CMPM 163 Final Project – Rainstorm

Jay Parikh – Student, UC Santa Cruz
Jesus Hernandez – Student, UC Santa Cruz
Jacquelene Pham – Student, UC Santa Cruz

Abstract

In this paper we describe our methods of research and implementation of code using Vertex and Fragment shaders along with Three.js and other JavaScript code from websites like Codepen.io and shadertoy.com to create a Rainstorm scene. The Rainstorm consists of three different parts rain, lightning, and clouds. The rain was created by using a particle system to create an effect of rain flowing in the wind. The lightning was created using noise functions to distort geometries to create a lightning bolt shape in the vertex shader and the fragment shader was used to regularly color the bolt yellow at specific intervals. The clouds were created by layering multiples of the same image to create the puff in a cloud using a plane to keep the images together then using the fog function from Three.js to create depth.

1 Rain – Jay Parikh

When researching how to create rain I first saw how many programs created rain using particles and saw that they all had the same basic ideas. They all used a fairly simple particle system that emitted particles down the Y axis and in a certain X direction to create an illusion of rain. They also used negative space very well by limiting the number of particles on the screen at one time or else the scene would have more noise and look more like static than rain.

The process I used was to extend the GPU Particle System that we had already used in our class for the second homework assignment. The original file emits particles from a single point outward which is not the best way to create rain so I went through the entire JavaScript file to see exactly how the particles are emitted, what images are used as particles, how the colors are chosen, and many other variables used to create this specific emitter. I changed the way the particles were created from a circular shape into a cone shape that forces the particles to have a downward trajectory to move the particles in a way rain should. However, because of this cone shape, the rain did not seem very realistic because the different rain particles were moving in opposite X directions.

```
var maxVel = 2;

var velX = velocity.x + particleSystem.random() * 100000;//velocityRandomness;
var velY = velocity.y + particleSystem.random() * 100000;//velocityRandomness;
var velZ = velocity.z + particleSystem.random() * velocityRandomness;

velX = THREE.Math.clamp( ( velX - ( - maxVel ) ) / ( maxVel - ( - maxVel ) ), 0, 1 );
velY = THREE.Math.clamp( ( velY - ( - maxVel ) ) / ( maxVel - ( - maxVel ) ), 0, 1 );
velZ = THREE.Math.clamp( ( velZ - ( - maxVel ) ) / ( maxVel - ( - maxVel ) ), 0, 1 );

velocityAttribute.array[ i * 3 + 0 ] = velX;
velocityAttribute.array[ i * 3 + 1 ] = velY;
velocityAttribute.array[ i * 3 + 2 ] = velZ;
```

Figure 1: Code converted to emit particles in down the Y direction as well as the X direction to create a cone shape with the particles emitted.

Altering the rain to look as if there was wind pushing the rain in a specific direction was probably the most challenging part. After scouring the web for multiple days, even looking at the same websites multiple times, I realized that creating the rain using Three.js and GPUParticleSystem.js was undiscovered territory. Many examples of rain exist on websites like Codepen.io but they were all created using HTML and JavaScript and not Three.js so I was not sure how it would interact with the code of my partners. I then realized if I just changed the position and rotation of the particle systems to start at one corner of the scene, I could create a very realistic looking rainy scene. Because the particles were changed to emit from one corner, all the particles that the camera sees are moving in the same positive X direction and adding a partial rotation removes the cone shaped lines that are created by the particle systems. Now, the single particle system was not able to cover the entire scene which was not the effect I desired because it seemed like only few clouds were raining so I added multiple particle systems to give off the effect of wind affecting the raindrops in different ways to as well as to cover the entire scene.

```
//options2.position.x = Math.sin( tick * spawnerOptions.horizontalSpeed ) * 25;
options2.position.y = 75;//Math.sin( tick * spawnerOptions.verticalSpeed )*2;
options3.position.y = 75;//Math.sin( tick * spawnerOptions.verticalSpeed )*2;
options4.position.y = 75;//Math.sin( tick * spawnerOptions.verticalSpeed )*2;
options5.position.y = 75;//Math.sin( tick * spawnerOptions.verticalSpeed )*2;
options6.position.y = 75;//Math.sin( tick * spawnerOptions.verticalSpeed )*2;

options3.position.x =  -110;//Math.sin( tick * spawnerOptions.horizontalSpeed ) * 25;
options4.position.x =  -60;//Math.sin( tick * spawnerOptions.horizontalSpeed ) * 15;
options5.position.x =  -75;//-Math.sin( tick * spawnerOptions.horizontalSpeed ) * 15;
options6.position.x =  -45;//Math.sin( tick * spawnerOptions.horizontalSpeed ) * 25;

particleSystem2.rotation.y = Math.PI / 4;
particleSystem3.rotation.y = Math.PI / 4;
particleSystem4.rotation.y = Math.PI / 2.5;
particleSystem5.rotation.y = Math.PI / 3;
particleSystem6.rotation.y = Math.PI / 4;
```

Figure 2: Changing the position of the particle systems.

2 Lightning – Jesus Hernandez

When I was first looking at how to create lightning I thought this was going to be awesome. It quickly became a nightmare after every source I found was either too complex given my limited knowledge on shaders or too time consuming to figure out how to make given my limited time. I went from wanting to make everything in the fragment shader, after every shader toy example on lightning had incomprehensible code, to bumping the idea down to simple geometry with a glow/ bloom shader. I kept the simple geometry but got rid of the bloom shader idea after I couldn't find good sources or explanations on how to make one and at that point I couldn't waste more time so I decided that all the fragment shader was going to was color the geometry yellow. At first the geometry was going to be a simple line that I was going to manipulate but it wasn't showing up and I ended up changing it to a plane geometry instead, so my final product was a simple, vertical plane with an incredibly simple fragment and vertex shader.

In THREE.js, I created my uniforms, time was actually the only uniform and it was set 0.0. The material was a raw shader material which contained my vertex and fragment shader and uniforms. I then created my plane geometry with a width of .05 a height of 25 and 50 width segment and 35 height segments. The original plane had 50 height segments but that was causing too many points of discontinuity in the lightning so I toned it down. There is still discontinuity but it's not as frequent as it was with 50. I then created the mesh. The final part is the animate function and in the animate function I generate a random number multiplied by 10. This sets a side variable I had created earlier to -1 if it is in between 0 and 5 or 1 if it is between 6 and 9. Then I have a conditional, if the time is greater than or equal to 10.00, it gets reset. Else increment time by .025. Finally, I change the position of the geometry each time it is rendered. I change the x position by first multiplying by the side variable. If it is -1 then lightning will come from the left, if it is 1 it will come from the right. Then I generate a random number multiplied by the width of the screen divided by 5. This is kind of arbitrary and I was mostly tweaking the position calculation around until I got the right frequency of lightning on the screen so there wouldn't be too much or too little and that seemed like the sweat spot. The z position was also altered randomly then multiplied by 10 to get z position ranging from 0 to 9 so that some lightning looks closer than other.

In the vertex shader there isn't much going on just the camera uniforms and my time uniform and a position attribute. There are two functions, noiseX (float y) to create noise in the x axis of the geometry and noiseY (float x) to create noise in the y axis. Both of the noise functions are actually really simple and were adapted from a simple noise function we had done in lab section with the TA. The noise function was altered with both sin cos and tan functions and all enclosed in a fract () function. I found that without fract () the lightning wouldn't appear very jagged. Both functions are different and they're different values being multiplied. In main () all I do is assign the position attribute to a vec3 variable named pos I change the x and y position of pos by applying noiseX and noiseY to each respectively and that's about it, the geometry then gets distorted and looks like lightning.

```
float noiseX(float y)
{
    float x = 0.0;
    x += fract((sin(y * 1.0 / SCALE + time * 1.0) + cos(y * 3.3 / SCALE + time * 1.5) + tan(y * 4.3 / SCALE + time * 0.2)));
    return x;
}

float noiseY(float y)
{
    float x = 0.0;
    x += fract((tan(y * 1.0 / SCALE + time * 1.0) + cos(y * 3.3 / SCALE + time * 4.5) + sin(y * 1.3 / SCALE + time * 0.6)));
    return x;
}
```

Figure 3: Noise function used in the Vertex Shader to implement the frac () function in the X and Y attributes.

The fragment shader is even more simple, it just uses the time uniform and in main all it checks is if time is equal to 0.0 or less than or equal to 5.0. If so, the color of the lightning is set to yellow and within that 5 second window, lightning can strike. That is also why time is set to 0.0 once it reaches 10.0; so that it can cycle back up. Originally there was supposed to be infinite time and I was going to take the modulo of time and 5 and it was 0 or close to 0 then lightning would strike but that was kind of complicated and lightning was kind of infrequent so creating the 5 second window instead allowed for better chances of lightning. When time is out of that 5 second range, the lightning is set to black with an alpha channel of 0 so that it does not get in the way of the rest of the scene. The only reason this was added was so that the chances of there being too much lightning on the screen was reduced as well as the frequency of the lightning appearing.

3 Clouds – Jacquelene Pham

Before implementing the shader for cloud simulations, I researched a bit their movements and physical looks. I noticed that they are soft, distorted, and like a patch of dense fog. I came to the idea to implement the clouds using the THREE JS fog library to give depth to the clouds, as well as use an image texture to form the different patches of clouds.

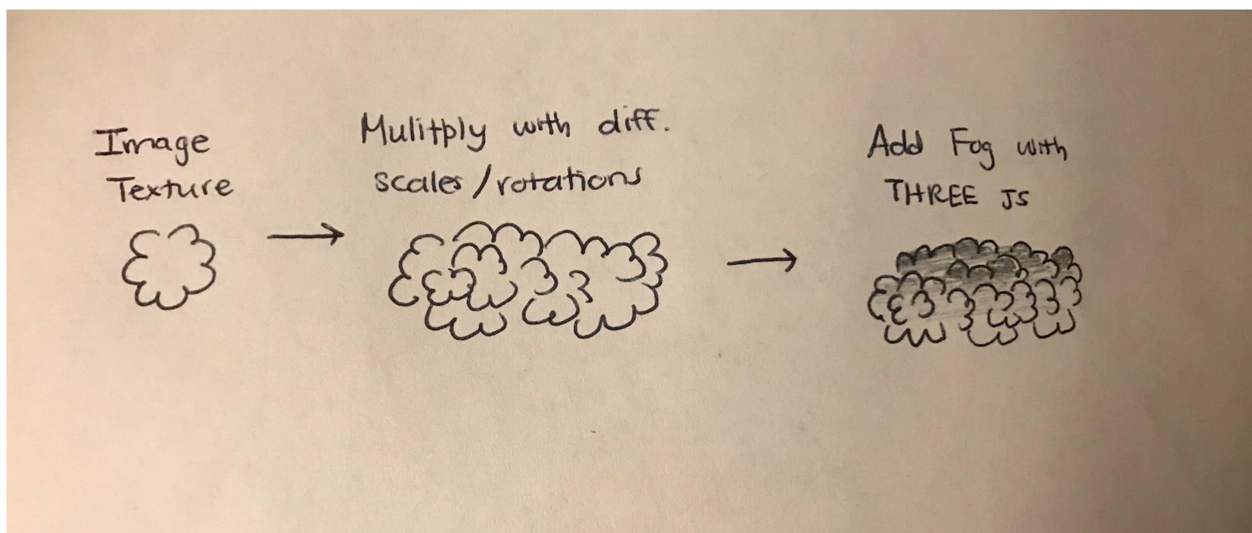

Figure 4: Process of cloud appearance

In making this cloud simulation, I had to find a cloud texture which will be used iteratively in a loop to duplicate to form a bunch of "cloud puffs". The image I was aiming for was clouds in the top screen slowly moving across. I used a texture loader to load in the cloud image, then needed to create a plane geometry for the image to be loaded on. This image would be placed on the top of the screen in a random position, with a random scaled size and with a random rotation. This randomization was repeated several times in a for-loop. By doing so, the effect would be a patch of clouds throughout the top of the screen with different sizes and rotation of puffs. A snippet of this code is shown in Figure 5.

```
var planeMesh = new THREE.Mesh(new THREE.PlaneGeometry(100, 100));
for (i = 0; i < 10000; i++) {
    planeMesh.position.x = i-1500;
    planeMesh.position.y = (Math.random() * Math.random() * 90 - 15)+700;
    planeMesh.position.z = Math.random() * 5500+50; //i+1000;
    planeMesh.rotation.z = Math.random() * Math.PI;
    planeMesh.scale.x =planeMesh.scale.y = Math.random() * Math.random() * 3.5 + 0.5;
```

Figure 5: For loop for random multiplied cloud texture

Scaling the image bigger would give the effect of softer and faded clouds. Minimizing the scaled size would give the effect of puffier and sharper clouds. There were two geometries needed to make the image appear several *separate* times as well as on the top part of the screen. A plane geometry was used to load the image onto a plan (this would be made each time in the for-loop). Another plane geometry was used to place each of these cloud image planes on one big plane so it would not be placed just anywhere on the screen. This strategy is used shown in Figure 6.

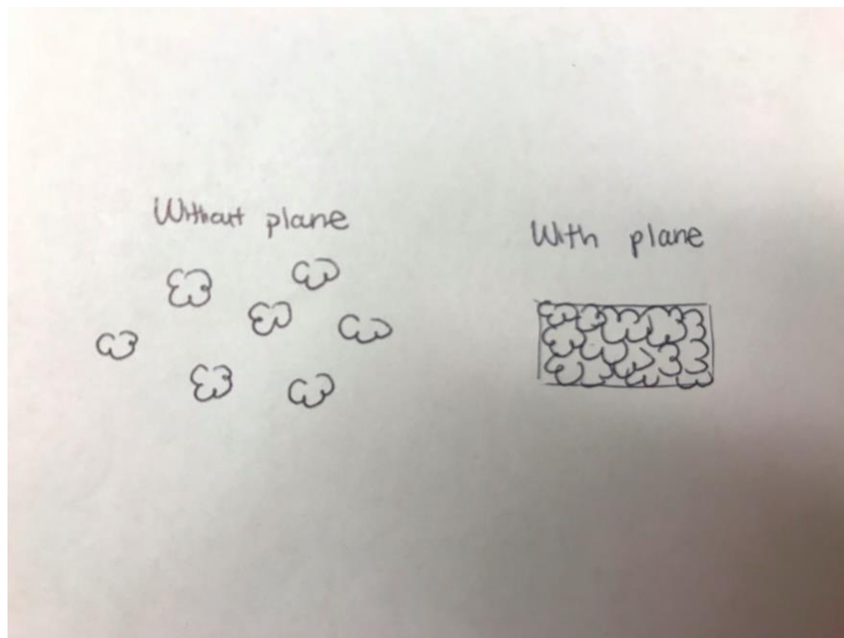

Figure 6: Plane geometry used to texture image positioning

After this process was completed, we would have a patch of clouds - However, with no depth. The next step was to use the Fog THREE JS library. This would give a layer of fog on the image plane texture to show depth - the far back of the cloud would be dark while the front would be paler.

THREE.Fog(color, minimum distance, maximum distance)

In this case, I decided to make the fog a lavender color with minimum distance being the minimum +z position of the image(front) and the maximum distance to be the -z position (the far back.). After this step, we now have a still render of a clump of clouds with fog that reveals its depth. The next step was to create the movement of the clouds. I aimed for soft movement - the clouds slowly moving across the screen and drifting up and down to show the heaviness from the rainfall. This was made by making the texture move to the left while moving in the y-direction up and down with a sine function. With these steps, I the final rendered cloud scene turned out pretty nice. If I had more time, I would make each cloud puff move independently rather than a whole.