CSE 333/533: Computer Graphics
Course Project: Realtime Procedural Terrain Generation

Anshul Mendiratta - 2018219
Abhishek Pratap Singh - 2018211

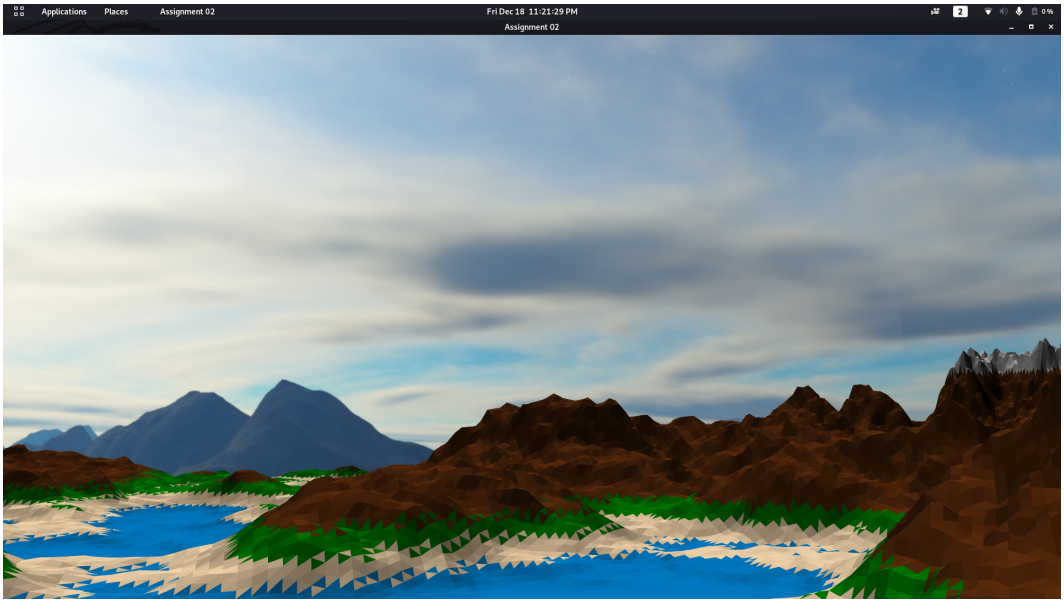


Figure 1: Current Output In The Project

1 ABSTRACT

Games, movies and VR/AR applications make use of rich and dense environments to make scenes look more realistic and give them more life. It can be very time consuming and labour intensive to make these environments manually. Procedural Generation provides a good way of producing realistic-looking environments without developers having to invest a lot of time and energy. We want to explore and come up with a method that can produce an aesthetically pleasing looking environment in real time, while giving the developer a good amount of control over how the environment looks.

2 Introduction

Procedural Generation is the method of creating data using algorithms and formulae rather than creating it manually. The data created is random in nature and different aspects of it can be controlled depending on the method used for procedural generation. While such formulas find many applications in different fields, we will mainly focus on using them for terrain generation in real-time.

The method of creating procedurally generated environments is analogous to making a painting and hence we will often use painting as a reference to explain concepts of procedural terrain generation. We will first go over the major steps for the creation of an environment using procedural generation -

1. Creating a noise/height map - This is a 2D matrix created using a noise function that will decide the height of each point on the terrain. This will be responsible for creating the peaks and valleys in the terrain.

2. Creating a mesh - Using the height map created in the first step, we will create a 3D representation of our terrain. This is analogous to making the outlines of the mountains, sun and river in a painting.

3. Adding Movement - We want to add the ability to move around in the world, so that we can explore it easily.

4. Adding details to the mesh - Once we have the surface of our mesh, we want to add details to it to emulate natural phenomena, such as erosion due to rain and heat, on our mesh. This is analogous to making the boundary of the objects in our painting more detailed, eg. adding ridges and crevices to the mountains.

5. Adding colour or texture to the mesh - Up till now our surface has had one plain colour. We want to make this look more realistic. We can make use of textures for this purpose. Texture can be thought of as an image that is wrapped around an object to make it look more realistic. This is analogous to colouring inside the objects that we had created before in our painting.

6. Adding Vegetation - To give our terrain more life we'll add trees to it. The trees are created through instancing and use textures.

7. Stitching multiple terrains together - We want to be able to stitch our terrains together to create an endless terrain. We can set the number of chunks visible at a point of time and only render those to improve performance.

3 Implementation

3.1 Milestone 1

3.1.1 Noise Map Generation

We can visualize our terrain as an initial flat 2D plane that we want to extrude at certain points to create peaks and valleys in our terrains. Extruding our terrain by a random amount at every point doesn't simulate the variations in height in real life. Hence we make use of a pseudo-random noise function that can assign an appropriate height to every point in the plane to simulate real-life environments. The algorithm we use is Perlin Noise Generation.

Developed by Ken Perlin in the year 1983 to create more realistic looking graphics for the movie, TRON. Perlin noise and its variations have since become the most widely used algorithms for procedural terrain generation. Perlin noise results in smoother peaks and valleys in comparison to using random values as it takes into consideration the influence from ths surrounding points. Further, we can get an infinite plane of height maps if we map coordinate (x,y) on the mesh to perlin(X,Y), hence helping us in creating our endless terrain.

We combine several Perlin noise maps together to get more realistic results. Our noise map generation function takes in the number of octaves(Number of overlapping perlin maps), persistence(Decrease in amplitude of octaves), lacunarity(Increase in frequency of octaves) and noise scale(Scale of the obtained map) as parameters.

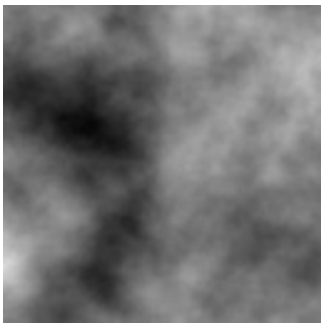


Figure 2: Noise Map Generated From Perlin Noise

3.1.2 Mesh Generation

Now that we have a heightmap, we want to convert it into a 3D mesh. To do so we will first create a triangulated 2D plane with length l and breadth b. Now using the noise map obtained in the previous step, we will assign heights to each vertex. The heights can be extracted by checking the colour in the generated noise map. So the more darker a point is the higher it will be in the mesh and vice versa. As the mesh is triangulated, when we move one vertex upwards or downwards, the nearby points will adjust to maintain continuity and hence we have our required 3d mesh.

Figure 3: Mesh generated using Perlin Noise, where vertices are colored according to their height

3.1.3 Adding Movement

Once we instantiate a world in OpenGL, we have a view matrix. To move around in the world, we can use the view matrix. OpenGL by itself doesn't have any module for a camera, but using the view matrix we can simulate a camera by moving all objects in the scene in the reverse direction. This creates an illusion that we are moving through the scene. The steps in creating a camera is as follows:

1. We begin by first defining an initial camera position and camera direction. We can use this to get the right and up axes.
2. We now specify the **glm::lookAt** function which tells OpenGL where to look - glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
3. Now the camera will look in the front-forward direction. We now process the user keyboard inputs using **glfwGetKey()**.
4. Finally we specify the camera movement speed as a function of scene framerate. We now have a camera that moves using keys.
5. However, games use the mouse for controlling the camera rotations. To do this, we first process the mouse input using

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

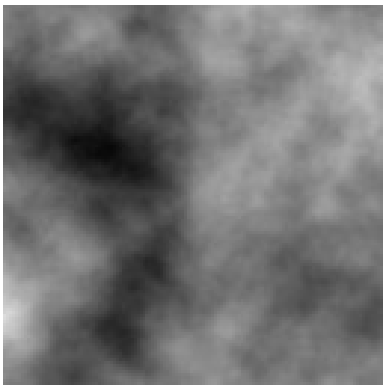
6. We then map the mouse input to a rotation function. The rotation function consists of pitch, yaw and roll which is also known as the Euler Angles.
7. To consolidate the above computations, we create a camera class. Now, camera is created as an object in the game.

3.2 Milestone 2

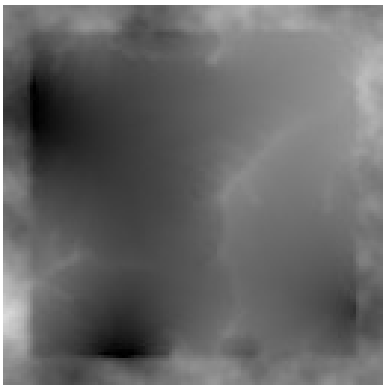
3.2.1 Adding Erosion To Mesh

Now that we have our base mesh we want to add some more details to it to make it look more realistic. To add more details to our mesh we can thermal erosion, hydraulic erosion, or fast erosion. We make use of hydraulic erosion in our project.

Hydraulic erosion simulates changes to the terrain caused by falling raindrops dissolving material, transporting it and depositing it elsewhere on mountains. We first will generate N random raindrops over our terrain. When a water droplet hits a surface it will carry some amount of sediment along with it as it flows down the terrain, depositing some of the sediment and gaining more as well. It will continue to do so until the raindrop has either evaporated or it has reached a flat surface where it will deposit all of its remaining sediment.



(a) Without Erosion



(b) With Hydraulic Erosion with 70,000 drops

3.2.2 Adding colour or texture to the mesh

We still haven't added any colour to our terrains yet. Hence, to make them look more real we add either colour or texture to our environment depending upon the style of the environment that we are going for.

One way of giving more life to our environment is by simply adding solid colours to it. For example, assigning blue colour to the water, grey to the mountains and so on. Colour is added by assigning a colour to each vertex. As of now, we have assigned different regions in our mesh according to their heights. Each region has a specified color. While this method doesn't give realistic-looking results, one may consider it if their environment demands it. One example of the resulting output, where colour is added according to the height of each vertex is provided below. (Note that we have used flat shading inside the fragment shader currently as it gives more appealing results.)

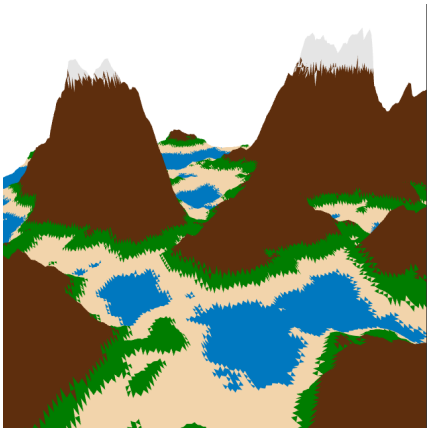


Figure 4: Mesh colored according to different regions, where each region is assigned a different color

3.2.3 Grass and Trees using instancing

To add flora on the terrain, we load trees, grass, bushes and shrubs models from objects (**.obj**) files alongwith their materials (**.mtl**). Since OpenGL doesn't have any **.obj** loading feature, we use the Assimp library to load the files. The Assimp Library is responsible for reading the **.obj** and **.mtl** files and mapping the object information to usable data structures. The steps for this task are:

1. Load model **.obj** and model material **.mtl** information using the Assimp Library.
2. Create mesh of the object using the above information. We have the information about the vertices and faces position which is then used to create triangle arrays.
3. Shade the mesh using the material information. The mtl file provides information of the texture for each vertex and face. We create a new shader class that does the texture mapping using the mtl information.
4. We now are able to load a model into the scene. The output is as follows:

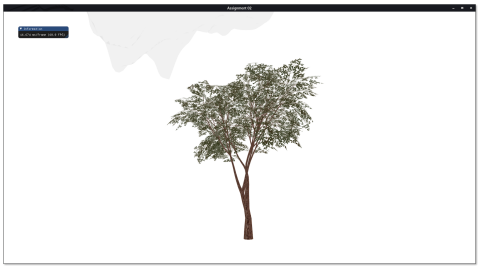


Figure 5: Single model loaded with materials.

5. However, we want to place multiple models in the scene. To do this efficiently, we use instancing. This is because the OpenGL render call **glDrawArrays()** is computationally expensive. Hence if we wanted to place thousands of models, the render call would become a bottleneck. Hence, we use the alternative instanced render call **glDrawArraysInstanced()** provided by OpenGL. This call tells the GPU the number of model instances before hand. Hence, the GPU does all model loading in a single render call, thereby increasing speed and performance.
6. Once several instances of the loaded is generated, we begin placing the models in the scene. To do this, we first create a transformation matrix for each instance using the Perlin terrain height map. We then apply random scale and random rotation to give a more natural feel. We perform instancing of different types of models so as to have a diverse vegetation in the world. The output of this is as follows:

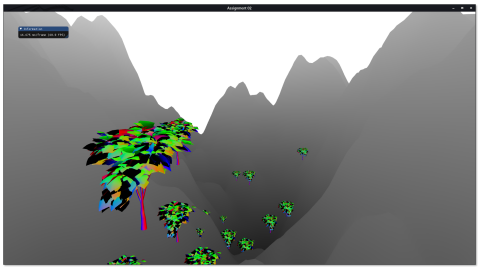


Figure 6: Instanced tree models placed across terrain.

3.2.4 Making our terrain endless

Currently our terrain is just a single plane. We want to be able to make our terrain endless. We first define the number of chunks we want to be able to see around us from the plane we are standing on.

Say we take a value of 1, then at any point we have 9 chunks of terrain in the world space, one is the one we are currently standing on and the other 8 are the chunks surrounding it. In this example we would start off with 9 frames. At each frame we check how many chunks are more than 1 unit away from our current position, we remove these chunks from our terrains and replace them by creating new chunks at a distance of 1 unit from our current position. Hence, there are again 9 chunks in world space.

The continuity is maintained using the Perlin Noise function. While creating a new chunk, we simply pick up from where we left off in the previous chunk, hence ensuring that there are no irregularities or visible seems at the edges of 2 planes.

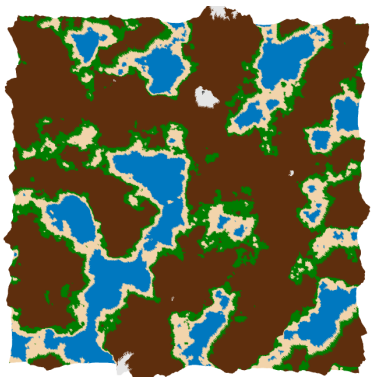


Figure 7: Top view of 9 meshes stitched together

3.3 Milestone 3

3.3.1 Adding Lighting

Up until now, our terrain looked flat and one couldn't tell if there were crevices on the surface or not. Hence, the details we had added weren't as clearly as visible. To make these visible, we added lighting to our scene.

We made use of the Phong Lighting Model for this purpose. Our terrain is made up of several triangles, hence we first calculated and stored, the normal for each of these triangles. Using the normals obtained we calculate the corresponding specular, ambient and diffuse lighting for each triangle. As we aren't interpolating colors for each triangle and are instead assigning a single color to the entire primitive, it didn't matter whether we used Gouraud or Phong shading. We used the normals inside the vertex shader to find diffuse and specular light and hence, ended up using Gouraud shading.

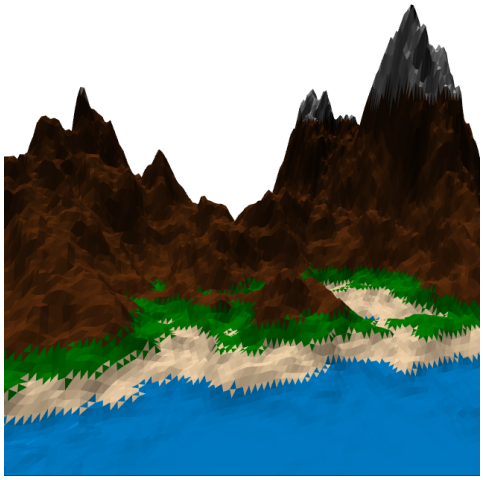


Figure 8: Terrain after Phong Lighting

3.3.2 Adding Fog

We now wish to add fog to our scene to give our environment more life and give a greater sense of distance and depth. We want the objects placed far to be less visible than the ones close by.

There are a few common ways of implementing fog in OpenGL. OpenGL has an inbuilt method to add fog as well, however, we implemented our own method. We chose to use an exponential function which decides the amount of fog using the distance of the viewer from the fragment to be shaded and the fog density which will be some float value deciding how thick the fog is. We first pass the camera position and the fragment position to the vertex shader. There we find the distance between them and using our formula we find the amount of fog color we want to blend with our original color. This way we can color the further away objects to look like they are covered in fog, whereas the close by objects will be clear.

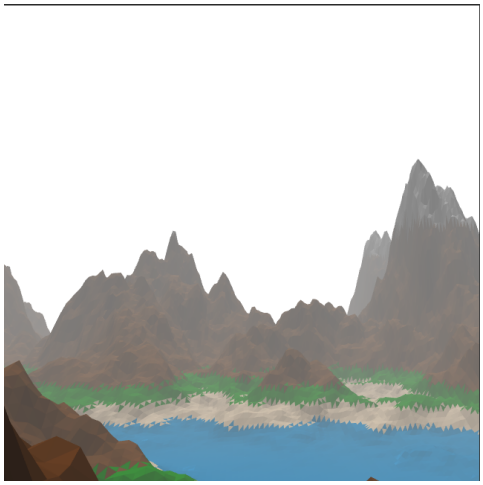


Figure 9: Our scene after adding fog

3.3.3 Adding Skybox

The world environment lacked realism since the background was white up until now. To resolve this, we added skybox to the world. We used a cube box with textures of sky on its faces. The HD textures were obtained from the internet. The textures were chosen such that they would seamlessly join at the edges. The cube was then mapped with the camera's movements. Hence, we created a skybox that fills the complete world background.

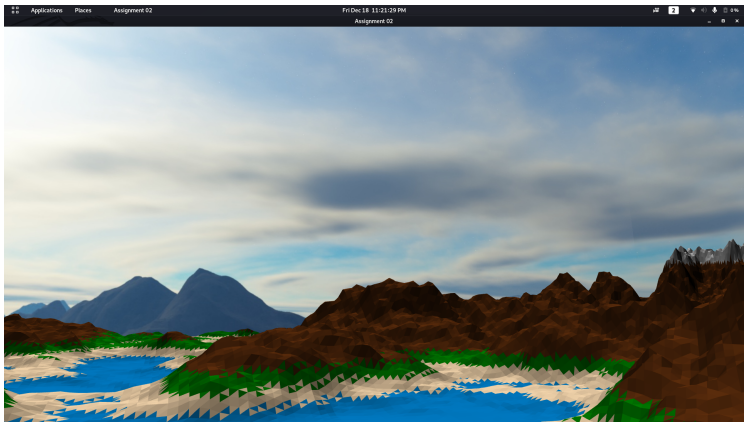


Figure 10: World with skybox in the background

4 References

1. https://assimp-docs.readthedocs.io/en/latest/, Assimp model loading library
2. https://learnopengl.com/index.php?p=Getting-started/Shaders, MTL material shaders
3. https://www.firespark.de/resources/downloads/implementation%20of%20a%20methode%20for%20hydraulic%20erosion.pdf, Implementation of a method for hydraulic erosion - Erosion reference
4. https://web.mit.edu/cesium/Public/terrain.pdf - Erosion Reference
5. //https://mrl.cs.nyu.edu/ perlin/noise/ - Ken Perlin's Implementation of Perlin Noise (Used in our project)
6. https://learnopengl.com/Advanced-OpenGL/Instancing
7. http://www.diva-portal.org/smash/get/diva2:1355216/FULLTEXT01.pdf
8. https://sci-hub.do/https://ieeexplore.ieee.org/abstract/document/8976682