Life Flight

Pierce Maloney, Mitchell Cooper

Abstract

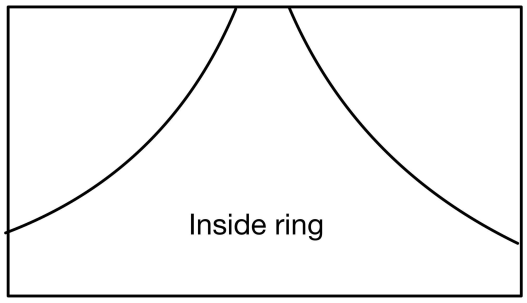
This project is a Javascript web application that simulates an airship exploring infinitely generated terrain. The terrain is on the inside edge of a ring and grows in complexity as the ship approaches, starting as flat and colorless upon generation, and ending as mountainous, colorful, and populated with trees. We were able to achieve the fundamental goals of this project: infinite terrain generation, movement along a ring, user interactivity and a smooth increase of world complexity, with the next step being to add more complexity to the terrain as the airship approaches.

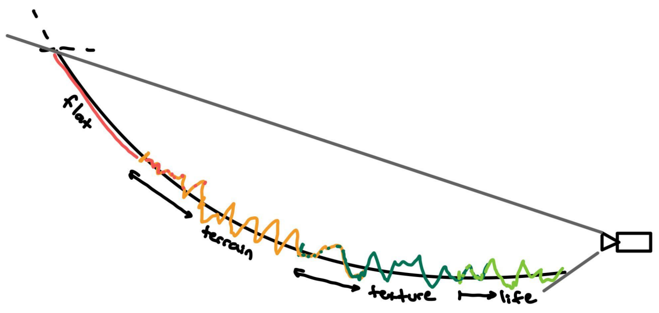
Introduction

The purpose of Life Flight was to develop an interactive computer graphics simulation that is aesthetically pleasing and endless. We drew inspiration from games and other graphics projects that we enjoyed, discussed below.

Goal

Our vision for this project was to create an infinite terrain generator that grew increasingly complex as the terrain came nearer. Early on we decided that the terrain curve upward in the distance and be bounded to the left and right simulating the inside of a ring in space like the popular game Halo. Below are two concept images we drew up while brainstorming project ideas.





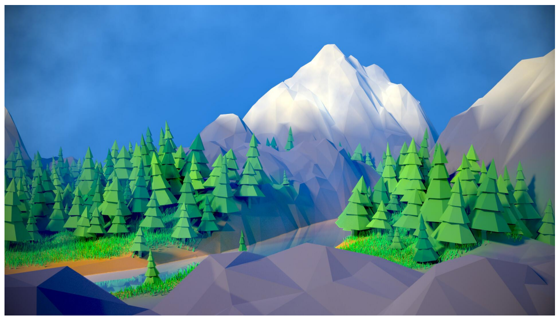
Our specific goals for this project were to create an infinite terrain generator in a ring shape that gets increasingly more complex and could be modified by user input. One of our stretch goals was to create a Google Chrome extension so that our project could be used as the background of the Google search page.

Previous Work

This project was inspired by the Fractal Brownian Motion portion of the ray tracing assignment as well as the Dreamworld project from the 426 project examples and the loading screen from Halo: Infinite which is a pre rendered planetary ring in space. One of the core aspects of this project is simulating moving over infinite terrain. There are multiple methods that have been used to simulate movement of the terrain. Some methods use a single mesh and adjust the height of the vertices using a noise function that depends on time (which creates the illusion of mesh movement), and some methods create multiple meshes, adjust the height of the vertices during construction, then stitch them together and move them within the scene. The first approach works well when the camera is stationary within the scene, because the camera cannot be moved to view areas beyond the mesh, but it does not work well when the camera is given free roam because the camera can move away from the mesh. This is where the second approach is useful. As the camera moves in a direction, a new chunk of terrain can be generated, much like the popular game Minecraft.

Two images that sparked inspiration:

The Halo Ring from Halo: Infinite


Terrain from u/lumenwrites on Reddit

Approach

We chose to use a chunk based approach rather than a single mesh. Our first step was to create a single chunk that contained a mesh resembling natural terrain. We then added multiple chunks together with an offset in position and in the noise function to create a larger terrain that was continuous with the initial chunk. Next, we moved each of the chunks in unison to simulate motion. To create the effect that the chunks were part of a large ring, we then adjusted the vertical position of each chunk and tilted it toward the center of the ring.

Methodology

Coordinate System

Our project extensively translated the positions of vertices within meshes, and translated and rotated meshes within scenes in a non linear manner. For this reason, we felt that it was important to include a brief explanation of the coordinate systems within the project. For the seed scene that all chunks are located in, positive `x` moves straight away from the camera, positive `z` moves from below the camera to above the camera intuitively, a larger `x` means that a chunk is further from the camera and a larger `z` means that the chunk is higher above the camera. Each chunk contains a mesh that represents its terrain. This mesh starts out as a plane segment before its vertices are scaled. The center of this plane segment represents the position of the mesh and therefore the chunk within the scene, and also serves as the origin of the mesh coordinate system, with the axes being the same as the seed scene before the mesh is rotated.

Chunk

Chunk is a class we used to contain all the data for a section of the terrain as well as the methods to update it.

Terrain Generation

To generate interesting, non-repeating terrain that will stitch together between chunks, we used Simplex noise combined with Fractal Brownian Motion to generate realistic, interesting terrain. The FBM compounded the simplex noise at varying frequencies and amplitudes, as described in lecture, to improve the look of the terrain. By using the Simplex `noise2d` function to generate the noise, stitching the chunks together was made easy because the noise corresponded to specific `x` and `y` coordinates, which we could just change between chunks so the noise would align. This allowed us to generate infinite terrain.

After creating the noise, applying FBM, we then took the x and y coordinates of vertices and “jittered” them, displacing them by a random amount from their original position. This made the terrain look much more natural, instead of the gridlike look that `PlaneGeometry` generates. A problem that this posed, however, was that when stitching chunk meshes together, the random jitter would be different for vertices on the edges. We solved this by conditioning the jitter on whether or not is it on the edge, using the conditional to only jitter if:

```
(vertex.x + width / 2) % width != 0 && (vertex.y + height / 2) % height != 0)
```

A few small changes in addition to the jittering that improved our terrain generation included: Setting a water level, and clamping all heights for vertices below the water level to be at water level; Scaling the peaks of mountains using a small exponential term to increase their steepness; Dropping the `z` values on the far left and right of the mesh to 0, which created a fuller appearance of the mountains on the edges; Decreasing the height of the mountains near the middle of the ring and increasing those on the sides in order to create a subtle concave path for the airship.

Colors

We use six main colors to color the terrain:

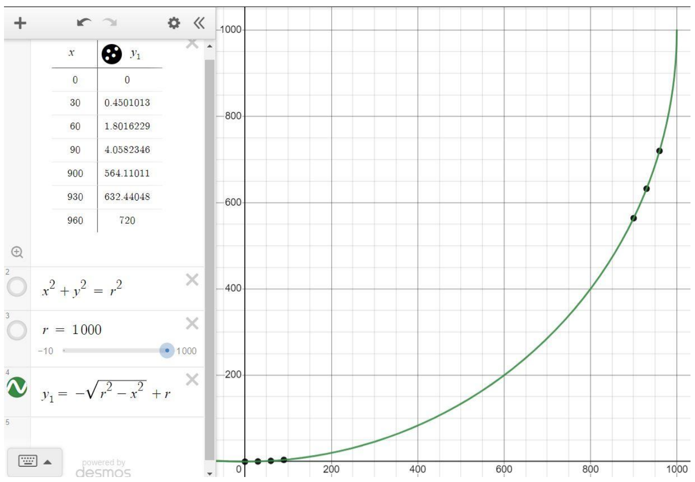
Name	Hex	Description
`Flat`	`0x777777`	Flat terrain in the distance
`Water`	`0x44ccff`	Any triangle at or below a certain water level
`Beach`	`0x483c32`	Colored triangles just above water level
`Grass`	`0x356520`	Between the beach and cliffs, interpolated into `cliff` for effect
`Cliff`	`0x335577`	Cliff color, interpolated from `Grass` and to `Snow`
`Snow`	`0xcccccc`	White for snow-capped mountains

In order to speed up computation time, the colors for the terrain are decided as soon as the Chunk generates and are based on the height map of the Chunk's vertices. For each face on the mesh, the max height of its three vertices determines the face's color. There are explicit values for which colors correspond with which heights, except for ranges of values where color interpolation takes place. The implementation of smooth color blending via color interpolation was a very positive decision that improves the look of the terrain. Before the interpolation, the lines between grass and the cliffs were stark and unnatural, and this was compounded by the use of flat shading. After implementing the interpolation, it looked much smoother. To interpolate, we used the hex values to directly pull the RGB values, interpolated between that and our end color with the alpha term proportional to the distance between the min and max boundaries, and then updated the color of that face in the `faceColors` array.

Updating

A central feature of this project is simulating the movement of infinite terrain while "adding life" by making it more complex as it gets closer to the camera. This was all handled each time the update function was called for a chunk.

To move the infinite terrain, all the chunks were moved in unison by a small amount for each update. As we originally implemented the infinite terrain in a flat context, we were translating each chunk's `x` positions linearly in the negative `x` direction (towards the camera). This became an issue however when we wanted to change the position of the chunks so they were in a ring. Our first pass was to calculate the mesh's `z` position on the ring as a function of the mesh's `x` position. This worked well for the first few meshes, but further out, we noticed gaps in the meshes that got increasingly bigger until they were so large we could not see the meshes anymore. After some extensive debugging, we noticed that this was due to the fact that we were adding meshes linearly (caused by originally adding them in a flat context). Because the meshes are 30 units wide, they are spaced out one every 30 units. Closer to the camera, this causes only a minor difference (about `2` for the meshes at `x = 60` and `x = 90`) in the `z` positions of the meshes which is desired. Further from the camera however there is a major difference (about `90` for the meshes at `x = 930` and `x = 960`) in the Z positions in the meshes. As the meshes are only `30` units wide, there are `60` units of empty unfilled space between the meshes at a greater distance. This difference can be viewed in the table and graph below.



x	y_1
0	0
30	0.4501013
60	1.8016229
90	4.0582346
900	564.11011
930	632.44048
960	720

$x^2 + y^2 = r^2$

$r = 1000$

$y_1 = -\sqrt{r^2 - x^2} + r$

The empty space issue prompted a redesign of the way we were positioning chunks. To remedy this, instead of moving chunks linearly in `x` and calculating `z` from that, we decided to move them by an amount theta along the unit circle, then calculate the corresponding `x` and `z` values using basic trig. The result is that chunks are evenly spaced circularly along the ring rather than linearly.

Adding Life

The main way that our project emulates “adding life” by increasing complexity is through linear interpolation. Using different boundaries, we scale the terrain from flat to mountainous, the color from colorless to colorful and grow trees from sapling to fully developed. Each of these have their individual complexities, but the basic form of each is that there are three areas separated by two `x` boundaries. The area furthest from the camera is the 0 or default, or lifeless state, and the area closest to the camera is the 1 or alive state. The area in between has an alpha value ranging from 0 at the far boundary to 1 and the near boundary which is a function of the `x` position of a vertex/face/tree in a mesh. The alpha value is used to interpolate between the lifeless state and the alive state for this middle area. The result is a nice gradual shift from one state to the other.

Breathing

The last feature that is handled by the update function is breathing. Breathing gives the effect that the terrain is inflating and deflating. It works by scaling each vertex’s `z` position using a compounded sine function of the timestamp, user input scalar and the `x` and `y` position of the vertex. When the user input `breathingTerrain` is 0, breath is 1.0, so it does affect the terrain. The full equation is below. Through experimenting with different equations, the magic numbers in the equations below give the (subjectively) best results.

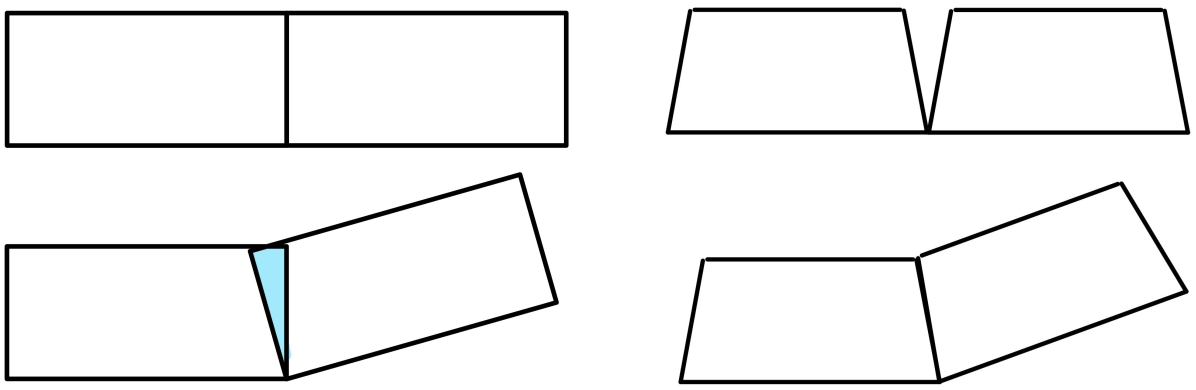
```
time = timeStamp / (10 * 60) * breathingTerrain;
breath = 1.0 - Math.sin(time) * Math.sin((xPos+time)/63) *
Math.sin((xPos)/29) * Math.sin((yPos+time)/45)* Math.sin(yPos/75);
```

Chunk Manager

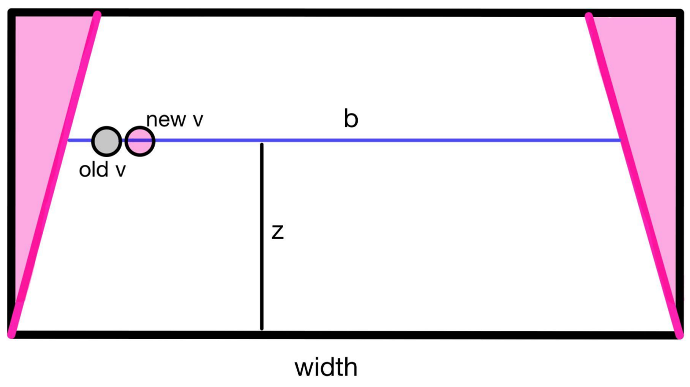
We create a class called ChunkManager to manage all the chunks within the scene. The ChunkManger had two essential functions, initializing all the constants needed for placement and movement on the ring, and creating/deleting chunks. The organization of the classes in this way was necessary and it became much more clear to manage the chunks in the ring. Each timestep, the ChunkManager would update each of its children chunks

Mesh Overlap Due to Rotation

By taking each chunk and rotating its mesh relative to its position on the ring, the meshes on the chunks inevitably overlapped (left figure). In order to rectify this overlap, we had to ensure that the `x` positions of the edge vertices on neighboring chunks were equal. To accomplish this, we could scale the mesh and its vertices in proportion to each vertex's height (`z` position), transforming a slice of the rectangular prism bounding box to that of a trapezoid. This would allow us to rotate the meshes without overlap (right figure).



In order to do this scaling, we could map the `x` position of each vertex on the chunk to one closer to the center, proportional to its height. The mapping for this would be, using the variables from the figure below: map `vertex.x` from `(-width/2, width/2)` → `(-b/2, b/2)`.



In order to find `b`, we could use the ratio of `width/radius` which would be equivalent to `b/(radius-z)`, so `b = width*(radius-z)`. We could then apply the mapping function as discussed above (linear interpolation) and we are left with the new mesh.

Meshes Rendering in Distance

Once we had solved the mesh overlap issue, another presented itself. Chunk meshes in the distance would appear with thin lines of empty space between them, likely due to its distance from the camera and how meshes that are touching would render (and possibly floating point arithmetic errors). In order to solve this problem, we introduced an EPS term that would overlap the meshes by a slight degree. While this solved the problem in the distance, up close, there existed awkward holes and overlaps to a degree that was insufficient for our final product. So we got rid of the EPS term and instead linearly interpolated each chunk's distance from the camera, and we overlapped chunk meshes directly proportional to their distance from the camera. So now, the meshes towards the back of the ring are much more overlapped than those close to the camera, removing the lines of empty space while preserving better continuity close to the camera.

Lighting

Originally, we wanted to place our project within a sphere with a texture of the sky and have a sun and moon as point light sources rotating throughout along with the sky texture, but we were not able to implement this feature in time. Instead, we chose to implement two types of lighting, basic lighting which illuminates all faces evenly, and a point light representing a nearby star. To have shadows on the terrain, we used a `MeshLambertMaterial` for the terrain and enabled the properties for casting and receiving shadows. You can toggle between these two lightings using the checkbox in the GUI.

Airship & Camera Movement

Though we moved the meshes underneath the camera in the `x` direction to give the illusion of forward movement of the camera, in order to achieve the airship and camera moving perpendicularly to the movement of the ring, we moved the positions of the airship and camera. At first, after adding an event listener to detect left and right arrow key presses, we had the camera move a small step left or right. This implementation created jittery camera movements, likely because the event listener does not update as frequently as the rendering update loop does. So the next step was to add states for whether or not the key was pressed, and if so, then change the position each step in the update loop. This was less jittery, but any directional change was sharp because there was no acceleration. So we introduced a `yAcc` term that changed `yVelocity` at each time step, and it allowed us to create smooth accelerations of the camera and airship. A couple small additions that improved the experience included rotating the airship to turn with the camera based on `yVelocity`, as well as pulling the camera back to give the illusion of a faster airship based on `state.speed`.

Conclusion

The approach we decided to take is promising. We were able to attain four of the major goals of the project with this approach, but it did limit some aspects that we wanted to explore. Compared to using a single large mesh, chunks are less intuitive and require more overhead to properly manage them. The benefits of only having to adjust the height upon creation of a chunk were nullified by the project's goal of interpolating between flat and full scale terrain. It could have made more sense to use a single large mesh and directly update the terrain using the noise function on every update, which would also allow for a much easier way to adjust the parameters for noise. However, having chunks does make the movement of the trees and other potential objects within chunks easier.

What We Learned

We learned a great deal while working on this project. On an abstract level, structuring and architecting the entire project and each component’s dependencies required foresight and gave us experience with tackling a large project. Also, having mostly used `Vector3` in the past assignments, it was really cool to see everything that was available in Three.js. The extensive library was incredibly useful for so much of this project, especially regarding mesh manipulation. Building upon the GUI was also a first, where we could add parameters that manipulate the output of the web app in real time. Computational limitations and memory management also certainly came into play, and we had to be sure to limit our loops as well as free up memory as soon as it was not needed. Though it is not visible intentionally, behind the camera, each chunk and the object within it are discarded from the scene as well as references to them, freeing their memory.

Results and Success

While we wish we had time to do so much more, it really built our self confidence that we were able to create as much as we did in such a short time frame.

The most difficult part of this project was curving the terrain into the ring and moving chunks along that circular path. Figuring out the behavior described in the chunk updating subsection where the chunks had gaps between them at long distances took a very long time. The chunks move large amounts in the `z` direction at far `x`'s as shown on the graph, so it was hard to catch the intermediary step where that was happening. It took inputting all of the equations into a graphing calculator and setting up the table shown in the figure in the updating section to realize what was happening at those large distances. Fortunately, we did not have to handle rotation of the chunks in addition to translation because of the very helpful `Object3D.lookat()` function where we could just pass the location of the center of the circle to the function.

We measured our success by how many of the desired features we were able to implement. We were able to successfully attain the goals of infinite terrain generation, moving the terrain along a circular structure, “adding life” by increasing complexity and modifying the terrain based on user input. We were unable to create a chrome extension so that we could use the project as the background for a Google search page. We spent some time looking into it, but it looked like it was going to be much more complicated than we expected so, given the time constraints, we decided to focus on the functionality of the project, with the next step of creating a chrome extension. We also wanted to add dynamic life such as animals with flocking behavior which would be a cool addition for the future.

Improvements

There are a few issues that need to be revisited. As described in the Chunk Manager subsection of the Methodology section, we struggled to stitch together the chunks after curving them inward. Shifting the vertices inward to create trapezoidal cross sections eliminated many overlaps and holes in meshes, but there are still a few spots where the boundaries between chunks don’t line up perfectly. Additionally, we began to experience minor framerate issues as the project grew. Although low framerates might be unavoidable given that the nature of increasing the complexity necessitates updating every vertex for every mesh for every timestamp, the project would certainly appear more polished if the frame rate was higher.

Contributions

Pierce and Mitchell worked together to implement terrain generation within a chunk. Mitchell implemented the GUI and its effects as well as creating and moving chunks along the ring. Pierce developed the camera movement with the airship and “adding life” (scaling height, color, and adding plant meshes) as the chunks got closer to the camera. The pair effectively and equivalently divided the work in this project, and though it may seem like they built a web app, they really built a friendship.

Acknowledgements

We would like to acknowledge our TA, Vedant Dhopte, for discussing fundamental ideas with us and providing useful suggestions and advice.

Works Cited

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

Dreamworld COS426 Project by Lauren Johnston, Joanna Kuo, Elizabeth Petrov, & Lydia You:
https://github.com/dreamworld-426/dreamworld/blob/25a8dc8dcec98c3ff7935902d090b3fd23af9c25/src/components/objects/Bird/Bird.js

Low Poly Tree Mesh:
https://sketchfab.com/3d-models/low-poly-tree-6d986e0b24b54d85a5354e5cac6207a1

Low Poly Airship Mesh:
https://sketchfab.com/3d-models/low-poly-airship-fa317292c6f142b68d64620251f99b40

Desmos Graphing Calculator: [Online]. Available:
https://www.desmos.com/

Documentation and Projects on ThreeJS: [Online]. Available:
https://threejs.org/