Shader Programming: A Really Cold Place

Evan Thomson
UC Santa Cruz
ethomson@ucsc.edu

Emma Yates
UC Santa Cruz
epyates@ucsc.edu

Mark Rahal
UC Santa Cruz
mrahal@ucsc.edu

Xiaoxuan Zhang
UC Santa Cruz
xzhan209@ucsc.edu

ABSTRACT

This paper describes a 3D scene implemented with Three.js [1] and shaders with GLSL [2].

KEYWORDS

Threejs, OpenGL, bump mapping, reflection, refraction, particle systems, noise functions

ACM Reference Format:
Evan Thomson, Emma Yates, Mark Rahal, and Xiaoxuan Zhang. 1997. Shader Programming: A Really Cold Place. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97).* ACM, New York, NY, USA, Article 4, 2 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

In this paper, we will introduce the implementation details with regard to the shader techniques used to produce the expected effects. To simplify our implementation, we used Threejs library to set up everything except for our shaders.

2 WATER

We wanted to implement water that reacts to light in the same way that it does in real life. To do this, water needs to reflect some of the light, and refract some of the light. For each of these, we need to figure out what direction the light will end up traveling, and how much light has reflected or refracted.

To find out which direction the reflected light will travel, we reflect the incidence vector across the normal. To do this we use the equation

$$R = I - 2(N \cdot I)N \tag{1}$$

Where R is the reflected vector, I is the incidence vector, and N is the normal of the surface we are reflecting off of.

When light goes from one substance to another that has a different index of refraction, the light changes direction slightly, causing refraction. We can find the resultant vector using the following equation.

$$T = \eta 1/\eta 2 * (I + cos(\theta 1) * N) - N * sqrt(1 - (\eta 1/\eta 2)^2 * sin^2(\theta 1)) \tag{2}$$

Where $\eta 1$and $\eta 2$are the indexes of refraction, N is the normal of the surface, I is the incidence vector, and theta is the angle between I and N.

Now that we have values for the reflection and refraction, we need to figure out how much light was reflected versus how much was refracted. To do this, we use the Fresnel Equations. They state that the amount of reflected light is given by:

$$FR = 1/2(FR \parallel + FR \perp) \tag{3}$$

$$FR \parallel = ((\eta 2 cos\theta 1 - \eta 1 cos\theta 2)/(\eta 2 cos\theta 1 + \eta 1 cos\theta 2))^2 \tag{4}$$

$$FR \perp = ((\eta 1 cos\theta 2 - \eta 2 cos\theta 1)/(\eta 1 cos\theta 2 + \eta 2 cos\theta 1))^2 \tag{5}$$

Since all of the light must be either reflected or refracted, the amount of refracted light is equal to 1 - FR.

To add a bit more realism to the water, we wanted it to have rise, fall, and swell a bit. To do this, we used a combination of sin and cos of the current time to displace vertices on the plane of water. This gives it some nice gentle swells.

3 TERRAIN

A height map is used in the vertex shader to create our basic terrain geometry. However, the initial terrain is not perfect for the vibe of our entire scene because it is too rugged. So we need to calculate the average color of the adjacent pixels in the height map texture and use that averaged value to displace every point over the surface, in order to smooth off the rough edges of the terrain.

In addition, to cover the surface of the terrain by snow and ice to match the northern lights, we also use a normal map to fake the bumps and dents often seen on a snowy or icy surface. There are certainly other techniques that we could use for this effect, but bump mapping is a cheaper solution for the sake of speed and lower computational cost . This step is done in the fragment shader. We first extract the surface normals from our normal map texture and then calculate the lights using clamp(dot(normal,light),0.0, 1.0), in which "light" is the position vector of the light. Both the normal map and the resulting effect are shown in Figure 1.

The next step is to tweak the lights according to the general tone of the final scene, including ambient light, color of the lights and so forth. We will not go into detail here because it is all about experiments and experiences.

[1] https://threejs.org/
[2] https://en.wikipedia.org/wiki/OpenGL_Shading_Language

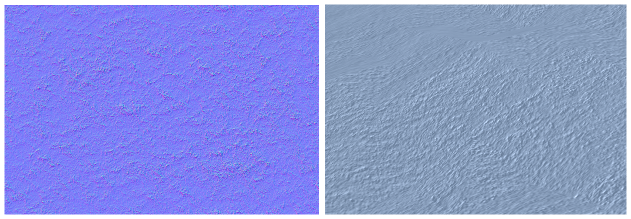

Figure 1: A normal map(left) and its resulting effect(right).

4 SNOW PARTICLES

We initially wanted to implement a shooting stars effect, but we decided to implement a snow effect instead. To do this, we needed a particle system that defined how many snow particles spawned at any given time, their translucency, their speed of descent, et cetera.

While there are numerous ways to create particle systems, we chose to create this system via signed distance functions (SDFs). Two functions were created in the fragment shader, one that determines the roundness of the particles using fractional logic, and another that draws the particles using a smoothstep function to give a faded, semi-translucent look to the particles as they fall. The numbers throughout are fine-tuned, and can be altered to provide different effects.

In the main function of the shader, the texture (which in this case is the frame buffer object of the rest of the scene, the reason for this will be discussed later) is mapped into the scene with the following code.

```
gl_FragCoord.xy / vec2(rx, ry);
 //textureBuffer is the texture of the frame buffer object
gl_FragColor = texture2D(textureBuffer, uv);
```

Afterwards, a for loop is established that loops through every snow particle in the scene. In this example, we used 200 as the maximum amount of particles allowed. In this for loop, a float we called speed determines how fast the particles fall in relation to the center, which is defined next. Once all calculations have been done, the particle is added to the gl_FragColor so that it falls over time. Here is the loop specific to our use (again, numbers are fine-tuned):

```
for(int i = 0; i < 200; i++) {
j = float(i);
float speed = 0.01+rnd(cos(j))*(0.7+0.5*cos(j/(float(210)*0.25)));
vec2 center = vec2((0.25-uv.y)*0.1+rnd(j)+0.1*cos((time*0.05)+sin(j)),
        mod(sin(j)-speed*(time*0.1*(0.1+0.2)), 1.0));
gl_FragColor += vec4(0.65*drawCircle(center, 0.002+speed*0.012)); }
```

The reason the scene is rendered this way is due to the nature of the scene. Since the rest of the scene was created without the use of SDFs, it is necessary to create a texture out of the scene so that way SDFs can be rendered onto it. Thus, the entire scene is created in a buffer and a texture of the buffer is created and applied to this shader for use.

5 AURORA BOREALIS

We wanted to be able to include a aurora effect and to implement it using a skybox. The greatest challenge with this effect was cheaply simulating a large scale, dynamic 3D natural phenomenon on a 2D box surface. In order to do this cheaply, we used a combination of ray marching and noise functions to create a deceptively voluminous effect. Because there is no actual volume to the surface of the box, the auroras are generated in the fragment shader. Here, we establish a background color (in this case a dark blue), a ray origin (or the buffer camera position since we first render to an off screen buffer), and ray direction. Using the aurora() function, we update the fragment color based on the origin (*ro*) and direction (*rd*) of the ray using a ray marching for loop:

```
for(float i=0.0;i<20.0;i++) {
        float pt = ((.8+pow(i,1.4)*.002)-ro.y)/(rd.y*2.+0.4);
        vec3 bpos = ro + pt*rd;
        vec2 p = bpos.zx;
        float rzt = triNoise2d(p, 0.06);
        vec4 col2 = vec4(0,0,0, rzt);
        col2.rgb = (sin(1.-vec3(2.15,-.5, 1.2)+i*0.043)*0.5+0.5)*rzt;
        avgCol = mix(avgCol, col2, .5);
        col += avgCol*exp2(-i*0.065 - 2.5)*smoothstep(0.,5., i);
}
```

We can see the fragment color (*col*) is updated using a combination of the average color (*avgCol*) and a generated color (*col2*) whose rgb values are based on the ray depth (*i*) and has an alpha value (*rzt*) created by the continuous noise function triNoise2D(). This function takes time as an argument and updates accordingly based on other smaller noise functions but deeper detail would be trivial to this explanation

In short, the alpha values of the auroras are generated based on time and rgb values come from the distance into the scene. The "shapes" of the auroras are therefore nebulous - the ribbons are not defined by a shape or function. Instead, the effect is closer to a dynamic 3D pattern.

6 CONCLUSIONS

We have demonstrated in this paper the development details of a 3D scene: A Really Cold Place. Several techniques were used to create the four elements (water, terrain, aurora, snow) in the scene, including reflection, refraction, bump mapping, particle systems and noise functions.

REFERENCES