– COS 426 Final Project May 2020 –

Dream World: Meditation Simulation

Lauren Johnston, Joanna Kuo, Elizabeth Petrov, & Lydia You
Professor: Felix Heide

Abstract

This project entails the creation of a Javascript-based world simulation for focused meditation. We hypothesized that we could facilitate the meditation process by designing a dream-like world in which users can customize their own calming terrain through the perspective of a flying bird while listening to peaceful music and guided meditations. With the COVID-19 pandemic affecting more people each day, we wanted to explore ways to remain calm and reduce stress through meditation. We were successful in our goal of creating this simulation, and in particular achieved our target results: implementation of infinite terrain generation, realistic bird movement, and music and terrain coordination.

1. Introduction

The goal of this project is to create a web-based graphics simulation that facilitates meditation. Though there are many forms of meditation, it can broadly be defined as a set of practices designed to improve well-being and stabilize emotions [8]. Meditation has been shown to fundamentally alter brain activity associated with anxiety and increase cortical thickness [10]. Furthermore, meditation practiced for a period of at least four days can improve executive function and working memory [13]. There are two key types of meditation that researchers have widely studied: "focused meditation", or meditation centering around a specific object, and "open monitoring meditation" which involves taking account of one's mental state in a non-judgmental way [11].

In this project, we focus on the former type of meditation: focused meditation around an object. We hypothesized that meditation around an object (like a graphics simulation) can reduce the barrier to individuals who may be daunted by the task of an open-meditation with no center of focus. Despite the popularity of meditation apps like Headspace, many people feel like their guided meditations are too boring in a world defined by attention-grabbing digital interactions [1]. Therefore, we hypothesized that we could create a middle ground to ease individuals into the meditation process through a simulation-based focused meditation. Previous graphics simulations have focused on random-noise terrain generation and rendering computer-generated worlds. We wanted to combine some of these techniques using ThreeJS tools to create an immersive experience to assist and enhance meditation. Our Dream World can be used by anyone, particularly those who are stressed and need to immerse themselves in a soothing, meditative experience.

2. Methodology

Bird

To accomplish the meditative aspect of our project we decided to view the world through the perspective of a bird. We reasoned that this angle would allow users to explore the entire world as well as relax by watching the bird fly continuously in the infinite space.

2.1. Bird Mesh

We added birds into our scene by downloading glb files of three different birds — a stork, parrot and flamingo [2] (Figure 1). These files were then converted to gltf files using THREE.js's GLTFLoader class. To animate the bird, we used an AnimationMixer to loop through each bird's animation clip and used the difference in timestamps in the update function to calculate the duration of each frame (which equates to the bird's wing speed). The bird type can be switched using the GUI.

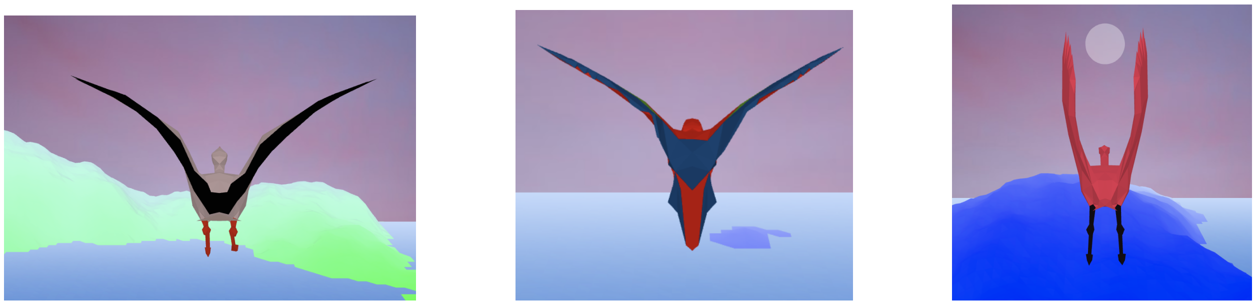


Figure 1: Bird options: stork, parrot, and flamingo (from left to right)

2.2. Bird Movement

2.2.1. Keys The birds are controlled using the w-a-s-d keys, where w is up, a is left, s is down and d is right. We attached two event listeners to detect when a key is pressed and when it is released. When a key is pressed an array called keysPressed uses the event's keyCode to set its value to true. Then the function birdHandler() is called to determine which key was pressed and execute the appropriate action on the bird. Finally, when the key is lifted the value at the event's keyCode is set to false. By saving the keys pressed in an array it allows multiple bird actions to execute at the same time; for example, if you want to make the bird turn right and go up at the same time.
2.2.2. Wing Speed The speed of the bird is dependent on its velocity, where the higher the velocity the faster the wing speed. However, the wing speed remains constant at velocities below 2 to simulate the bird flying in place, as well as above 4 to ensure the bird is flapping at a plausible rate. When the w key is pressed, the speed of the bird's flapping is increased whereas when the s key is pressed, the flapping speed is decreased. However, in addition to the decreased wing speed, we also edited the bird's animation track to simulate it soaring through the world. Each bird has a NumberKeyframeTrack that is an equivalent 14x13 2-D array where each column corresponds to a specific animation frame / morphTarget (located in the bird's morphTargetDictionary) and the rows are the number of frames in the clip. Each row contains all 0's except for the frame that is going to be displayed which is set to 1. By first looking at the 13 different frames of each bird and determining which frames best represents the bird soaring and flapping, the corresponding 1-D (0 - 181) indices in the values array are set to one. Because the number of morphTargets were different for each bird, different indices were set to 1 in each of their NumberKeyframeTracks.
2.2.3. Rotation The bird was also made to rotate about its origin using its XYZ coordinate system. When the w or s key is pressed, the bird can rotate between -0.5 and 0.5 radians about the X axis. On the other hand when keys a or d are pressed the bird banks by rotating about its Z axis between -0.5 and 0.5 radians as well as rotate between 0 and 2 about the Y axis. After a w-a-s-d key is pressed, the timestamp of the event is recorded. In the update function, if the key's timestamp plus 1000 is

less than the current timestamp, the bird is slowly rotated back to its original flying position. For example if the s was pressed and let go, if its timestamp + 1000 is less than the current timestamp, then the bird returns to its original animation and is rotated back to 0 in the X direction.
2.2.4. Position We chose to keep the bird at the origin and simulate movement by instead moving the terrain to avoid floating-point errors (if the simulation ran for very long, continually adding offsets to the bird's position could create long floats). However, this meant we had to keep track of the bird's "position", which was calculated based on its velocity and rotation to determine how fast and in which direction the terrain should be moving. This was done by setting the Bird's parent's state's XYZ coordinates to the position of the bird, which was then used by the Terrain to appropriately shift the world underneath.

2.3. Camera Attachment

To put the user in the perspective of the bird, we attached the camera behind the bird and made it rotate with the bird as well. We accomplished this by passing in the camera into the scene and to the Bird class. By doing so we were able to update both the bird and camera at the same time. The camera's XYZ positions were updated based on the sin() and cos() of the bird's X and Y rotations. Because we made the bird stay at the origin, the camera's lookAt() property was set to the origin.

Terrain

Our goal in the terrain generation portion of our project was to allow the user to visually control the world around them and to allow them to create their Dreamworld that goes on forever. Hence, we wanted to give the user as many customization features as possible, which are discussed below. In terms of the terrain appearance, we were aiming for a polygonal, non-realistic, colorful terrain. It also matches the geometric look of the bird, and helps save processing power.

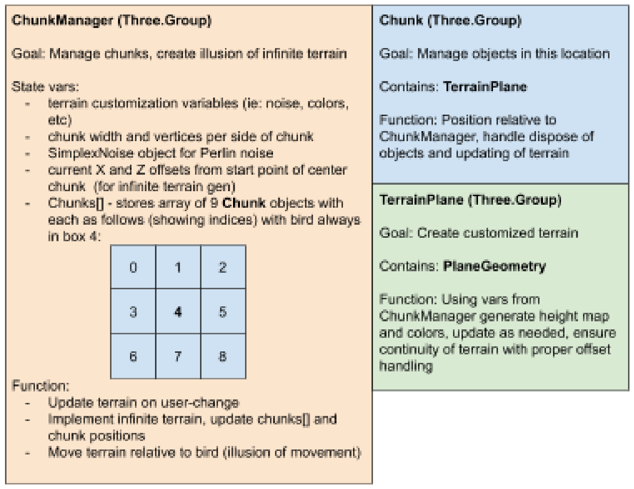
The terrain is created using a Perlin- noise- generated height map that is then manipulated according to user-set variables and then applied to the vertices of a PlaneGeometry object. Each face is then iterated through and set to a specific color (discussed below).

To create an infinite terrain, we used a procedural generation method that we came up with ourselves that uses a "ChunkManager" that controls a 3x3 matrix of "Chunk"s that are made up of the terrain, called "TerrainPlane". This class overview is seen in Figure 2 below.

2.4. Terrain Noise Generation

The main component of terrain implementation is how it is randomly generated using Perlin noise. In our implementation we use a SimplexNoise package with a given random seed. Simplex noise is similar to Perlin noise, but has fewer directional artifacts and a lower computational overhead in higher dimensions. We find the height map of dimension $[numberverticesperchunk]^2$ by feeding in each index location, accounting for the offset to the octave() function, which calculates a value [0, 1]. The octave function is highly adapted from a starter tutorial we followed [3]. In short, it layers a given number (called 'octaves') of frequencies together, and uses simplex noise in 2D to get a randomized terrain. By increasing the starting frequency, you can get a much more peak-full terrain. By decreasing the octaves, you get a smoother terrain. In this way, changing the SimplexNoise seed gives you an incredible diversity of terrains.

After creating a height map, the PlaneGeometry vertices are iterated through and set to the corresponding height map value. This value is then edited according to other variables, such as


ChunkManager (Three.Group)

Goal: Manage chunks, create illusion of infinite terrain

State vars:
- terrain customization variables (ie: noise, colors, etc)
- chunk width and vertices per side of chunk
- SimplexNoise object for Perlin noise
- current X and Z offsets from start point of center chunk (for infinite terrain gen)
- Chunks[] - stores array of 9 **Chunk** objects with each as follows (showing indices) with bird always in box 4:

0	1	2
3	**4**	5
6	7	8

Function:
- Update terrain on user-change
- Implement infinite terrain, update chunks[] and chunk positions
- Move terrain relative to bird (illusion of movement)

Chunk (Three.Group)

Goal: Manage objects in this location

Contains: **TerrainPlane**

Function: Position relative to ChunkManager, handle dispose of objects and updating of terrain

TerrainPlane (Three.Group)

Goal: Create customized terrain

Contains: **PlaneGeometry**

Function: Using vars from ChunkManager generate height map and colors, update as needed, ensure continuity of terrain with proper offset handling

Figure 2: Chunk Class Overview

setting it to the power of a value (this creates valleys), multiplying it by an exaggeration factor (this exaggerates the features), or rounding the value for a terraced look. We multiplied the entire variable by 10 (chosen arbitrarily) to further make the terrain features noticeable. These terrain customizations are adapted from a very good overview we found online [9]. To implement water, if a vertex value after manipulation was less than the water level, it was set to the water level, making the water seem flat. The user can set all these variables to customize the world. An overview with can be seen below in Table 1:

Variable	Description	Low Value	High Value
"Jaggedness"	# octaves of frequency added to get final value	Smooth	Jagged
"Peaks"	Start frequency when going through each octave	Flatter	Hilly
"World Seed"	SimplexNoise random seed	NA - generates dif world	NA - generates dif world
"Exaggeration"	Amount height map is multiplied by. Exaggerates features	Flatter	Intensified
"Valleys"	Power height map value is raised to. Creates flatter areas	More chaotic	More valleys, exaggerated mountains
"Water Level"	Every height below this value is considered water	More land features	More islands
"Terraces"	Creates terraces using the height map values [1]	More plateaus	More levels

Table 1: Terrain Customization Variable Overview

The terrain geometry is created in the TerrainPlane class, the noise-based height map specifically in generateTexture(), and the final terrain heights in updateTerrainGeo().

2.5. Terrain Color

The terrain uses flat shading, with some clever tricks to make it graphically appealing. Our goal with colors was not to be realistic, but match the polygonal aesthetic and allow for wacky "dream-like" color schemes while saving computational space. We use 4 colors - for water, bank (land nearest to water), middle (center of terrain feature), and peak. To set the color, we loop over each face in the terrain geometry and find the average height of the vertices. If this value is greater than a certain value (which we set to the exaggeration var * 7, as before we multiplied the height by 10), we consider the face a peak and set it to the peak color. Otherwise, we find the percentage it is up the mountain/feature. If it is greater than the user-set variable middleGradient (called "Peak Color Height" in the GUI), then the percentage is adjusted and the color is the appropriate interpolation of the middle and peak colors. If less than middleGradient, it is the interpolation of the bank and middle colors. This creates a beautiful 3-color adjustable gradient. Water is set to the water color. To create some semblance of texture, we add a random offset to each RGB color channel that the user can set the intensity of through the “Color Texturing” variable in the GUI. Table 2 gives an overview of color customization:

Variable	Description	Low Value	High Value
“Color Texturing”	Random offset intensity for each face, adds fake texture	At 0 - just gradient	Textured look
“Peak Color Height”	Gradient position of middle color	Peak color dominant	Bank color dominant
“World Seed”	SimplexNoise random seed	NA - generates dif world	NA - generates dif world

Table 2: Color Customization Variable Overview

2.6. Infinite Terrain Generation

Many games, such as Minecraft, generate the terrain block by block to save computational space and create an illusion of an infinite world. In Minecraft’s case, the player stays at the origin and the terrain moves about the player and loads chunks as needed. We attempted to implement something similar. We implemented a 3x3 Chunk grid that is controlled by ChunkManager, and the Chunks are shifted and destroyed/created as needed.

ChunkManager first initiates the 9 chunks, with 8 being generated according to a pixel offset related to the pixel width of each chunk (chunk 0, for example, is at (x offset - px width, z offset - px width) (change in sign due to Three.js axis orientations)). This offset is used in the chunk’s terrain to accurately generate a continuous chunk using the noise function. As long as the noise seed is the same and offsets are correct, the terrain should be continuous. If you fly away from a point and return to it later, it should have the same terrain.

A big challenge we ran into is terrain continuity. To generate continuous terrain, an offset needed to be applied to the noise function calculation. As each terrain is made up of a double array of vertices, the end row/col of one chunk would overlap with the next. In calculating the noise, we needed to calculate the current index value in relation to the 3x3 grid, which has 3*(verts along each chunk) - 3 vertices per side (as chunks overlap). Through a lot of trial and error, scribbling array grids on paper, and hours of debugging, a solution was found. The following equations lead to the proper noise, correctly accounting for the offset of each chunk and axis orientation. n_x and n_y are later fed to the noise function, i and j are the indices of a specific vertex in the chunk, and the chunk offset is dictated by xOffset and zOffset:

$$n_x = (i - chunkVertWidth - xOffset + 1 + floor(xOffset/chunkVertWidth))/(totalVertWidth - 3)$$

$$n_y = (j + chunkVertWidth + zOffset - 1 - floor(zOffset/chunkVertWidth))/(totalVertWidth - 3)$$

To visualize the effect this equation has, here is a comparison to just adding the x and z offsets to the i and j values and not using the floor function with a coloring scheme to make this more visible. As seen, the floor function helps ensure proper edge alignment.

The next challenge was destroying/loading chunks as needed. It is worth noting that because ChunkManager moves in relation to the bird (with the bird staying at the origin), this complicated things a bit as it added extra variables to contend with. We solved this by looking at the x, y, z virtual position of the bird (set in the Bird class). If this position was at the edge of the center chunk,

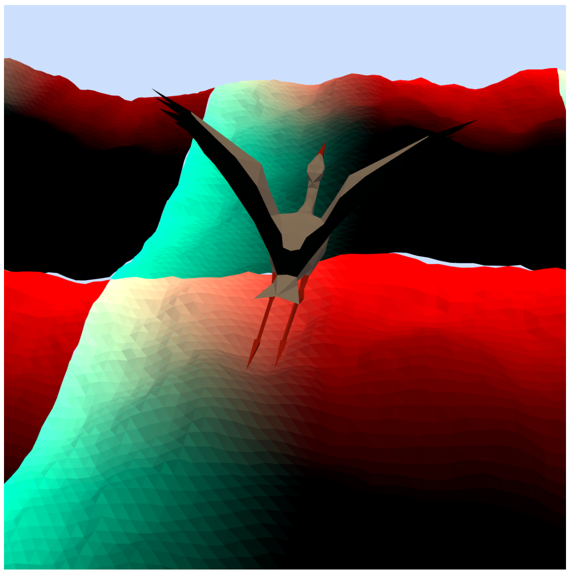


Figure 3: With only adding offset, no floor: Gaps seen

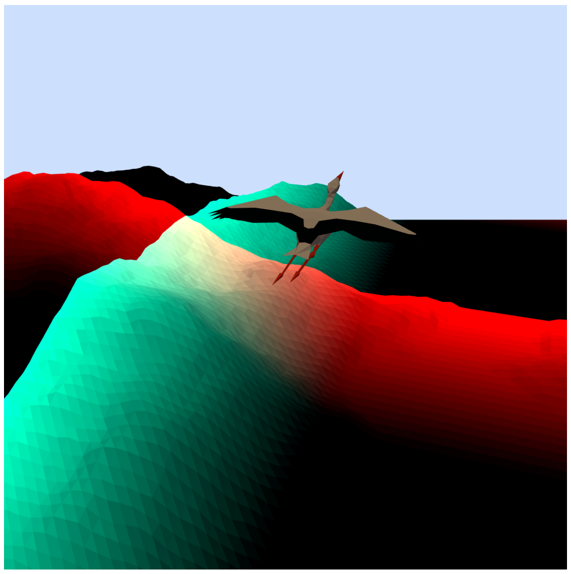


Figure 4: With proper alignment: No gaps between chunks

we shifted the chunks accordingly, disposed of the 3 unneeded chunks, and generated 3 new chunks with the correct offsets. The ChunkManager was then hopped so that the bird was on the opposite edge that it crossed, and the directional position of the bird was hopped back as well (both by the chunk pixel width, but in opposite directions as the terrain and bird move opposite relative to each other). Figure 5 below shows a visualization of our process.

This is adapted for each of 4 directions (+/-x, +/-z). An insidious bug we ran into was a visual "glitch" in which the next row of chunks being generated appeared at the origin before moving into its correct position. We were ultimately able to solve this issue by updating the ChunkManager's Terrain twice in the render loop to account for this position correction. Overall, our implementation creates the illusion of an infinite terrain, handles memory correctly, and creates continuity of chunks by properly keeping track of the offset. This is implemented with a modular approach through the ChunkManager, Chunk, and Terrain classes.

Clouds

To make the scene feel more complete, we added volumetric clouds to each chunk. Having volumetric (3D) clouds means that the user has the option of flying through the clouds. In order to accomplish our goal of creating volumetric clouds, we wanted to ensure that we could generate a large number of clouds without slowing down the simulation. After doing some research, we determined that the best way to do this was to create an InstancedMesh for the cloud that allows us to specify geometry, material, and count (number of instances). So, we would create one Cloud geometry for each InstancedMesh that could be randomly translated to any point on the sky above a certain y level. See Figure 6 below for a picture of a single instance of a cloud. Ultimately, we chose to include multiple InstancedMeshes to accommodate the need for variation in cloud shapes.

The geometry of the cloud that you see in the image above was created by merging four IcosahedronGeometry objects. The material of the cloud is a MeshLambertMaterial object with a texture map that was created by mapping the vertices of the cloud geometry to a custom texture created in

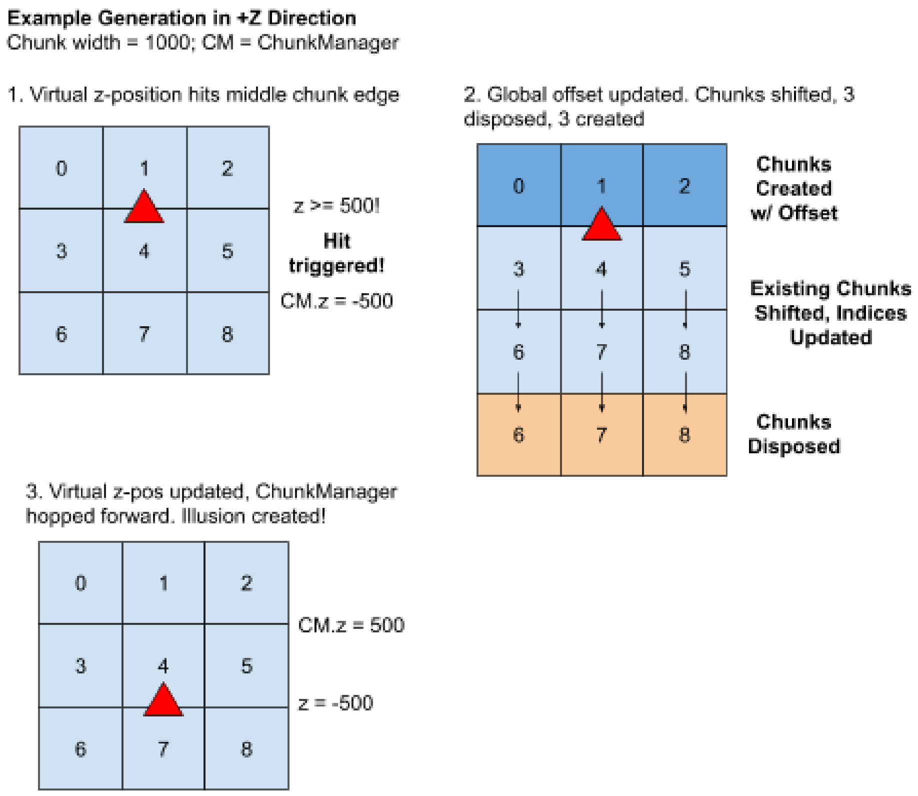

Example Generation in +Z Direction
Chunk width = 1000; CM = ChunkManager
1. Virtual z-position hits middle chunk edge
0
1
2
3
4
5
6
7
8
z >= 500!
Hit triggered!
CM.z = -500
2. Global offset updated. Chunks shifted, 3 disposed, 3 created
Chunks Created w/ Offset
Existing Chunks Shifted, Indices Updated
Chunks Disposed
3. Virtual z-pos updated, ChunkManager hopped forward. Illusion created!
CM.z = 500
z = -500

Figure 5: Sample terrain generation

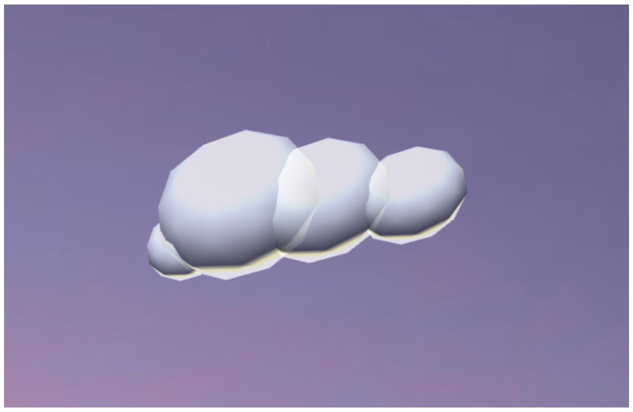


Figure 6: Sample cloud instance

the Figma design tool and also partial opacity (alpha = 0.75). Using a Lambertian material allows for the cloud to be impacted by lighting in the scene.

An alternative approach to creating clouds would've been to procedurally generate them like the chunks. However, we felt that using an instancedMesh for better performance was more important than procedurally generating them. That way, more computationally expensive operations could be reserved for the terrain.

A future extension of this is to add noise to the vertices of the cloud geometry though this was out of scope for the project. The challenge of adding noise to vertices is that it must be smooth (so edges are not jagged) and also that it is computationally expensive to use noise for each time the cloud instancedMesh is created.

Lighting

The lighting, while not an extremely technical part of the project, was particularly tricky to configure in order to make the scene look relatively realistic and not over-saturated. While our intent was not to make the scene look entirely realistic – we wanted to preserve some level of stylization to enhance the dream-world effect – we still wanted to make the lighting soft and relatively natural. To this end, we decreased the intensities of the Hemisphere and Ambient lights in BasicLights.js, which helped to reduce the lighting intensity on the entire scene, and added realistic power and decay parameters to the lights. We then added a light right above the bird, at the position the camera was attached, to illuminate the bird.

2.7. Sun

An additional visual effect we wanted to create was a rising and setting sun. In order to do this, we created a WorldLighting.js class comprising of two DirectionalLights (pointed at the bird), with sphere meshes over them, to represent the sun and the sun "aura". We then made the sun rise and set by constantly rotating the entire object along an axis, ensuring that the radius of rotation was large enough so that the setting sun was able to be not rendered by the camera and "disappear" behind the terrain.

2.8. Sky

We originally wanted the sky to be a skybox comprised of a MeshLambertMaterial and texture so that the sunlight would be able to appropriately interact with it, but realized that it slowed down our application significantly and was not worth the visual effect. We also tried Color-Tweening to slowly lighten the sky in conjunction with the sun position, but weren't satisfied with the results. To this end, we simply uploaded three sky images as textures onto the background of the SeedScene, with the user being able to toggle between "Starry", "Dusk", and "Sunset" skies, and also added environmental presets, such as "Mars".

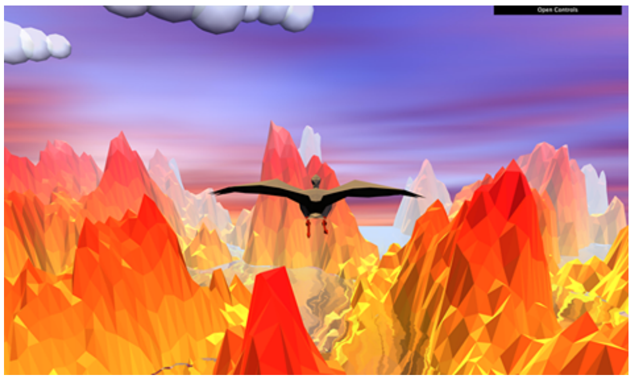


Figure 7: The "Mars" preset

Audio

A key component of our application is the audio component, as it allows the user to engage another sense and immerse themselves further into the Dream World. We created a Music.js class to handle all audio, and attached an AudioListener with a corresponding Audio object to the camera. We then added event handlers so that the audio would pause and start when ’p’ is pressed. The user can also choose amongst 5 audio tracks, one of which is a guided breathing exercise. Sources of the royalty-free audiotracks linked here: [4], [5], and [6].

2.9. Audio-Terrain Visualization

A technical extension of the infinite terrain generation we chose to pursue is to modify the terrain based on the playing audio track’s average frequency. We believe that the terrain responding to the music helps create a cohesive meditative experience and enhances the soothing effect of the simulation. Thus, we used an AudioAnalyser object to “listen” to the active sound being played. Then, in the update function of our Music object class, we determine the average frequency of the sound being played (if there is a sound being played) and map average frequencies to terrain exaggeration and color changes based on the magnitude of the frequency.

Quotes

To increase meditative thoughts, we added a slideshow box at the bottom of our application to display meditative and inspirational quotes from Headspace, a company that specializes in meditation and mindfulness [12]. The slideshow was implemented using HTML elements and adapted from the slideshow demos from the W3.schools website[7]. In a class called Text, a container, two arrow buttons, and a quote element was created. CSS was used to stylize the buttons and container, and a timer was set in the auto() function to change the quote text every three minutes. A constant array of quotes was created in a separate file called Quotes.js for modularity purposes and was imported into the Text.js file. When the arrow buttons are pressed, a function called plusSlides() takes in an integer to determine the index of the new quote to be displayed.

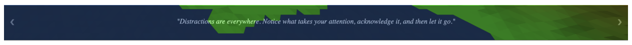

"Distractions are everywhere. Notice what takes your attention, acknowledge it, and then let it go."

Figure 8: Slideshow

3. Results

The main goal of our project was to create a world that was calming, customizable, and visually appealing for the user. On the technical side, we wanted to explore the challenges of infinite procedural generation with efficient frame rates. We believe we accomplished these goals well, particularly in the terrain generation aspects. After our initial MVP, we changed our code dramatically, working hard to fix visual glitches and to speed up our generation through efficient memory management and modularization. Furthermore, we implemented several features designed to enhance the meditative experience of the user, including floating text, procedurally-generated clouds, calming audio tracks, affirming quotes, and natural lighting. To test our product, we ran the simulation for 30+ minutes with different terrain parameters to ensure the infinite terrain generation was, in fact, infinite.

4. Discussion

We decided to utilize the extensive visual tools that Threejs provides to build our simulation, both for aesthetic quality and also good modularity and style. We were very satisfied with our end-result: a customizable, smoothly generating, infinite terrain with many appealing supplements to facilitate meditation and calm. Along the way, we had to make a number of important design and implementation decisions: would we be moving the bird around the terrain, or vice versa? How would we actually generate the plain - by moving previous chunks, or generating new ones as we go? How do you efficiently generate chunks without compromising smoothness? In the end, we tackled all these problems with a modular approach utilizing multiple classes, creating separate Chunk, ChunkManager, Terrain, and TerrainPlane classes, not to mention our Bird and other environmental variable classes. Even so, in the future, we would opt for even more modularity, and we'd be extremely careful about what objects we're updating in the render loops, as doing unnecessary checking and updating slows down the application considerably. There's a breadth of follow-up work that could be done, such as implementing collision detection, removing small visual "glitches" that occur when the bird moves left/right, or implementing textures and foliage on the terrain.

We learned so much over the course of this project. On the technical end, we became well-versed in noise-based-terrain-generation, memory management and general usage of Threejs, CSS, HTML, manipulation of meshes, and a lot of debugging technique. In addition, just the actual prospect of implementing a full graphics project from start to finish with four members was a profound learning experience, and gave us invaluable engineering and collaborative skills.

5. Conclusion

As acknowledged before, we believe we have successfully attained our goal of implementing a calming, infinitely-generating Dream World to facilitate meditation. Some next steps would include taking our Dream World into VR to allow users to follow immerse themselves into this digital world and enhance their relaxation. Furthermore, we really wanted to look into implementing more realistic lighting, perhaps through raytracing, but understand that raytracing a dynamic terrain would be enormously expensive in terms of computation power.

6. Contributions

To accomplish this project we had to divide up the task among the four of us. Joanna was assigned implementing the bird and quotes, while Elizabeth completed the terrain generation. Lauren made the clouds and Lydia worked on lighting and audio. Everyone contributed to debugging, optimizing the terrain generation, and adding polishing effects.

7. Acknowledgements

We wanted to acknowledge the helpful feedback we received from TAs Lucas, Reilly and Will that was instrumental to our success. We also want to thank Ethan (Grad. TA) for gathering and sharing proposal feedback with us.

8. Honor code

We pledge our honor that we have not violated the honor code in the writing of this paper or implementation of this project.

/s/ Lauren Johnston
/s/ Joanna Kuo
/s/ Elizabeth Petrov
/s/ Lydia You

References

[1] [Online]. Available: https://www.theringer.com/tech/2018/10/25/18010314/meditation-headspace-insight-timer-app

[2] [Online]. Available: https://discoverthreejs.com/book/first-steps/load-models/

[3] [Online]. Available: https://medium.com/@joshmarinacci/low-poly-style-terrain-generation-8a017ab02e7b

[4] [Online]. Available: https://www.fesliyanstudios.com/royalty-free-music/

[5] [Online]. Available: https://www.mindful.org/audio-resources-for-mindfulness-meditation/

[6] [Online]. Available: https://www.bensound.com/royalty-free-music/track/slow-motion

[7] [Online]. Available: https://www.w3schools.com/w3css/w3css_slideshow.asp

[8] R. Cardoso, E. de Souza, L. Camano, and J. R. Leite, "Meditation in health: an operational definition," *Brain research protocols*, vol. 14, no. 1, pp. 58–60, 2004.

[9] R. B. Games. [Online]. Available: https://www.redblobgames.com/maps/terrain-from-noise/

[10] S. W. Lazar, C. E. Kerr, R. H. Wasserman, J. R. Gray, D. N. Greve, M. T. Treadway, M. McGarvey, B. T. Quinn, J. A. Dusek, H. Benson *et al.*, "Meditation experience is associated with increased cortical thickness," *Neuroreport*, vol. 16, no. 17, p. 1893, 2005.

[11] A. Lutz, H. A. Slagter, J. D. Dunne, and R. J. Davidson, "Attention regulation and monitoring in meditation," *Trends in cognitive sciences*, vol. 12, no. 4, pp. 163–169, 2008.

[12] A. Puddicombe. [Online]. Available: https://www.headspace.com/meditation/quotes

[13] F. Zeidan, S. K. Johnson, B. J. Diamond, Z. David, and P. Goolkasian, "Mindfulness meditation improves cognition: Evidence of brief mental training," *Consciousness and cognition*, vol. 19, no. 2, pp. 597–605, 2010.