Alien Worlds Generator

Nathan Coggins (Junior - Computer Science)
KateLynn Pullen (Junior - Computer Science)

i. Overview

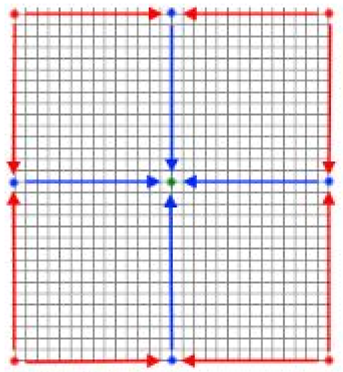
In keeping with the theme of ‘Alternate Worlds’, we decided to create an interactive program that creates a different alien landscape on each run. These landscapes differ based on fractal landscapes and water and variants of color, size, and placement for stars and planets. The user has access to four sliders to alter the parameters which are used to generate the image. Some of the interesting technical features used in this project include animation, generation of fractal terrain, transparency, and layering of scenes.

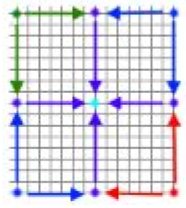
ii. Randomly Generated Elements

Our program randomly generates many of its values. This includes the size, position, texture, light color, and rotational speed of moons, and the geometry of clouds and terrain.

a. Fractal Terrain

The terrain is generated using a variation of the Diamond-Square algorithm, a recursive method of generating terrain.



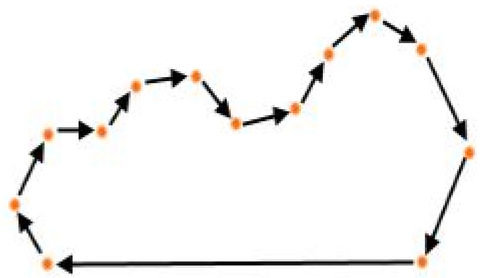


Each recursion, four points of an array containing the height values of four corners of a piece of landscape are passed into the algorithm. These four points are used to create four outside edge points (represented in blue on the top graph to the right) and one inside middle point (represented in green). The process is then repeated for each new square created, as illustrated in the bottom graph for the lower right quadrant of the top graph.

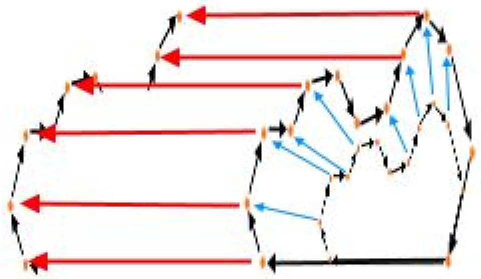
The height values of the new points are the average of the height values of the points used to create them plus some random skew. To smooth the terrain, the random skew applied to each point is divided by how many recursions have happened so far to the power of 1.5. For example, the first recursion’s skew would be divided by $1^{1.5} = 1$, the next by $2^{1.5} = 2.83$, and so on.

The algorithm completes when it has filled up the entire landscape array, which in our implementation is represented by checking to see if any generated midpoint is a non-integer. Because of this, maps need to be size 2^n so they can be divided into their smallest subsquares perfectly.

b. Clouds



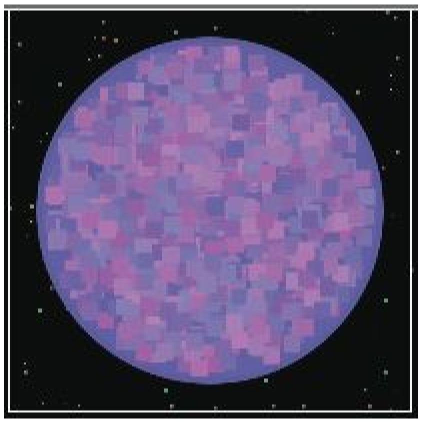
The clouds are created using three.js' ExtrudeGeometry and Shape methods. Shape is used to generate a cloud-like 2D object via points and lines. For clouds, we randomized three values - left height, right height, and width - then created lines to points using those values. The nine points at the top of the cloud are created using combinations of left height, right height, and a random skew.



After the shape is created, it is given to three.js' ExtrudeGeometry method. This creates a 3D mesh out of the 2D shape and also applies bevel to the flat faces at the front and back, ensuring the cloud looks rounded and puffy from all angles.

c. Textures

Textures are generated at runtime using HTML's 2D Canvas context. For the near moons, we first generate a circle with a random color value. Within that circle we layer 1500 squares of random nearby hues (within 60 points of the initial color value on a [0, 360] scale), then export the canvas as a texture using three.js' canvas-to-texture support.



The star map is generated similarly. The stars are represented by 4000 squares of random pixel size (0, 2]. The star map is set to repeat four times for width, and twice for height. This minimizes texture stretching, although the poles of the sphere this texture is mapped to still exhibit some distortion.

iii. Lighting Effects

Since the terrain's material does not change, we needed a way to make the terrain appear distinct between generations. To achieve this, we added low-intensity directional lights to the moons. Since these vary in position and color, the terrain varies in color and shading, which creates the illusion of different landscape materials.

iv. Transparency

The clouds, water, and atmosphere are all semi-transparent. This caused problems with the rendering of moon sprites, as sprites are drawn last by default in three.js and any semi-transparent object or plane in front of the them ends up completely culling the sprite. This also happens with semi-transparent objects in front of other semi-transparent objects, but that can be solved by turning off depth writing. We attempted to solve the sprite problem by manually setting the order of objects to be rendered in the scene, but this solution still caused transparency issues.

To solve this, we create two different scenes - a foreground scene and a background scene. In the background, we add the star map and the moons. In the foreground, we add everything else. We then render these scenes on top of one another.

v. Celestial Simulation - Orbit, Placement, Lighting, and Animation

While the end product only implements the animated rotation of the star map, animation for the planets/moons and their light sources is mathematically supported in the code and the same values used for animation are used for celestial placement according to the time of day.

All celestial animation follows the same principle - at each re-render of the scene, a number is added to the object's x, y, and z rotational values. For all satellites, the specific numbers are randomly generated at initial generation to simulate standing on the surface of a planet with a rotation different from our own.

The star map rotates every re-render by an x, y, and z value within $[-\pi/4000, \pi/4000]$. The movements of moons are created based off of these values plus/minus some randomization, to simulate the fact that, while they have their own independent movement, the perception of that movement is affected by the rotation of the planet (which is, in turn, simulated by the rotation of the star map).

To rotate light sources and meshes around a point, we create a parent object at the point of rotation, then add another object to this parent at the desired radius. To the pivot, we add the mesh or directional light, and then we rotate the object by whatever values have been generated for it. During initial generation and placement, all such objects are rotated by a random x, y, and z between $[0, 2\pi]$.

The central star, or 'sun', of the system is simulated differently because we wanted the time slider to be intuitive for users. The central star does not rotate with the star map as would be expected, but instead rotates at the same rate in the x, y, and z directions. This creates a simple 50% day/50% night schedule that's the same between all planets. This central star does not have a mesh, instead it is represented as a strong directional light.

Both the atmosphere and central star change transparency and intensity following the equation $\frac{1}{2}(\cos(x - \pi)^{1/3} + 1)$ where x is time and $[0, 2\pi]$ represents a full day. This ensures the directional light of the central star is not shining through the bottom of the land mesh (as the intensity would be zero or very near zero), and that the atmosphere is appropriately clear of refracted light (which is represented simply as a change in transparency) at night. [1]

[1] cos graph created using desmos.com/calculator