Stemscapes

Abstract

Our project is a scene-based stem player where the player can enter different quadrants of an environment to trigger a stem of a song (drums, instrumental, bass, vocals). Each quadrant has an associated stem as well as season that informs the objects within it. For example, the vocals quadrant corresponds to winter, so it is fully furnished with snow topped trees, rocks, and bushes.

Introduction

Goal

The goal of our project was to create a world based stem-player. A stem player is an audio remix device that allows for the individual stems of a track to be layered onto each other. These stems often include bass, drums, or isolated vocals. Our aim was to create a world that was subdivided into quadrants where walking into a specific quadrant triggers a specific stem to play. Each quadrant would be associated with a specific season. By walking around the different quadrants, the player is able to layer the stems to create a pleasing melody. Players would not only be stimulating their visual experience by walking through the pleasant scenery, but also their auditory experience by layering the different beats.

Previous Work

A previous project called “Inside Music” was created by the Google Creative Lab. This application allows a user to step into their favorite song by seeing and hearing the individual layers of the music. The user is surrounded by different spheres that play different parts of the song, such as lead vocals, guitars, bass, etc. When a user turns to a certain sphere, this activates a specific stem. The application gives the user an immersive experience by utilizing spatial audio to bring the individual pieces of the song together.

Any attempts to run this project though failed. The application seems to be outdated and the steps laid out in the project’s Github repository did not allow us to successfully test the application. From the screen grabs of the project, it seems that the user is constrained to a specific location. The camera is stationary, but the user can move their mouse to change the orientation of the camera to activate different spheres. The surrounding background is just a black screen. Thus the user doesn’t have much of an immersive visual experience.

Approach

We utilized ThreeJS to create our stem-player visualizer. We created a floor that is separated into four quadrants. The floor is differentiated by different colors to mark different seasons. The trees, bushes, and rocks all also look different based on the season of the quadrant it is in. The objects are randomly interspersed throughout each quadrant. The user can move themselves throughout the scene using the arrow keys. By moving their mouse around, they can look at the scene from different orientations.

In terms of the music capability, we utilized Spleeter by deezer. This is a source separation library with pretrained models that splits up a song into its different components. Based on what song the user selects, this sends a request to our Node server. Splitter runs from the command line, so we have a script that initiates this music splitting process. The program then plays the different stems but they are all muted for the player. The user can use the arrow keys to move around, and once their location is inside a specific quadrant, the specific stem is played. When the user leaves the quadrant, the stem is muted. If a user clicks the sustain button, then the stem will keep playing and layer with the other quadrants they visit.

Methodology

Interaction

For my part of the assignment, I implemented camera functionalities. I created event handlers for the arrow keys/WASD and the space bar. By utilizing the PointerLockControls Camera, this allows the user to move the camera's view around with their mouse. I also implemented a screen with instructions to guide the user on how to use our application. Overall this allows for a first person POV. I also utilized a raycaster to allow for object detection. This prevents the camera from passing through objects. This also prevents the user from leaving the allotted land space as mountains surround the perimeter of the world.

Another implementation possible was for controls that allow the user to fly where left click allows them to move forward in the direction of the mouse and right click to move backward. We decided against this implementation as we wanted the user to be able to appreciate the scenery we created. By keeping the camera at a constant height, the user is constrained to the world we created.

We also did not implement body animation for the first person POV. This was a stretch goal that we did not have time to attempt.

Objects

In order to construct a semi-realistic scene of a forest, we needed to import 3D models of rocks, trees, bushes, clouds, signs, and mountains. We chose to use low-poly models to meet the project's aesthetic as well as to reduce loading times since we anticipated having many instances of each type of object to populate our environment. We found all of the low-poly models in our project on poly.pizza as .glb files. To be able to use the well-supported and

well-documented GLTFLoader, we had to convert these .glb files into the .gltf file format using the 3D open-source modeling software Blender. Once converted, we would simply load the model at the specific file path using the Three.js loader.

To use the imported 3D models, we followed the modular approach started in the seed project. That is, we created a class for each model, and then instantiated that object in the scene. Within an object's class, we load the model with the GLTFLoader and add it as a child to the parent Group. Initially, the object constructors only supported passing in the position the object should be placed at as a Vector3, but after starting to populate the environment we needed additional transformations to create a more natural-looking scene. So we added the ability to pass in an integer representing a scale factor and a Euler object representing the rotation. The position, scaleFactor, and euler parameters were made to compose a transformation matrix that was then applied to the model after loading. These parameters as well as our modular approach allowed for us to easily populate the environment with randomly placed, scaled, and rotated objects.

Once objects could be added to the scene, we focused on the stretch goal of animating these objects. In particular, we wanted to have swaying trees and moving clouds. We accomplished the former by creating the animation in Blender, exporting it with the .gltf file of the model, and then playing the clip from the AnimationMixer object created from the model's embedded animations. Animating the model in Blender allowed us more control over the movement of the mesh than rotation, scaling, and translation would give us if we had used code to simulate the swaying trees. We created animations for the pine trees specifically since how they would sway was clearer than that of the other types of trees. To simulate moving clouds, on the other hand, we simply translated the position of each cloud in the x-direction. When the cloud's position exceeds the bounds of the environment, we reset its position to the other side of the world to create a constant stream of clouds. These animations were ultimately simple to create, but required an entirely different animation system.

That is, the animation system in the seed project needed to be refactored to allow us to animate the trees and clouds. The problem was that the seed project relied on the current timestamp to animate objects. For example, the flower bobbing animation uses a sinusoid function of the timestamp to determine the current rotation. The timestamp monotonically increases while the application runs, but this is no problem because a sinusoidal function is periodic so the rotation remains within appropriate bounds. However, this poses a problem for linear animations such as moving clouds or swaying trees where a constantly increasing timestamp makes the animation faster and faster. To fix this problem, we used a Three.js Clock object to compute the change in time since the last rendered frame (AKA delta), and then passed in this delta into the update function of certain objects instead of the current timestamp. Some of our animations, such as that of the water, still relied on the timestamp so we used conditionals to pass in the appropriate parameter to an object's update function. This approach not only fixed the issue of the animation speeding up, but it also maintains the same animation speed across different frame rates.

Our final object related goal was to allow for shadow casting. Although it would be more efficient to export shadows from a Blender than to use the Three.js shadow map, we do not have enough experience with Blender to be able to do that ourselves. More importantly, we wanted to programmatically generate our terrain, so using Blender for shadows was out of the

question. Instead, we enabled the shadowMap of the renderer and adjusted the directional light to cast shadows over a large enough portion of the world. Once we enabled shadow receiving of the floor of our world and shadow casting of the objects, we finally had a semi-realistic forest scene.

Environment

In facilitating the traversal of a scene that triggers audio output, the scene itself needed to be created. Since we wanted to embrace a “low-poly” look as part of our project’s aesthetic, the minimum viable product for this scene required there to be some form of low-poly ground to traverse across, separation of this scene into 4 distinct quadrants (for each of the 4 music stems), population of the scene with the models we imported, and some form of boundary marking the limit of the scene’s traversal.

In designing the low-poly ground, there were many options to decide between, each posing their own challenges.The first approach used was the creation of a fully customizable geometry (through the use of the removed “THREE.Geometry” constructor). This produced the desired look, but proved to cause issues in the separation of quadrants–since no uvs coordinates were defined in the creation of the Geometry object, texture-mapping wasn’t supported. Upon investigation into this issue, we realized that support had been removed for Geometry, so we pivoted to using a more efficient BufferGeometry. This posed its own challenges because, even after finding a way to adjust the vertex positions to create random noise in the terrain, we could not create sufficient separation of faces to produce the same low-poly look produced by the original THREE.Geometry approach. However, through the use of an example implementation from Three.js, we learned how to adjust the color of the plane between vertices, allowing us to create gray-scale variation in the plane that was then used to shade our texture map and create distinct triangular partitions in the terrain.

In order to visibly distinguish between the different quadrants, we entertained various strategies like delineating the quadrants with lines placed along the terrain’s mesh, separating the mesh with lines of objects such as fences, pebbles, or paths, or making each quadrant a different a different color by lining up four distinct meshes. We ruled out lines due to a lack of visual appeal and rows of objects due to the implications of limited traversal it would convey to the user. The complications of ensuring that four separate meshes would remain joined at the scene ruled out that possibility as well. We were left with the option of using a texture map to create different colors for each quadrant. As such, one of our team members drew up a texture map with our desired colors split into four quadrants on a square canvas. The use of the texture map revealed issues in our initial terrain implementation and inspired the transition to the method described above that we actually used. We took advantage of the fact that if a texture map is applied to a mesh with a predefined color, the resulting color of the texture mapped mesh is the composite of the two colorings. Using the grayscale shading defined in the mesh combined with the solid coloring of the texture map, we created the visual appearance of a low-poly terrain separated into four quadrants by four distinct colors.

The insertion of objects was easily implemented due to the creation of object constructors. As such, we–in most cases–needed only to declare a position vector and pass that to the object constructor to place that object at the desired position in the scene. Upon

placing the objects, we realized that resizing the objects could not be easily accomplished after they had already been added to the THREE.Group object type the models were loaded into. As such we made minor adjustments that allowed for passing a scaling variable to the object constructor which facilitated the scaling of the loader's scene, and the consequent scaling of the object itself.

The last feature of the MVP was the scene's boundary. Initially we thought of having a seemingly "infinite" plane that faded into the distance and that the player could not traverse due to an invisible boundary. We initially thought of using the fog property of a scene to do this. However, we then decided on creating a more enclosed space by creating a perimeter of mountain objects. We still utilized the fog property to allow the mountains and distant objects to fade into the distance with respect of the player's viewpoint.

We were also able to implement the stretch goal of inserting a small pond into the center of the scene that functioned as the "silent zone" where no music would be played. This was implemented by creating a mesh using a customizable THREE.Geometry, and animating that mesh at every timestep. The height adjustment of each vertex was influenced by the relative height of a cosine wave at that time step. This allowed the mesh to move in waves and emulate the simplified behavior of water. Further, we implemented our second stretch goal with regards to the scene by creating distinct color schemes for each quadrant and not relying on the different terrain colors to differentiate the different quadrants. This was accomplished by finding similar objects such as trees, rocks, and bushes differing in their seasonal themes.

As such we were able to accomplish all features of our MVP and all of our stretch goals with regards to the environment/scene.

Music

To implement the stem playing feature, we first pick an mp3 file from assets/songs/ folder. We provided some royalty free sample songs, but users are free to upload anything they want in that folder. To pick the mp3 file, the user just needs to enter the name of a file, the app takes this name and sends an http request to a server that runs alongside the main application. This server runs a terminal command that runs spleeter on the chosen file. Speeter is an existing program that uses artificial intelligence to split songs into their stems.

We split each song into 4 parts, the drums, bass, vocals, and other instruments. After spleeter was finished running, we had 4 audio files, each corresponding to one of the four stems. We loaded these files in using the ThreeJS audio library. After loading in the songs, we made them play on mute and made it so that when the player (the camera) enters one of the quadrants, it unmutes the stem corresponding to that quadrant. We also added a feature that allowed the user to 'sustain' a stem, meaning that it would keep playing even if they left the corresponding quadrant. This was done by simply setting boolean values for sustain variables corresponding to each of the stems.

We wanted to add a more user-friendly interface for uploading mp3 files and a dropdown to select mp3 files as stretch goals but unfortunately did not have enough time to implement these features.

Results

We measured success by making sure that the key components of our project such as traversal through a generated environment and stem playing worked. We made sure that the generated environments “made sense” and looked appealing. We also checked that entering different quadrants triggered different stems and that those stems only stayed playing while the camera was in that quadrant unless the user had toggled sustain for the stem in that quadrant. Finally, we made sure that the user was easily able to choose the song that they wanted to play with.

We also executed experiments to iterate on our project. We compared our animation with “real-life” examples to make sure that they were relatively realistic and faithful to their real counterparts. We tried different scenarios and uses for the stem playing capability and made certain that the program was acting as expected. And we got feedback from users to improve user experience.

Ultimately, our results indicate that our application works as expected and is easy to use.

Discussion & Conclusion

We effectively reached our goal by delivering a product that met our MVP goals as well as implemented a few of our stretch goals. We were able to create an traverable environment that enabled interactive music composition for our users. Further, we provided an aesthetically pleasing and peaceful environment in which the user can appreciate the beauty and excitement of creating music.

Our next steps would be to first finish the rest of our stretch goals. This includes the implementation of body animations that simulate the body of the user; since the camera functions as a first person POV, this would entail arm and leg animations in the peripherals/when looking down. Further, we would like to implement some wild life, including but not limited to a pet companion that accompanies that player as they traverse the scene. Lastly, it would be nice to create more distinct sections within the terrain, allowing for the separation of a song into each instrument rather than just the stems of a song. This would require a customized number of scenes based on the number of elements of a song as well as a pivot to a new coloring theme (as the 4 seasons would not make sense/can not be accommodated if there are more than or less than 4 audio components to layer).

One issue we would like to revisit in the current implementation of the project is the means by which objects are placed. At the moment, the position of an object is determined through the selection of a coordinate value from within the range of coordinates encompassed by each quadrant. This is a bit computationally expensive as two random numbers must be generated for every object placed into the scene. A potential fix to this problem would come at the cost of the visual distinction between the quadrants: reusing coordinates for objects within each quadrant. For example, if a tree was placed at position (x, 0, z) in one quadrant, then the corresponding tree in the next quadrant should be placed at (x, 0, -z). As such, the number of calls to Math.random() would be reduced by a factor of 4. This same process could also be

applied to the random sizing of those objects; each object sharing corresponding coordinates across scenes would also share the same randomly generated scaling factor.

Through this project, we all had an opportunity to delve deeper into the use of Three.js. We learned how to let a player move around a world, to import and animate scene objects, to create and populate an entire world, and finally play music while a player traverses our environment. All in all, we are proud of the product we were able to create.

Contributions

Thanya Begum

Setup GitHub Pages deployment as well as helped synthesize contributions via GitHub.

Implemented the importation of objects into the scene/environment:

1. Converted .glb files of the desired 3D models (rocks, trees, bushes, clouds, signs, and mountains) downloaded from poly.pizza into the .gltf file format via Blender
2. Loaded the .gltf files using Three.js and created reusable and customizable (i.e. positionable, scalable, and rotatable) Javascript objects
3. Created animations for swaying pine trees in Blender
4. Animated swaying trees and moving clouds using Three.js AnimationMixer and the animation system, respectively
5. Enabled shadows and adjusted lighting

Zoha Enver

Implemented camera functionality of the project:

1. Added interaction handles with the arrow keys and space bar
2. Added mouse interaction that controls the camera orientation
3. Created the instruction page for user
4. Object detection capabilities for the camera to prevent passing through objects in the scene/leaving the allotted world

Yusuf Kocaman

Implemented stem playing aspect of the project:

1. Set up a server.js file that would get requests from the main app and run spleeter in the command line to split given mp3 from songs/ folder into stems.
2. Loaded and started playing all stem files on mute when controls locked and made sure that the player could not move if the music was not loaded.
3. Made the audio file corresponding to a given quadrant unmute when the camera was in that quadrant.
4. Add capability to toggle sustain on any quadrant to keep that stem unmuted even if the player left the quadrant.

5. Set up hosting.

Ayah Saud

Implemented creation of environment:

1. Created terrain
    a. Created low-poly mesh with vertices of randomly varying height
    b. Created a texture map to delineate the four quadrants with four different colors.
2. Created pond object
    a. Created low-poly mesh with adjustable vertices.
    b. Set the height of each vertex as dictated by the value of the cosine function at the current timestep.
3. Placed objects in scene
    a. Randomly placed the themed objects within the corresponding themed quadrants
    b. Varied size of objects uniformly at random within a range (ranges different for each object)
    c. Placed clouds randomly throughout the scene, randomly sizing them within a given range, placing them at randomly varying heights, and rotating them about the central axis to create varying positions
4. Creation of perimeter
    a. Used mountain objects whose heights vary randomly within a range
5. Fog
    a. Set the fog property of the scene to be the same color as the background to create the impression of distant objects being out of view

Works Cited

Rock Moss by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/Oar39tFugM)

Rock Snow by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/eZRzCg5BcR)

Rocks by Jarlan Perez [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/72Q1FqBXvO7)

Rock by Danni Bittman [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/4TpBWdzKDf2)

Boulder by Poly by Google [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/3jql0qtape-)

Pine Tree by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/gX8WmgkeEm)

Pine Tree Autumn by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/n0X3N2yUFL)

Pine Tree with Snow by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/17vQv2X5rh)

Tree by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/2paAm1ja4w)

Autumn Tree by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/2lRubrT6Na)

Bush with Berries by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/TSbIxkDtxF)

Bush by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/92EytlU1El)

Bush Snow by Quaternius [CC0] (https://creativecommons.org/publicdomain/zero/1.0/) via Poly Pizza (https://poly.pizza/m/H4IEAwYl1z)

Clouds by Jarlan Perez [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/b3Kia9N2fS2)

Mountain by Poly by Google [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/099f6GxB1bj)

Spleeter by Romain Hennequin and Anis Khlif and Felix Voituret and Manuel Moussallam (https://github.com/deezer/spleeter)

Creating a Geometry from Scratch by Jos Dirksen, pg. 97 (https://services.math.duke.edu/courses/math_everywhere/assets/techRefs/Threejs%20Essentials.pdf)

Low-poly Ground from Misc Controls Pointerlock, Three.js examples (https://github.com/mrdoob/three.js/blob/master/examples/misc_controls_pointerlock.html) via (https://discourse.threejs.org/t/trying-to-get-a-low-poly-look-on-planegeometry/21273/3)