Daniel Huynh, Darren Zheng, Luke Maake

Princeton University

COS 426: Computer Graphics

Reactive Concert Simulator - Final Project Written Report

-Abstract-

This paper details the process of creating our "Reactive Concert Simulator." Created with NodeJS and Three.js, our project simulates a nightclub/rave setting, complete with a stage, an audience, and a visual effects show. The audio is visualized using a custom shader object that is manipulated using a vertex shader to update vertex positions and a fragment shader to update vertex colors. The custom shader object is manipulated according to the frequency data of the song being played. Additional effects, like fireworks, are also added to elevate the visual effects show.

1. Introduction

For our final project, being avid music enjoyers and concert goers, we wanted to create something that combined these interests with computer graphics. We decided that an audio visualizer was the perfect idea for this, so our main goal was to simulate a nightclub/rave environment in Three.js, using the audio visualizer as the main visual effect for the "performance." The scene also has a stage and an audience, as well as supplementary visual effects like fireworks.

Our target audience include music enjoyers and concert goers like ourselves. If someone wants to enjoy a visually stimulating virtual concert while they listen to music, this would be perfect for them. There exists a variety of audio visualization platforms, but we wanted our project to have an array of features that would make it slightly unique. Some of these features include fireworks, emulating the visual stimuli that would take place at a realistic concert, and user interactivity.

2. Previous Work

We made ample use of ThreeJS and its classes. The stage and people figures we used come from pre-existed templates that we modified to our liking. Additionally, ThreeJS contains an Audio class. This Audio class allowed us to extract the frequency data of the audio file we inputted and store that data into an array. In order to learn how to manipulate objects according to the frequency data, we watched a one hour YouTube tutorial video that showed us how to convert the audio file to the frequency data array and then use that frequency data array to create a wave consisting of bars that grew and shrank according to the frequency data array. This video is listed in the Works Cited section.

3. Approach

Our approach was to create the scene using Three.js and then add the visual effects we desired to enhance the scene. For the scene, we thought of adding a stage, some stage lights, and some people in the audience. Additionally, we manipulate the background colors to our liking.

For the audio visualizer, because the tutorial video we watched already showed us how to create an audio visualizer consisting of bars, we felt it would be more appropriate to create a different form of audio visualizer so that our project contains more original work. So, we settled on creating a shader audio visualizer. By using a shader visualizer, we can use vertex shaders to manipulate the positions of its vertices in accordance with the audio frequency data. Similarly, we can also use fragment shaders to manipulate the color of its vertices however we like. Initially, we thought to also have the colors change in accordance with the audio frequency data, but this did not look as impressive as we thought it would. So, we kept the colors static. We thought that a shader visualizer would be perfect for our project because we could use the shader as the "stage screen," making our rave setting more authentic as there are usually visuals being played on a screen on stage. Additionally, by manipulating the amplitude of the shader waves, we can make the shader extend out into the crowd, creating a laserlight effect. Furthermore, to make our light show even more visually stimulating, we aimed to add more supplementary visual effects. Currently, this is in the form of fireworks that go off at set intervals in the background. Overall, we found our approach for the project to be satisfactory, as we feel like we have covered the fundamentals of a nightclub/rave setting, and we can always add more effects and features to improve upon this approach.

4. Methodology

Using the starter code, we started by removing the flower and land meshes. Then, we changed the static blue background to a dynamic slideshow of neon colors to engender euphoric feelings similar to those when attending a colorful concert or rave. The speed at which the background changes colors can be controlled with the "Speed" slider in the user GUI. Next, we introduced our own meshes into the scene. All of the meshes were implemented and displayed using the Three.js library. The stage and person meshes were found using a free website: sketchfab.com. The six light beams on the stage are made using Three.js cones, with a slightly transparent appearance. Once we had the meshes in place, we then implemented an audio-reactive plane mesh, along with a rudimentary "firework show" that periodically displays an exploding firework.

4.1. Meshes

The stage was loaded into the project as a .bin file, a .gltf file, and a texture file. We were able to render the stage by making a Stage object and attributing it to the .gtlf file, which then in turn calls the .bin file. One challenge that we encountered was the Stage object's inability to find the .gtlf file, and after making a copy of it in the root of the directory, it was able to render. We vaguely considered making our own meshes, but we quickly decided against it after considering the amount of additional time and effort. The focus of the project is audio visualization, not if we could make a stage mesh from scratch, so we downloaded a free mesh off of a website. After rendering the stage, a problem that we encountered was centering and resizing the stage. Initially

the stage was rendering behind the camera, so the user had to manually perform a 180 degree turn in the scene to access the stage. We solved this issue by changing the target in app.js to the center of the stage, so that when the website starts, the user is already looking at the stage in the proper position with an optimal viewing angle.

The audience consists of 15 person meshes that were also found on the free website sketchfab.com. This mesh was chosen due to its low triangle and vertex count. This was an important feature we were looking for in the person mesh due to performance as the size of the audience grew. These meshes were loaded into the scene in a similar fashion as to the Stage mesh. A Person object was created, and the .bin and .gltf files were accessed the same way. We had to consider a few ways of creating a large number of the same mesh to resemble an audience at a concert. One approach we considered was to create 15 different Person objects in the scene, and another approach was to use Three.js to create *shallow copies* of a singular Person object. These shallow copies would all point to the same Person object and conserve memory in the program. We decided to perform the former approach first, as it was the quickest way, then if performance is not sufficient, we would switch to implementing the shallow copies. However, after implementing the naive approach, there were no performance issues, so we decided to keep this approach.

The light beams shining down from the top of the stage were created using Three.js cone objects. We considered a few different ways of constructing these lights: we considered creating a Three.js Light object, and set its origin at the top of the stage. We decided against this approach, as we have not experimented with having a light originating inside of the mesh and thought that this may not produce the solid beam look for which we are looking. Another idea that we considered was implementing the cone along with a Three.js light shining through it to give it the appearance of a beam of light. We decided against this because we found that having the cone alone that is partially transparent gave us the exact look that we were going for. After deciding to only use partially transparent cones, we made 6 deep copies of the cone, resized each one to match the height of the stage, and repositioned each cone to originate from the six lights at the top of the stage. After changing the transparency property of each cone to 0.5, we reach our desired result.

4.2. Audio Visualizer

First, we needed to be able to play an audio file in our Three.js scene, and surprisingly, implementing this seemingly-simple feature came with its own set of difficulties. When researching how to implement this feature, most methods involved adding the audio to an index.html file. However, our Three.js starter code had no such file, so we had to figure out the appropriate file to include the audio. Initially, we included the audio in app.js. Upon realizing that the GUI is in SeedScene.js, though, we moved the code to SeedScene.js so that the user can use the GUI to manipulate the shader audio visualizer. However, the audio still would not play, and the console revealed the error: "*The AudioContext was not allowed to start. It must be resumed (or created) after a user gesture on the page.*" We wanted the audio to play the moment the web page is loaded in, but it seemed like we would not be allowed to. So, to work around this, we instead made the audio play after the user presses the spacebar. Currently, the application is set to play a specific song of our choosing for roughly one minute. Once the song ends, the

user will have to refresh the page to play the song again. This is a bug that we have not yet resolved due to it being low priority.

Once we were able to play a song, the next step was to extract the frequency data of the audio file so that we can use that data to create the audio visualizer. We did this using the AudioContext and AudioAnalyser classes from Three.js. These classes allowed us to retrieve the frequency data of the audio file. The AudioAnalyser has a getFrequencyData() method that returns an array, dataArray, containing the frequency data of the audio file. With this array, we were able to access the representative data of the song.

Furthermore, the dataArray, time, and amplitude data are passed as a "uniforms" parameter for the shader audio visualizer. Our custom vertexShader() and fragmentShader() functions are also passed as parameters for the shader audio visualizer. We scaled and repositioned the shader such that it would look like it is appearing from the stage screen, and we set it to look like a dynamic wireframe. We create a render function that updates the uniforms at that frame and thereby updates the shader each frame. The amplitude can be adjusted by the user via the GUI, so the user can manipulate the appearance and intensity of the shader audio visualizer.

Our custom vertexShader() and fragmentShader() functions are coded in Shaders.js. In vertexShader(), for each vertex, we update its z-coordinate value using sin functions and the frequency dataArray in order to create wave-like effects. In fragmentShader(), we set the colors of the vertices in relation to their x-coordinate positions and y-coordinate positions. The methodology for updating gl_Position in vertexShader() and gl_FragColor in fragmentShader() was completed with essentially pure trial and error until the shader audio visualizer looked impressive to us.

4.3. Additional Special Effects

After the stage and scene were set and the audio-reactive shader was introduced, one last step we took to emulate the 'real-life concert' feeling was to add fireworks. Instead of introducing an additional firework mesh, we decided to construct these fireworks utilizing Three.js's Point's class. This was chosen due to its relative simplicity, along with the fact that fireworks lend themselves very well towards being visualized by points or particles. Additionally, representing the fireworks in this manner does not lead to the performance of the program suffering. Both the firework itself, trailing up to the sky, and the subsequent explosion are all simply differently colored points.

In the Firework class, there exists a constructor and then four functions to reset, launch, explode, and update. The reset function is fairly self explanatory, resetting all existing fireworks on the screen. The launch function dictates where the fireworks launch from on the screen and where they travel to before they explode, choosing a random value for each of the x, y, and z values that then compose a 'from' and 'to' Three.js vector3. The color for the traveling firework is set to be approximately white. Next comes the explode function, which removes the initial firework point from the screen and replaces it with around 100 'explosion' points, all random colors within a certain specified RGB range. The radius of the explosion is decided using random coordinates for 'from' and 'to' vectors. Finally, the update function calls the other functions

sequentially (besides launch, which is called from the constructor) to start the entire loop. Once the fireworks explode, the 'explosion points' are gradually faded out and then ultimately reset, ending the life cycle of the firework. The firework show itself is started in the render loop, which creates a new firework instance randomly and continually splices and cleans the array that stores all of the instances.

The resulting visual effect is a simplistic but effective representation of what a firework show might look like in real life. We considered adding a rudimentary physics system to it, primarily gravity in order to make the particles fall to the ground as they would normally; however, we decided to leave this as perhaps a future application or job, as making the fireworks behave in accordance with real-life physics was a bit outside of the scope of our project. One current bug with the firework system is that the firework particles seem to be appearing in front of the stage even when the z-distance of the projectile is behind the stage. This is an issue to be looked into and resolved for a more realistic and immersive experience.

5. Results

This project's goal was to provide a reactive and aesthetically pleasing light show for the audience that emulates a concert or rave. The product that we have delivered has met this goal, as the stage has a reactive screen that responds well to the music. Specifically, as the music reaches a climax, the vertices on the screen begin to protrude further and at a fast speed. This achieves an appearance of the screen being "alive." We also wanted the user to be able to control some parameters of the visuals, so we provided sliders to control the amplitude of the reactive screen and the speed of the changing colors in the background. The user is also able to drag the scene and view the show from different angles. This user intractability was a priority, as the user will be able to control the virtual concert to a setting that pleases their eye. These results indicate that creating a virtual concert emulator can be aesthetically pleasing, and can be largely interactive.

6. Conclusion

The approach we took for this project served to meet our MVP goals along with providing a foundation for future work and further customization to be done. Our approach of using pre-set stage and person meshes and worrying more about the custom light effects helped us to focus on our goal and not waste too much time on tangential pursuits. This being said, a different (and equally viable) approach to the project would be to custom-craft the meshes that compose our show in order to have greater control over what the overall scene looks like. A more in-depth project (with time allowing) could essentially build a concert completely from scratch from the ground up, from the meshes to the effects. However, given the time constraints of our project, we decided to go with freely available meshes online that served our purpose suitably well.

Ultimately, completing this project gave us a much better sense of how to efficiently utilize Three.js and its various libraries in projects of our own choosing. Previously in class, we had predetermined projects with scenes and environments that were already set up; for this project, however, we had to select meshes and build our scene ourselves. Whereas these other projects had given us the tools we needed to understand specific crucial aspects of computer

graphics, the freedom of this final independent project allowed us to explore in-depth a topic we are all interested in and provided us with the tools we will need to go about creating similar projects in the future.

7. Future Work

An issue that we noticed in our product is that the stage mesh that we loaded into the scene is only partially colored. The texture files provided in the stage were only partially used, particularly on the speakers and podium, but the remainder of the stage is not textured and merely appears as black. We are unsure of why the stage was only colored in certain areas and not colored in others. Due to time constraints, we were unable to investigate this issue, so future work could include resolving this problem to display a more colorful, aesthetically pleasing stage.

Another issue that we encountered was that the fireworks appear on the screen in front of all of the other mesh in the scene. This can visually look acceptable, but realistically, fireworks at a concert will appear behind the stage. Due to time constraints, we were unable to resolve this issue, but we suspect that it is related to the fireworks not being a mesh and the stage being set to be behind the cones and all other meshes in the scene. Future work could include resolving this issue to provide the user with a realistic concert experience

Additional future work can include making the audience more colorful and realistic. Currently, the person meshes that make up the audience lack color and detail. We hope to make the audience represent a realistic concert setting, as people are dressed in colorful clothing and are generally happy and dancing. Making the audience move to the music is also a stretch goal. In addition to the stage scene being reactive and dynamic, a productive addition would be to make the audience dance to the music. A basic idea that we had was to make the audience jump up and down to the music, and depending on the pace of the music, the audience would jump higher. Finally, future work can include making the six light cones change color with the music. We decided against changing the color of these cones because the reactive screen is sufficiently colorful and dynamic. We thought that the overlapping of these two aspects would visually overload the user. Future work can implement a reactive light cone feature without making the user overly stimulated.

Lastly, for our MVP, we only cared to be able to deliver a “performance” to one song. However, it would be ideal to give users the ability to upload an audio file of their choosing so that they can see how the shader audio visualizer reacts to their selected song.

8. Contributions

Darren Zheng was responsible for importing and implementing the person meshes along with the stage mesh. Additionally, Darren created the light cones that shine down from the stage. Daniel Huynh created the audio visualization shader that reactively moves based on the frequencies of an audio file being played. Lucas Maake implemented the fireworks show that displays throughout the ‘concert.’ All three members documented their implementations and approaches in this writeup, along with contributing equivalent amounts to the rest of the sections. We would like to thank Caio Costa for his assistance as our point TA. Caio helped us with bugs that we encountered and gave us pointers on how to navigate multiple implementations to optimally achieve our goals.

Works Cited

"JavaScript Audio CRASH COURSE for Beginners." *YouTube*, youtube.com/watch?v=VXWvfrmpapI&ab_channel=Frankslaboratory.

"Sketchfab - the Best 3D Viewer on the Web." *Sketchfab*, Sketchfab.com.

"Three.js – Javascript 3D Library." *Threejs.org*, 2019, threejs.org/.