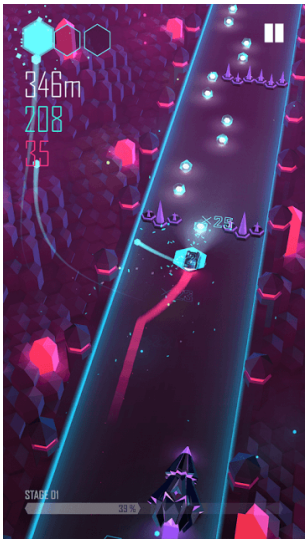


Fig. 1. Screenshot from BeatRacer by ZPLAY Games

Over the course of creating the game, we eventually steered away from using audio analysis data in order to create different obstacles; while beat detection was feasible, isolating constituent instruments from a user-uploaded .mp3 file proved non-trivial. Instead, we turned towards using beat detection to set the speed of the incoming orbs and track, and implementing our audio analysis in shading bands in the wall based on sampled frequencies.

Drawing upon this basic concept of a runner/racer game staged within an environment that reacts to uploaded music, we were able to successfully create an MVP with simple physics for player movement, and a 'treadmill effect' for incoming obstacles. Further iterations introduced further styling of visual elements such as the background, fine-tuning of wall reaction to audio, the addition of a player skin that reacts to collision events, and two power-ups, among others. In the end, we were able to create a polished end product that we found ourselves getting hooked on over the course of creating it!

Architecture

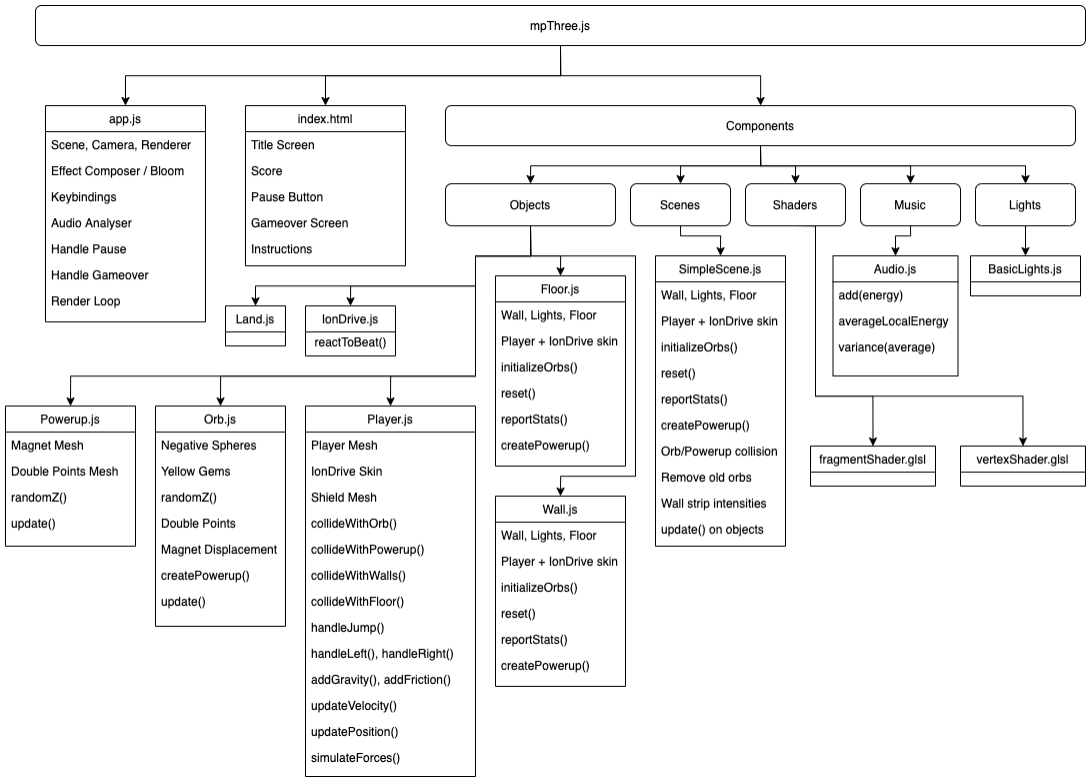


Fig. 2. General architecture of mpThree.js

Implementation

Audio Processing

Fig. 3. WebAudio API Flowchart

The flagship feature of mpThree.js is its interaction with the audio input. The basics of audio preloading, uploading, pausing, and playing actually are generally handled by the `index.html`, using the HTML5 audio element.

However, a more robust system is required for actually using the audio beyond simple loading and playing. All audio components for data analysis were handled with Mozilla's WebAudio API. The workflow for WebAudio API is extremely specific and must be handled in the exact manner specified. A WebAudio workflow usually involves the creation of an `AudioContext`, the creation of an audio source (file can be pulled from the HTML) within the context, the creation of effect nodes, and the setting up of a destination to output the sound (e.g. speakers). Once this is all set up, the input is connected to the effects, and the effects are connected to the destination (Figure 3).

In particular, the `AnalyserNode` was the effect that was used throughout the project. The `AnalyserNode` is an interface that is able to return audio frequency analysis in real time. It uses Fast Fourier Transform (FFT) to determine audio frequency data. Simply specify an FFT size (the number of bins to divide audio frequency into) using `AnalyserNode.fftSize` and call the `AnalyserNode.getFloatFrequencyData()` function to return a data array of length equal to half the FFT size. In the game, the walls were inspired by the audio visualization common in media players, and the color intensities of the strips were set by an AnalyserNode with an FFT size of 128.

The initial goal of this project was to have the orbs generated in sync with the beats of the music. The problem with this is that beats can only be determined in real time and the orbs have to be generated in advance. The only option would have been to use a buffer to preprocess the entire song upon loading, a task that seemed too much for the project. Instead, a minimal beat detection algorithm was used to vary the speed of the game.

Beat detection [2] is a surprisingly simple process that involves, at every instance in the audio, finding the instant energy of the frequency analysis. This instant energy is then averaged across a buffer of fixed size (an

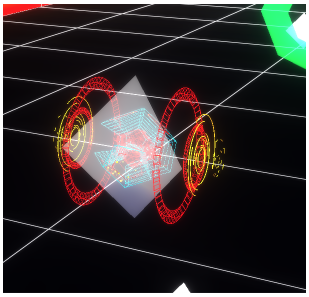
optimal buffer for an FFT of 1024 bins is considered to be 43 samples). If the instant energy at an instance is higher than the average energy multiplied by a certain constant, the instance is considered to be a beat. The determining of the constant is ideally handled by calculating the variance and using it in a linear equation, but generally, a beat-heavy track such as techno or rap will have a constant of 1.4-1.5, while rock songs can use a constant as low as 1.1.

For mpThree.js, an `AudioData` class was created in `Audio.js`, which is a linked list that acts as a buffer for 30 samples. The `add()` function adds an instant energy to the buffer and `averageLocalEnergy()` returns the average of the energies currently occupying the buffer. In `app.js`, the instant energy is compared to the average energy of the `AudioData` buffer using a constant of 1.15. The beats will be detected a little more frequently than desired when a beat-heavy track is used, but 1.15, through trial and error, proved to be a good constant for a wide range of songs. If a beat is detected, and the time elapsed since the last detected beat is short enough, the speed of the track is deemed to be fast and the game speed increased by 0.005. If enough time elapses since a beat without another detected beat, however, the song slows down by 0.005 at each frame.

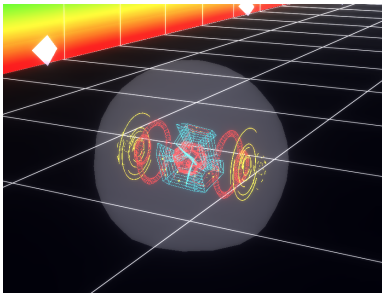
Gameplay Elements

The player seen in gameplay is an imported mesh called IonDrive from [6]. Also included within the player Group, however, are two initially hidden additional meshes that surround the player. The first is a diamond (1-segmented sphere) mesh that is used to create the animation for collecting a sphere by setting the opacity of the mesh to a 0.4 upon collection, then fading this opacity over time in the `update()` function. The second is a 16-segmented sphere that is used to visually implement the "shield" functionality - when the player presses the ArrowDown key, the opacity is increased to display the protective shield around the player. The four arrow keys are bound to four different player actions. The first, already mentioned, is shield - a state during which the player collects no points nor takes any damage. Jumping and horizontal translation is implemented with the use of a `netForces` vector similar to Assignment 5, which is then used to calculate velocity and position at each time step. For jumping, a gravity vector is added to the `netForces` of the player, and upon an event from the ArrowDown key, a vertical `jumpForce` vector is applied to `netForces` if the player is currently touching the ground. For horizontal translation using the ArrowLeft and ArrowRight keys, a translational force along the z-axis is applied similar to `jumpForce`, and a friction vector opposing this motion is added if the player is touching the ground. In the update function for player, functions are called to simulate these functions, collide with elements in the environment, and fade the orb collision diamond mesh over time.

There are three different kinds of "orbs" that are implemented in `Orb.js`. The first two are white and yellow gems, which give the player +100 and +500 points respectively. In order to make the mesh for the gem, we just used 1-segment Three.SphereBufferGeometry. The other kind of orb are the 'red death spheres', which are made of a 4-segment sphere. Whether the gem is white, yellow, or red is recorded in the the orb's state, and the mesh is created and colored based on the appropriate state attributes. These orbs are generated based on the passed-in z value of the previous orb - each successive orb has a 1/3 chance to either positively deviate, negatively deviate, or remain in the same 'track' (z-value) as the previous orb. Rather than staying static and having the player move forward towards them (which would require the generation of new chunks of wall and floor, for example), we instead have the orbs move towards the static (on the x-axis) player, with the walls and floor shaded to give the illusion of movement (see Environment). In their update function, these orbs simply are translated towards the player along the x-axis. These orbs are generated in `SimpleScene.js`, where current orbs in the scene stored in



(a) The diamond-shaped player mesh



(b) The shield mesh

Fig. 4

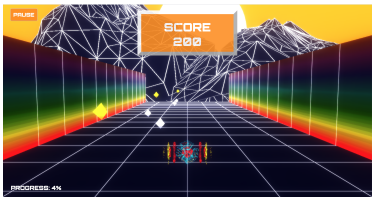
the `orbs` array. Once an orb moves behind the position of the camera, it is removed from `orbs` and deleted from the scene, and a new one is generated at the end of the array to take its place.

The last kind of collectible in the game are the two power-ups. The green torus makes all gems have double value (with spheres worth +2000 instead of -1000) and the blue cone 'magnetically' pulls all gems towards the player (with spheres worth +1000 instead of -1000). These power-ups are generated randomly in `SimpleScene.js`, and are implemented in `Powerup.js`. They translate towards the player similar to orbs, albeit at a slightly faster speed, and do not spawn based on the previous orb's z-value. Upon collision, these power-ups also iterate through the scene's `orbs` array to change the color of the meshes of affected orbs. Implementing double points just involves a simple state check during score calculations, while implementing the magnet involved the addition of a small translation towards the player in the `update()` function of the orb object. Finally, the last problem that had to be solved was the issue of timing - timing how long power-ups should last, and how often they should respawn. Rather than having some sort of asynchronous timer, we instead opted to mark time based on the creation/destruction of orbs (since this happens at a more or less regular rate). Therefore, we added `powerupTimer` and `powerupRecharge` attributes to the scene's state, which are set upon gaining a powerup or spawning a powerup, and are decremented when a new orb is created.

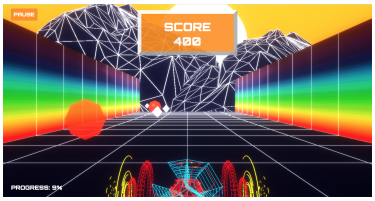
The final gameplay feature we added was the three different camera views, bound to the 1, 2, and 3 keys. This feature was implemented by simply change the pose of the camera upon the corresponding keyboard event. For view 2, the camera follows closely behind the player. For this one, we simply update the position of the camera in the render loop of `app.js` based on the current position of the player.

Collisions

The only relevant collisions in the game are between the player and environmental structures (such as the wall and floor), and between the player and orbs/power-ups. For colliding with the wall and floor, after simulating forces, if the calculated position is beyond the bounds defined by the structural element, the position of the player is simply clamped back to that bounds position. If the player is at its lowest y-value, then the function `currTouchingGround()` returns true, which is used to determine whether the player can jump, and during friction calculations. For orb/power-up collisions, rather than implementing a strict collision with the mesh of the orbs and that of the player (involving ray-tracing from points between the two meshes), we instead check for a collision between rough bounding boxes around them (a comparison between x, y, and z coordinates of the player and of the orb). When a player collides with an orb, the orb's `visible` is set to false (which causes its



(a) Camera view 1



(b) Camera view 2

Fig. 5

(a) Camera view 3

Fig. 6

opacity to go to zero), and the diamond mesh surrounding the player is set to an opacity of 0.4 and colored the same color as the collided orb. As previously mentioned, in the `update()` function of player, this opacity is then gradually decreased (in this case, by .0035 per time step), which then creates the effect of a brief diamond-shaped flash around the player on orb collision.

Environment

For our soundscape experience we combined 4 different components, the first was an edit to only use the mountain mesh from [7]'s 90's vaporwave landscape. In order to fully customize this landscape; however, we needed to modify the gltf's materials to become unlit in order to fit our aesthetic. Once we made those changes, it was easy to match the color of the mountains to floor. To add a nice garnish to our environment we added a "Sun" behind the mountains to give a feel that the player is going on a calming drive as can be seen in 7.

In order for this to be a true music game, we needed to have some elements that reacted to these signals. For the visuals, we took two approaches, the first was a simple effect on the mesh (which can be seen as it collects gems) that would have reacted to the output of the beat detection step mentioned previously. This beat effect was created using a simple linear interpolation method, so that at each update call the Ion Drive would interpolate halfway between the scale it is at and a target scale value. The way this system works is that an outside force would call `reactToBeat()` with a specific target scale and that would set the target scale variable. In the update loop, the Ion Drive will calculate the linear interpolation half-way between the current scale of the red rings and `targetScale`. This essentially smoothly drives the scale towards target scale. At the same time, the Ion Drive is also decaying the target scale by 5% each frame, this will cause the target scale to start getting smaller and smaller until it hits a minimum value of 0.8 (the resting state). These steps create the pulsing effect that can be seen as the player collects Gems as can be seen in 8

The second music reactant element, which made it into the final build, was a wall of strips that reacted to levels from the Fast Fourier Transform output. This wall was laid out programatically to have a variable number

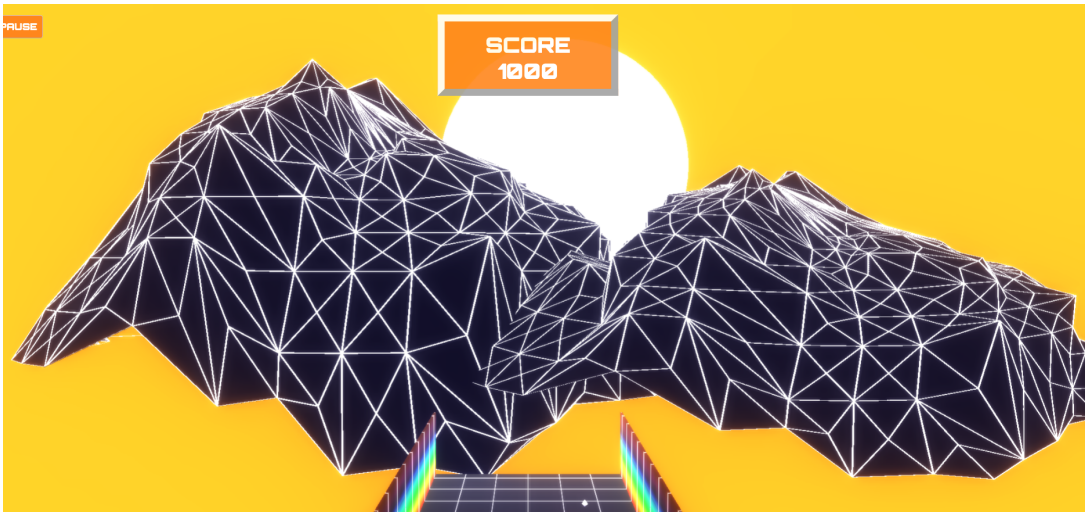
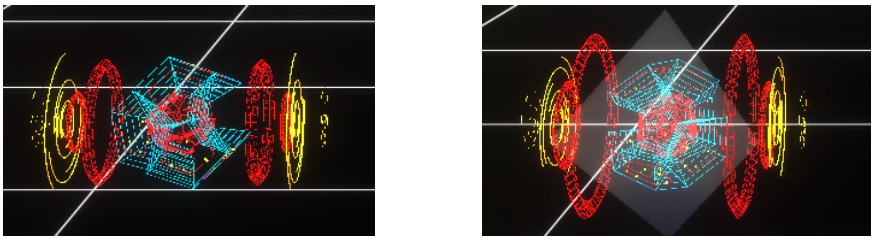


Fig. 7. Close up of our mountain sunset environment



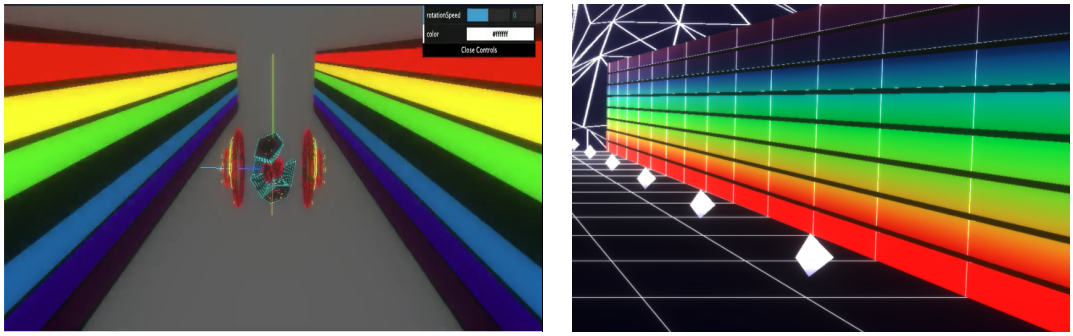
(a) Resting State (b) Pulse

Fig. 8. Orb collision effect on the Ion Drive

of strips (Box Geometries), each with their own margin and padding values as can be seen in Figures 9a and9b. To make these interactive, the next step was changing the colors of these box geometries at will. To this end we defined a custom Vertex and Fragment Shader that could take custom uniforms from the ThreeJS runtime via the ShaderMaterial object. In this custom shader we defined a custom uniform for a static color that the strip needed to take on, as well as an intensity. At runtime, our game will set a dictionary of `uniform` variables to the Custom Shader. The visual end result of this specific system can be seen in 9a.

The strip intensities work similarly to the Ion Drive, however, instead of scale being decayed over time, it is the intensity of the color. This means that the colors will be decayed over time until they reach a resting state (0.15). The scene will let these strips know the current music intensities at a slower sampling rate, which will excite the strips until they decay over time to their resting value. By doing it this way, we ensure that there's no lag between what is being heard and what we are visualizing.

The final piece to our environment was creating a Treadmill effect, as can be seen in 11. We chose a treadmill effect for performance reasons and found that it creates a very compelling effect of forward movement even though the Ion Drive is fixed on a 2D line in 3D space. This effect is made entirely by manipulating uniform variables in the Fragment Shader and is based on [1]'s approach to an anti aliased grid shader using screen space



(a) Simple Strips

(b) Our walls are made of a variable number of strips that react to sound information

Fig. 9

partial derivatives. We extended this shader in our project by adding a light falloff in the distance as well as normal calculations to draw a grid on horizontal surfaces and vertical lines on vertical ones. We also introduced sizing and offset uniform variables we set in our code to both scale and offset the grid by a uniform amount in order to create the perceived effect. Finally, our shader does some color interpolation between strips in order to create a smoother gradient as shown in 9b.

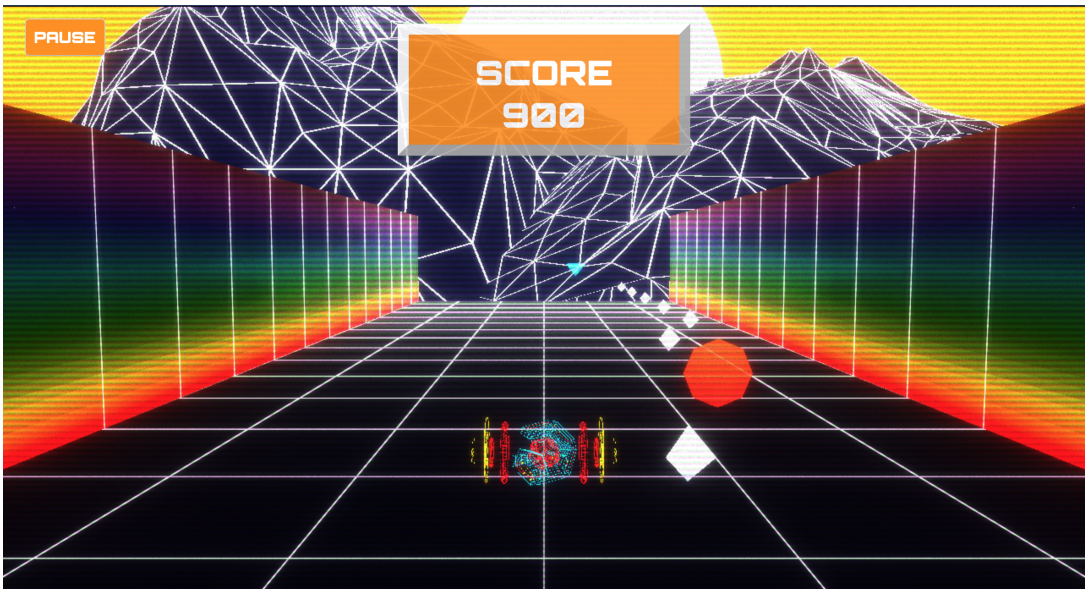


Fig. 10. A shot of the game in action

Future Work

Currently, The biggest limitation on level design is the fact that we have static planes of a set size giving the illusion of moving forward. Player engagement would increase if tracks of varying shapes could be implemented, which this project currently does not support. Additionally, the illusion of movement is lost if the player sets the pose of the camera outside of the "treadmill" (example in Figure 11. Instead of having moving objects go down a set hallway, a player could traverse a sophisticated space that also interacts with sound. Another item to improve upon is the sound reactive shader. Currently, this shader does not react to lighting at all and thus it looks flat against other shaded object. In order to mitigate this, we implemented the ThreeJS UnrealBloom and Film render passes. These post processing effect passes were attached to our initial rendered output and successfully changed the ambiance of the game into something more aesthetically pleasing. In the future this could be vastly improved by building on top of more sophisticated shaders that do interact with lighting. Additionally, more robust audio processing could elevate this game further by interpreting audio signals in different ways.

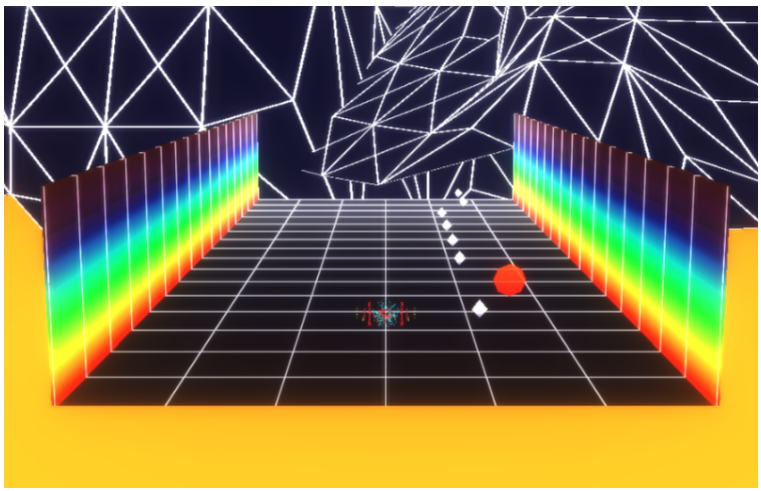


Fig. 11. A view behind the illusion we create

Conclusion

For creating a continuously moving game, a Treadmill approach is imperative for low memory consumption and high frame rate requirements such as processing audio. A simple approach with an unlit shader went a long way with a minimalist aesthetic.

P.S Try refreshing the page after you perform a Vibe Check.
P.P.S Try pressing S and H at the same time on the keyboard at any point.

Contributions

- Hanson
  - AudioData and WebAudio API implementation
  - Game dynamics
  - Frontend development of menus, score, buttons, etc
  - UI Work
- Johan
  - Laying out environment
  - Vertex/Fragment Shader that reacts to music
  - Worked on Walls and Floor interacting with uniform variables for the shader
  - Connected Bloom and Film effects
  - UI Work
- Greg
  - Movement physics
  - Power-ups
  - Orb creation/destruction
  - Environment/Orb Collisions
  - UI Work

REFERENCES

[1] "Anti-aliased grid shader - http://madebyevan.com/shaders/grid/." [Online]. Available: http://madebyevan.com/shaders/grid/
[2] "Beat detection algorithms." [Online]. Available: http://archive.gamedev.net/archive/reference/programming/features/beatdetection/)
[3] "three.js." [Online]. Available: https://threejs.org/docs/
[4] "Webaudio api." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
[5] D. Egurrola, "15 of the best royalty-free tracks of 2018." Envato Tuts, Oct 2019. Available: https://music.tutsplus.com/tutorials/15-of-the-best-royalty-free-tracks-of-2018--cms-31894
[6] I. Follow, "Primary ion drive /// - download free 3d model by indierocktopus (@indierocktopus) [d3f50a6]," Jan 2017. Available: https://sketchfab.com/3d-models/primary-ion-drive-d3f50a66fee74c6588dd9bc92f7fe7b3
[7] D. T. Yamaguchi, "90s vaporwave neon grid by diego t. yamaguchi," Dec 2019. Available: https://sketchfab.com/3d-models/90s-vaporwave-neon-grid-animated-0b8208f8e2fc46d19a086e353c9ae63a