Programmatically Generated Landscapes

Darshan Kapashi, darshank@stanford.edu
Sagar Chordia, sagarc14@stanford.edu

Abstract

One of the largest areas of research in computer graphics deals with natural phenomena. In almost every computer game, animated film and physics application one might see it simulated in one or more forms. **We want to create a world where we have generated with terrain, clouds, grass and water.**

Here is an example of a long shot. [7]

Index

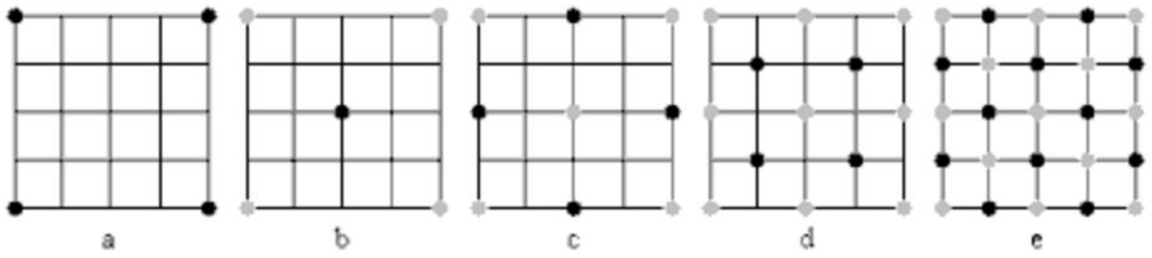
Terrain Generation

Height Map

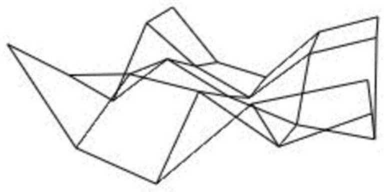
We evaluated various algorithms used in computer graphics practice to generate terrain. Some algorithms which attracted us included fault iterative algorithm, circle iterative algorithm, diamond-square iterative algorithm. We chose **diamond and square** algorithm for its simplicity and good results. Given length and width of terrain this algorithm outputs height map for each point. It provides parameters to control roughness of terrain, density of terrain etc. It is also known as the random midpoint displacement fractal, the cloud fractal or the plasma fractal, because of the plasma effect produced when applied.

You start with a large empty 2D array of points. To make it easy, it should be square, and the dimension should be a power of two, plus one (e.g. 33x33, 65x65, 129x129, etc.). Set the four corner points to the same height value.
This is the starting-point for the iterative subdivision routine, which is in two steps:

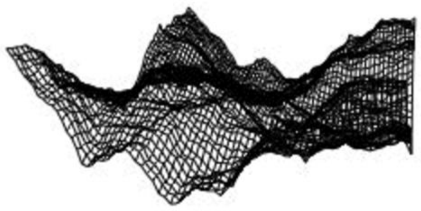
- *The diamond step*: Taking a square of four points, generate a random value at the square midpoint, where the two diagonals meet. The midpoint value is calculated by averaging the four corner values, plus a random amount. This gives you diamonds when you have multiple squares arranged in a grid.
- *The square step*: Taking each diamond of four points, generate a random value at the center of the diamond. Calculate the midpoint value by averaging the corner values, plus a random amount generated in the same range as used for the diamond step. This gives you squares again.



Diamond-square algorithm. (b) and (d) refer to square step. (c) and (e) refer to diamond step.



Terrain after 3 iterations



Terrain after 20 iterations.

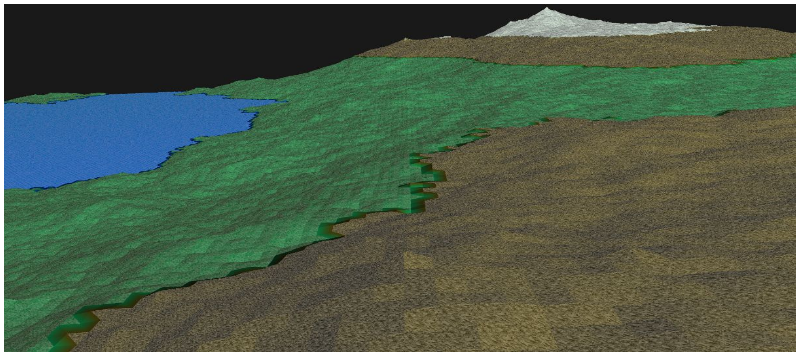
Note intuitively above algorithm ensures set of nearby points have almost similar height, but height across far-off regions could vary randomly. And thus we can generate mountains or valleys using above algorithm.

Assigning regions

Once we have height for each point in terrain, we can decide whether point belongs to ocean or water based on height thresholds. Color of point is decided accordingly. Let H be maxHeight in our landscape.

```
if (height > 0.7 * H) {
  color = iceColor;
} else if (height > 0.5 * H) {
  color = mountainColor;
} else if (height > 0.1 * H) {
  color = grassColor;
} else if (height > 0.0 * H) {
  color = beachColor;
} else if (height < 0.0 * H) {
  color = waterColor;
}
```

We noted that mixing openGl colors with textures makes surface look more realistic. So we loaded different textures as jpg images corresponding to different colors. Following image shows our basic generated terrain colored with different textures mixed with base openGl colors.

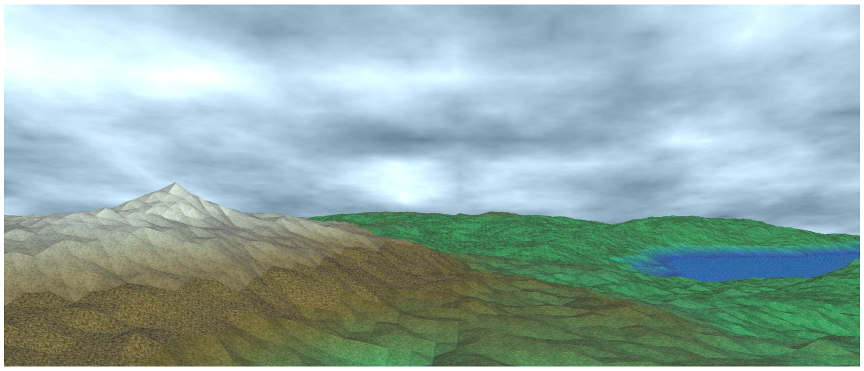


<u>Interpolation regions</u>

In above image because of sudden change of openGl and texture colors we see discontinuity in image for example when we go from mountain to ice. To solve this problem we define interpolation regions, where colors from surrounding regions are mixed based on distance from each region. So equation from above will be modified to:

```
if (height > 0.7 * H) {
  color = iceColor;
} else if (height > 0.6 * H) {
  minHeight = 0.5*H;
  maxHeight = 0.7*H;
  color = interpolateColor(
    height, mountainColor, iceColor,
    minHeight, maxHeight
  );
} else if (height > 0.5 * H) {
  color = mountainColor;
}
...
double interpolateColor(
  height, minColor,
  maxColor, minHeight, maxHeight
) {
  return minColor +
    (maxColor - minColor) * (height - minHeight) /
      (maxHeight - minHeight)
}
```

Our landscape looks something like this at this point. Now we don’t see discontinuity across regions like going from mountain to ice.



Vertex Normalization

In current image we see terrain is triangulated. This is because we have normals defined per face instead of per vertex. Basically for vertex shared across k multiple triangles we render that vertex k times, each time having normal as corresponding face normal. As a result fragment shader while interpolating points on a particular triangular face will use same normal for all points on that surface.

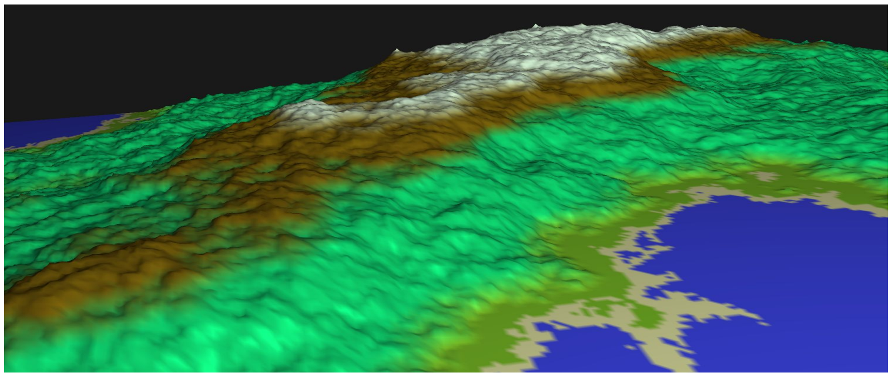
To solve this we define only one normal for shared vertex. This shared normal is average of normals of faces surrounding that vertex. Thus 3 normals of any triangular face won't be parallel. As a result fragment shader while rendering a point inside triangle will take weighted average of vertex normals. So entire triangular face will have normals changing smoothly from one vertex to other vertex. As a result we get smooth texture.

Implementing this was bit tricky because we needed to find out all neighboring faces for given vertex. We solved this cleverly by defining map keyed by vertex. If vertex already exists in map, then average normal else define new entry in map.

```
unordered_map <vertex, normal> weightedNormals;
for every triangular face t:
  for every (vertex v, normal n) in t:
    if (weightedNormals.keys() contains v)) {
      weightedNormals[v] += n;
    } else {
      weightedNormals[v] = n;
    }
    for_each(weightedNormals as (v,n)) {
      weightedNormals[v] = normalize(n);
    }
```

Laplacian smoothing

Even after vertex normalization there were some points in grid which had very different height compared to its neighboring points. This resulted in sudden peak on depression in terrain. To solve this we processed heights of our terrain map using laplacian smoothing which essentially averages vertex height based on surrounding vertices height. Following image shows our landscape after vertex normalization and laplacian smoothing.



Lighting

We implemented phong-shading model inside our *vertexShader* to process *objectColor* of each point. Our landscape has one diffuse and specular light source. We used one source to resemble sun. We also have ambient light source to give effect of light diffusion through atmosphere. To make scene more realistic we had to increase specular and diffuse strength of waves which made water more reflective. We eventually assigned different specular and diffuse strength to ice, sand, grass and water.

Water waves

Displacement computation

In order to animate waves in water we used current_timestamp and location of point in grid to compute change in height. We model it by sinosidual equation which has parameters to control epicenter of wave, speed of wave and magnitude of wave.

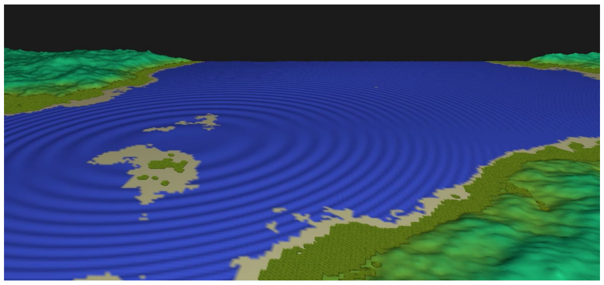
$$\Delta Z = d * sin(a(x - x_0)^2 + b(y - y_0)^2 + c * current\_time)$$

In this equation (x0, y0) denotes epicenter of wave which provides us option to center the wave where desired.

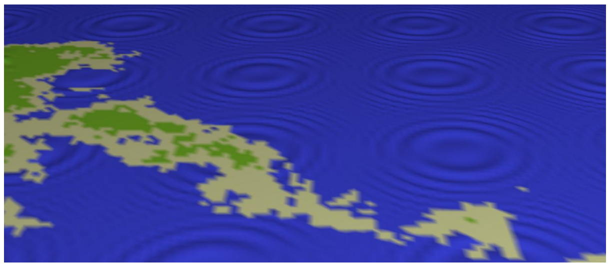
- a, b control the wavelength of wave
- c controls speed of wave
- d controls height of wave

Controlling these wave parameters can produce amazing images as shown here:

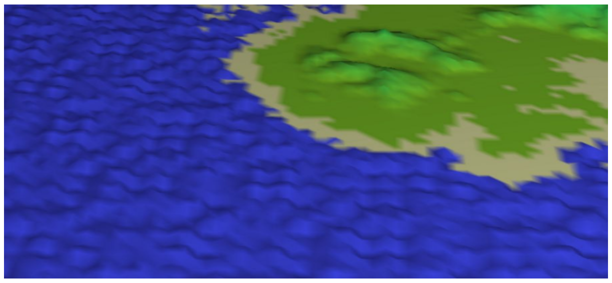
a and b are set to very low values
a = 0.01
b = 0.01
c = 1
d = 0.5



a and b are set to reasonable values
a = 0.5
b = 0.5
c = 1
d = 0.5



c and d values are increased
a = 1
b = 1
c = 5
d = 1



Normals computation for waves

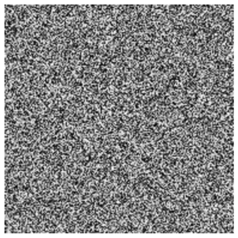
To give reflectance effect to water we need to recompute normals based on changed z values. This was bit challenging since vertex shader doesn't have access to neighboring vertices. But we can determine deltaZ of each neighboring point based on (x-1,y), (x+1,y) .. and so on. Since z of each point is zero, its possible to compute normal of each surrounding triangular face based on deltaZ. And finally because we have recomputed normals for every face, we can recompute normals for each vertex. As a result normals of each point change with time and that gives reflectance effect to water.

Note modeling wave using this equation was our innovative contribution. We searched net but couldn't find satisfying material. This equation is quite simple and gives lot of power to model wave.

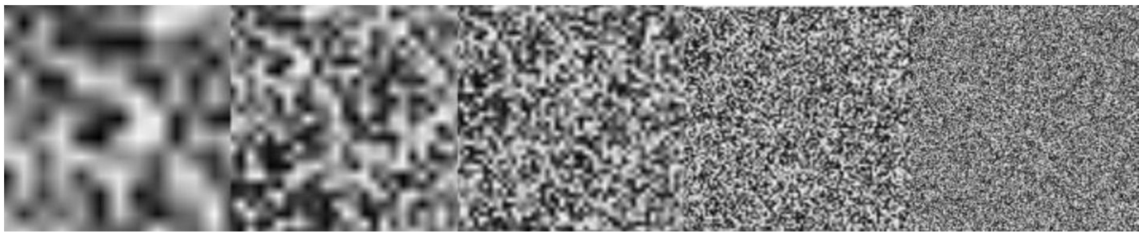
Clouds

Perlin noise

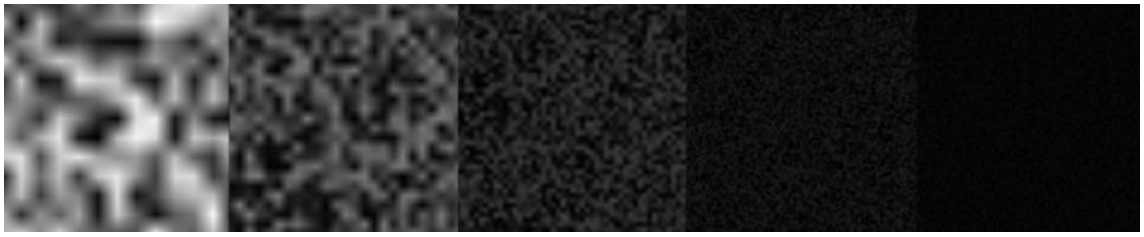
One of the major challenges is to make the clouds look natural. Clouds have a random shape, however a uniform noise doesn't quite look like a cloud.



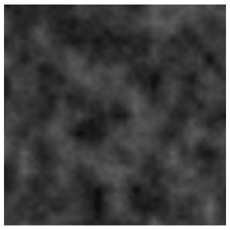
We use a different kind of noise called Perlin Noise [1][2]. The idea is to generate random large features, then add random noise to small parts of these large features, then add more noise to these 2nd level features and so on. Specifically, we take the uniform noise and zoom it 16x, 8x, 4x, 2x and 1x as shown in the images below.



Then, notice that the smallest turbulence should be weighed less to give an effect of local noise. Here are the same images, but they are divided by 1x, 2x, 4x, 8x and 16x.

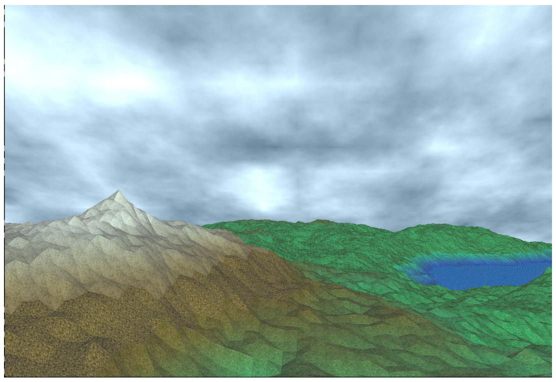


Finally, we take the average of these noise images and get a natural looking cloud-like image.



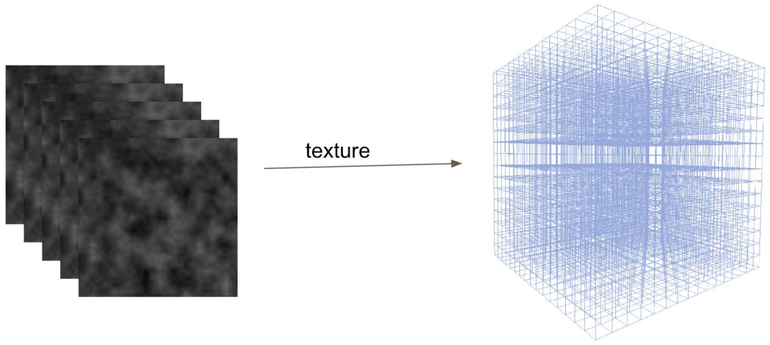
An additional challenge here is to get a smoother image when zooming in. For this, we use a smoothing function, which linearly interpolates between pixel values, this makes the image look non-blocky.

There are 2 ways to go from here. In approach #1, we take such a 2D noise image and use it as a texture. We render a giant quad and place it appropriately in the scene. We did this as part of the milestone.



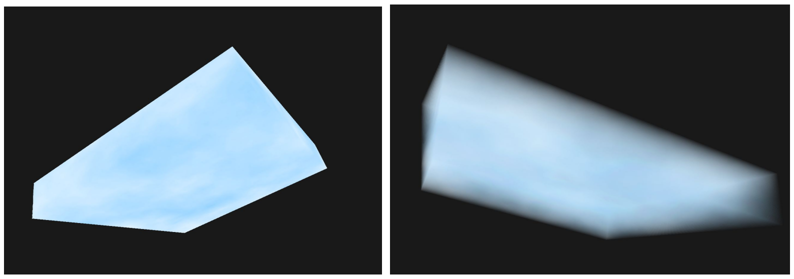
3D Perlin Noise

In approach #2, we make the clouds 3D. We generate 3D perlin noise, same as 2D perlin noise but with an extra dimension, you can think of it as N layers of 2D perlin noise. For each layer, we use the noise as a texture on this layer.



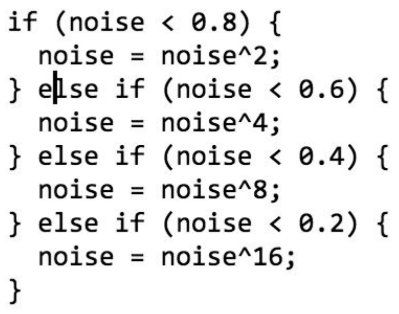
Making it cloud-like

This gives us a decent 3D cloud (left image below), but we can do better. So far, we used *(perlin, perlin, perlin, 1.0)* as the color vector. We get a better effect if we tweak the alpha channel. Using *alpha = 0.1 * perlin*, we can make the cloud look softer (right image below).

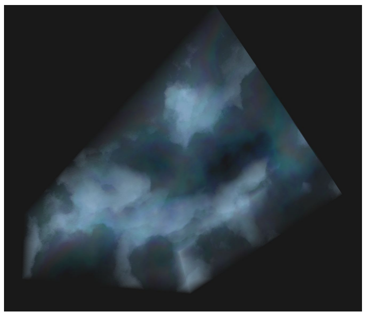


However, it is still a cube, it needs to have some shape. We tried 2 approaches, #2 worked better. The idea is to make the cloud thinner in some areas which will give rise to shapes.

1. Scale the alpha values proportionally to the distance from the center of the cloud and using a random number.
2. Exponentially drop off the color if it is lower than some thresholds. This way uniform noise is no longer uniform but drops off faster. The specific function we used is given below. "noise" represents the value of the grayscale color computed from the perlin noise texture.

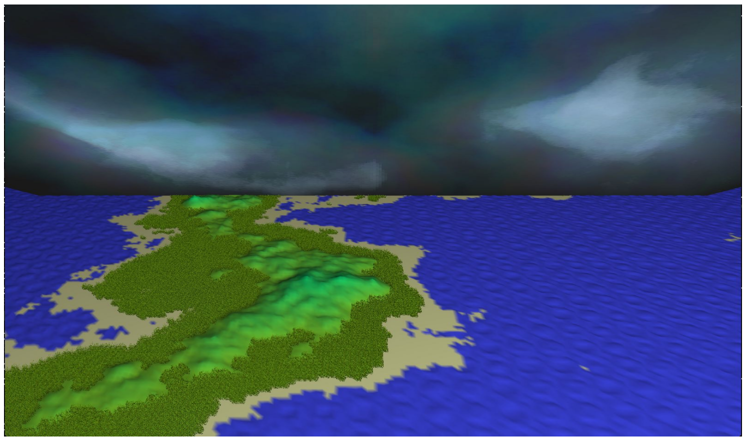


```
if (noise < 0.8) {
  noise = noise^2;
} else if (noise < 0.6) {
  noise = noise^4;
} else if (noise < 0.4) {
  noise = noise^8;
} else if (noise < 0.2) {
  noise = noise^16;
}
```

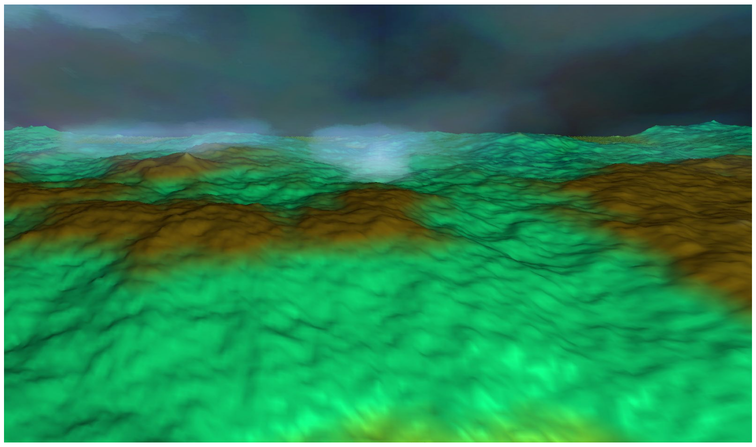


Transparency

Finally, to get the desired effect, we need to render the clouds last in the scene. OpenGL doesn't support transparency in its natural sense. When you use GL_BLEND, it uses the current value of the color in the framebuffer and blends the new color. This means, to render transparent objects correctly, you need to sort all your primitives in the order from farthest to nearest and render them in that order. If you don't, we get the image on the right.



If you do it right, i.e. render the cloud last, after the terrain, then we get this.

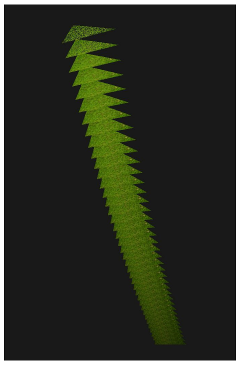


Grass

Using layers of triangles

It is a challenge to render grass accurately because it requires lot of primitives. If we want to walk through the grass, we need one approach, if we want to look at it from farther away, we need a different approach, for efficiency. We decided to tackle the case of the macro-view.

[3] We model grass using triangles. For each triangle in the scene, we add N triangles on top of this triangle. We use an image which looks like grass and apply it as a texture to each of these triangles.



Making strands

In a terrain which spans a large area, grass like this appears very coarse. To make it grassier, we create strands. We already have N more triangles for each triangle in the scene, where typically N is 20 or 40 for a good look. We can't possibly subdivide these triangles. We use a different approach using textures and tweaking the color and alpha values. We create another texture, in which we care about 2 channels, the R and the A. Let's talk about the alpha channel first. We set a parameter, which controls the density of the grass. Based on the number of layers and the density, we compute the number of points we should sample.

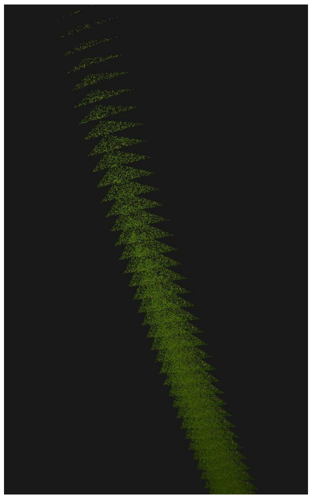
```
int numStrands = (int)(density * weight * height);
```

We initialize the texture to 0, i.e. transparent. For M samples, in each iteration we pick a point (x, y) and set it as opaque. Finally, we have 2 textures, one which gives it the grass color and one which makes some points transparent. Notice that the same point in each layer becomes transparent, so we get a strand like look.

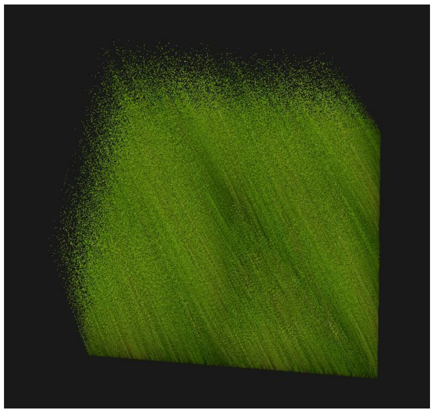
To close the loop on making it look more realistic, we model 2 more effects:

- The lower layers appear darker than the higher layers. To achieve this, we simply set the color of the points at a lower layer to be darker in proportion to its layer number.
- The strand appears thinner at the top and thicker at the bottom. To achieve this, we tweak the alpha value and make it smaller at the top and higher at the bottom, again in proportion to the layer number.

For each strand, we compute a number which is a fraction of the strand height. We use this in the fragment shader to change the transparency and shadow, which gives the effect of varying strand heights and a more realistic looking grass.



Finally, putting these concepts together, a tiny patch of grass looks like this image.



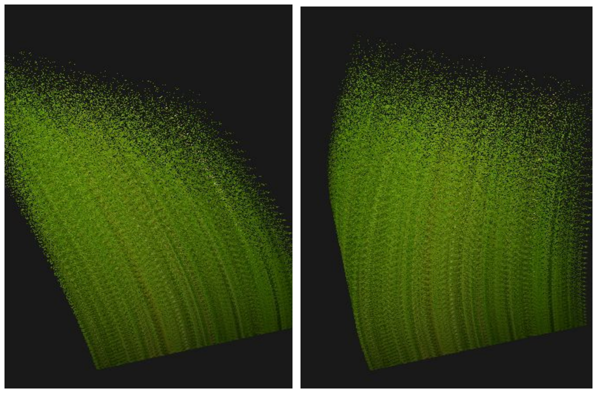
Animation

To animate the grass realistically, note that the roots of the strands don't move whereas the top is displaced the most. We compute a displacement number in each iteration:

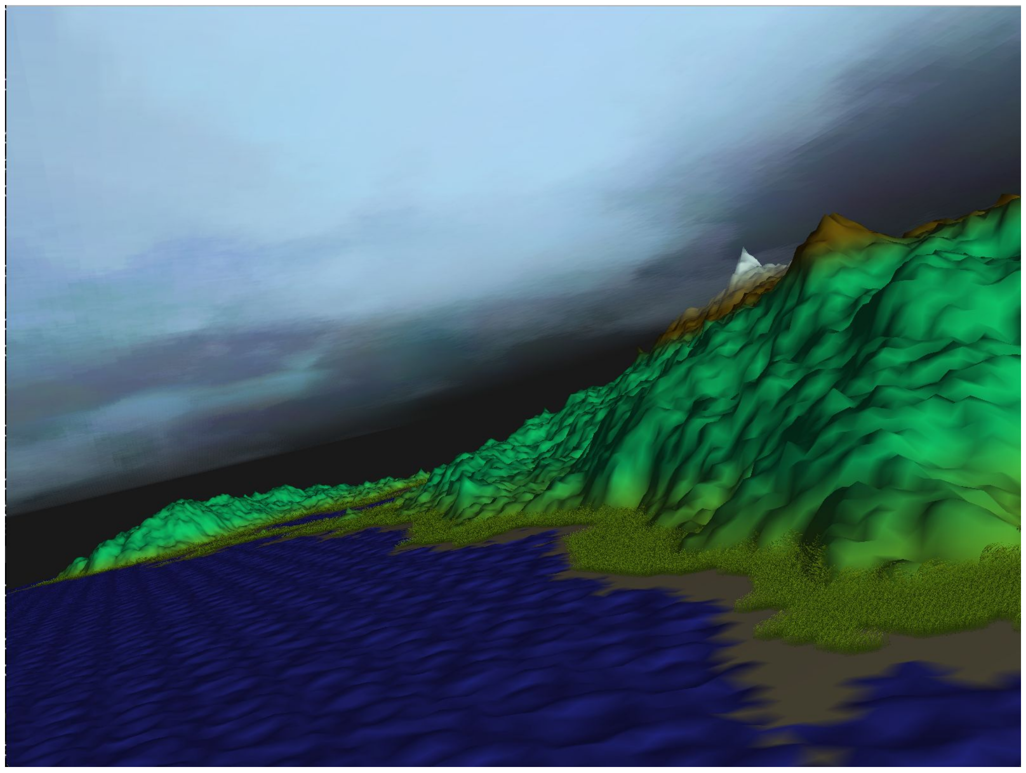
```
glm::vec3 gravity(0.0f, -0.8f, 0.0f);
glm::vec3 force(sin(glfwGetTime()) * 0.5f, 0.0f, 0.0f);
glm::vec3 disp = gravity + force;
```

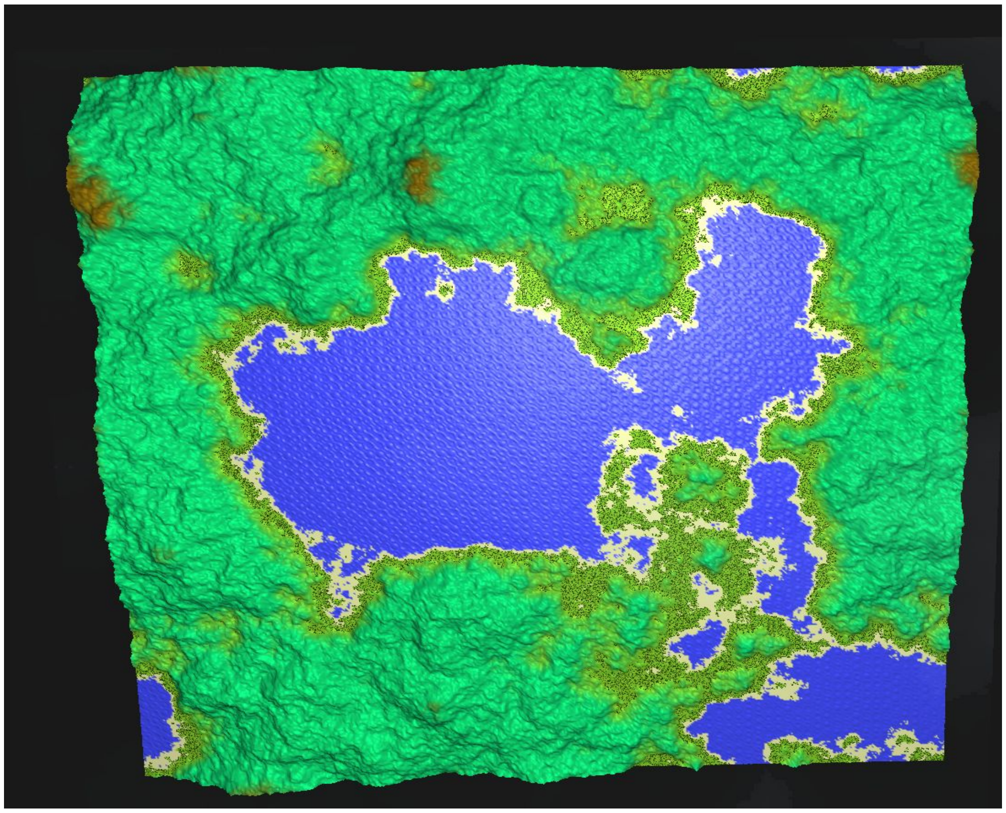
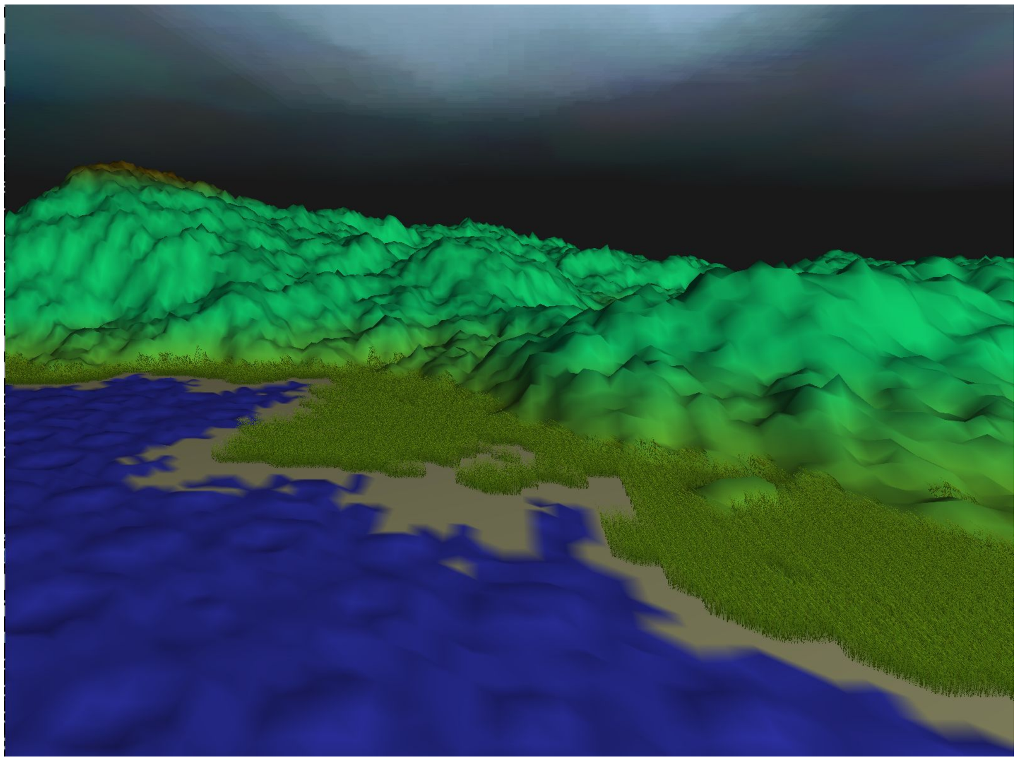
We use this in the vertex shader to move points proportionately to their layer:

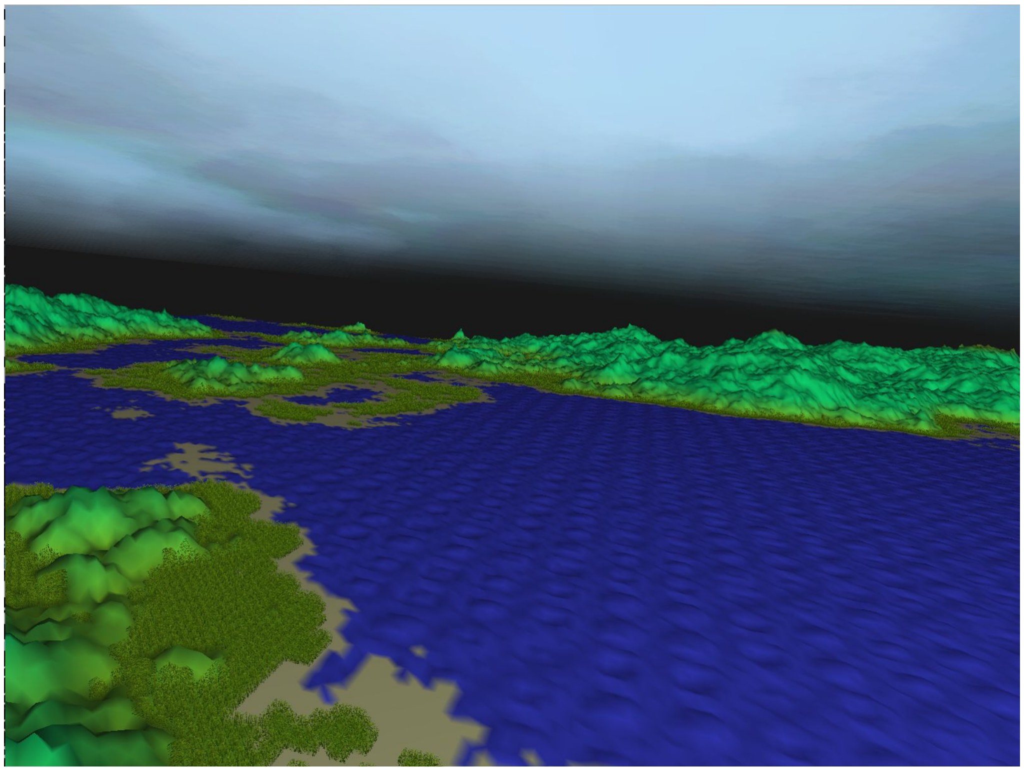
```
vec3 layerDisplacement = pow(layer, 3.0) * displacement;
vec4 newPos = vec4(pos + layerDisplacement, 1.0);
gl_Position = projection * modelView * newPos;
```



Final Result





A video showing a fly through a complete scene: https://youtu.be/ZRCC4YToGH8
A video showing grass and waves: https://www.youtube.com/watch?v=7nh_auTrte4

Code structure

(a brief high level overview)

We have organized our code in separate cpp classes like *TerrainViewer*, *CloudViewer*, *GrassViewer*. Each viewer has its own vertex and fragment shader. In *terrain_main.cpp* file we call *setup()* function for each class which does following things:

- Instantiate *Shader* based on *vertexShader* and *fragmentShader* passed.
- Calls *glGenBuffer()* on *VBO* object. Calls class member functions to generate data like points, normals, uv color and calls *glBindBuffer()* to set up buffer array in *VBO*
- Defines *vertexArrayObject VAO* and sets up input parameters in vertex shader.
- Sets constant uniform objects

Finally inside render loop in *terrain_main.cpp* we call *draw()* for each class, which does following:

- Declares which shader to use
- Sets up *modelMatrix* in *vertexShader*
- Calls *glDrawArrays(GL_TRIANGLES, 0 , num_vertices)*

Extra items

- We integrated support of assimp library to load any .obj file in our landscape. We were able to load trees using obj files from internet but it didn't look natural with other landscape elements and hence we didn't render it finally.

Running the code

We need the following libraries for the project:

- OpenGL 3.3
- GLEW
- glfw3
- SOIL
- Libpng
-

Extract the contents of the zip file. Then run,

```
cd terrain/
make
./terrain
```

You can use the familiar WSAD controls to move around the scene or the mouse to look around. You can also use the arrow keys instead of the mouse. Z and X allow you to go up and down. Press Q to quit.

Division of Labour

There were 4 main components to the project, plus the basic framework setup. We believe the work was evenly spread between both the teammates.

1. Code framework and organization: Darshan, Sagar
2. Terrain:
   a. Diamonds and squares algorithms: Sagar
   b. Shaders and interpolation: Darshan
   c. Smoothing and final touches: Sagar
3. Water:
   a. Normal computation: Sagar
   b. Animation and waves: Sagar
4. Clouds:
   a. Perlin noise core: Darshan
   b. Mapping it to a 3D box of quads: Sagar
   c. Making it cloud-like and final touches: Darshan
5. Grass:
   a. Layers of triangles: Darshan
   b. Making strands, making it look grass-like: Darshan
   c. Waving grass: Sagar
   d. Mapping the grass to the terrain: Darshan

References

We wrote most of the code in the project, except for small snippets.

[1] Perlin noise. https://en.wikipedia.org/wiki/Perlin_noise
[2] Texture generation using random noise. http://lodev.org/cgtutor/randomnoise.html
[3] Fur rendering. http://www.catalinzima.com/xna/tutorials/fur-rendering/
[4] Terragen. https://en.wikipedia.org/wiki/Terragen

[5] Nature in computer graphics.
http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/nature-in-computer-graphics-r2398
[6] Generating random fractal terrain. http://www.gameprogrammer.com/fractal.html
[7] https://500px.com/photo/101390981/home-by-mr-friks-colors-
[8] Learn OpenGL. http://learnopengl.com/
[9] Diamond and square algorithm. https://en.wikipedia.org/wiki/Diamond-square_algorithm