Attempting to Create a Desert Oasis Using Three.js

Alfredo Cuevas
CMPM 163
UCSC
acuevas5@ucsc.edu

Robert Gaines
CMPM 163
UCSC
rngaines@ucsc.edu

Trenten Kaufeldt-Lira
CMPM 163
UCSC
tkaufeld@ucsc.edu

ABSTRACT

We set out with the goal of creating a desert oasis that featured video game style water, shifting sands, dynamic clouds, and procedural plant generation. Unfortunately, we were not able to fully implement our vision, however, the final results increased our knowledge of working with shaders.

KEYWORDS

Water, Sand, Clouds, Desert, Oasis

1 INTRODUCTION

Initially, the goal was to try to make a video game style desert oasis. The final product wouldn't be photo realistic but wouldn't be out of place in a simple video game. The final version of the desert only has water, sand, and clouds. Unfortunately, time complications made it difficult to work on the procedural plants. This is unfortunate because the water looks less impressive with nothing to reflect. The water that was created manages to implement reflection and refraction. It shows ripples that move and has spectacular highlights. The sand that was created shifts over time and creates a crater in the center of the ground. Finally, we use a skybox to fill the sky. The clouds in the sky will be created using textures that make them look like they are moving and changing.

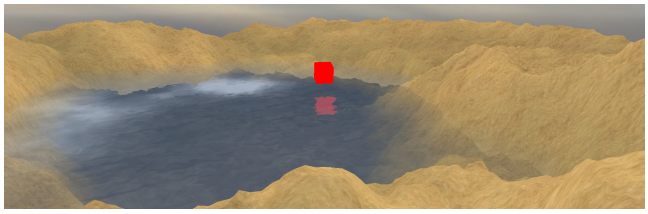

Figure 1: Close up of the water and sand for the final version of the project

The rest of this paper will explain how the team went about creating the water, sand, and clouds. The water was implemented by Alfredo Cuevas. The sand was implemented by Robert Gaines. Finally the clouds were implemented by Trenten Kaufeldt.

2 ALFREDO CUEVAS: CREATING WATER

2.1 Inspiration

The inspiration for creating this water came from looking at the slides in *Water Flow in Portal 2* by Alex Vlachos. (http://www.valvesoftware.com/publications/2010/siggraph2010_vlachos_waterflow.pdf). The slides went over minimizing the performance impact by applying normal maps and color maps to a plane. My goal was to create a very simple version of this. I got a lot of help by looking at guides on YouTube, especially videos by Thin Matrix. He used OpenGL to make water with a plane. (https://www.youtube.com/channel/UCUkRj4qoT1bsWpE_C8lZYoQ).

2.2 Frame Buffer Objects and Clipping Planes

The first step to creating the water is to make a flat water plane. Then make two Frame Buffer Objects. These will be used to create two textures; one is for reflecting objects above the water and one is for refracting objects below the water. All objects in the main scene should be used in the Frame Buffers except for the water itself. The reflection texture will be slightly different from the refraction texture in its camera position. When rendering the reflection texture the camera should be moved down by two times its height distance from the water plane. Then the pitch of the camera should be inverted so that it is looking in the same direction that it was looking at before it was shifted.

Two different clipping planes will be used for this project. One clipping plane will be used during the reflection. This clipping plane should make it so that everything below the water is removed from the scene. The clipping plane used in the refraction will remove everything above the water.

Once the two textures are created they should be passed to the water's shaders during the main rendering. These textures will be used to create the effect of objects reflecting off the water and looking through the water. When reading the textures it is important that you use Normalized Device Coordinates (NDC), not the regular UV values. You can get NDC in the fragment shader by doing the following:

$$ndc = (clipSpace.xy / clipSpace.w)/2.0 + 0.5$$

The clip space will be the same value given gl_Position so you need to pass this into the fragment shader as well. It is also important that you invert the values used to read the reflection texture because you will want the texture to be flipped vertically.

2.3 Ripples using a DuDv map

The ripples in the water will be created using a dudv texture. The dudv values will be used to create distortions in the water. This is done by reading in the texture values and then adding them to the Normalized Device Coordinates. It is important to note that the dudv values will all be greater than 0. So after you get them from the texture you should multiply by 2 and subtract 1. This will give you a range of -1 to 1. When calculating this final distortion you can also multiply by a predefined value. This will increase or decrease the strength of the ripples.

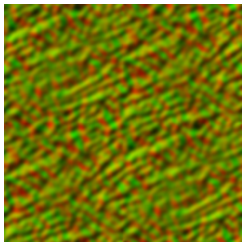

Figure 2: The DuDv texture used in creating our water

2.4 Fresnel Effect

Now that we have read the reflection and refraction textures using our distorted ndc coordinates we can mix them. However, instead of mixing them by a static value we can mix them based on the angle between the camera and the surface normal of the water. First we need to get the vector from the camera to the water plane. This is done in the vertex shader by taking the world position in the scene and subtracting it from the camera's position. We then take the dot product between the camera vector and the normals of the water's surface. The normals for the water can be defined as a vector3 of value (0.0, 1.0, 0.0). However, in our case we used a normal map to try to achieve a better Fresnel effect. To do this you need to get a normals texture and use the values to define the normals for the surface of the water. You read the values using the distortion value from before.

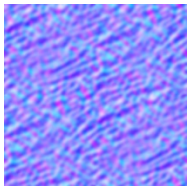

Figure 3: The normal texture used in our water.

2.4 Specular Highlights

Now we will add specular highlights using the same normals that we talked about in the previous paragraph. To do this you need to get the vector from the light source to the surface of the water. You do this in the vertex shader, similar to how we got the camera vector. However, this time we subtract the light position from the world position. Once you have this light vector you need to normalize and reflect it using the normals that we talked about before. You then need to get the max of the dot product between this value that we just got and the camera vector. This will give you the value we are calling specular. Next we will take specular and raise it to a predefined value (shine damper). Then finally we take this value and multiply it by the values of lightColor and reflectivity which are predefined values. This will give us the final value of specularHighlight which we can add to the gl_FragColor. Before we actually add specularHighlight to the gl_FragColor we can mix gl_FragColor with a blue color vector 3 to give our water a tint of blue.

2.4 Soft Edges using a Depth Texture

Using the refraction frame buffer object we can pass in a depth texture into the water shader. We will use this texture to determine the depth of the water from the ground. This way we can increase the transparency of the water the smaller the water depth is. This will make it look like the water is slowly blending with the ground instead of just stopping obruptly. Here is some pseudo code for how to go about reading the depth:

1)Float near = camera near value;
2)float far = camera far value
3)float depth = textures2D(depthTexture, refractionTexCoords).r;
4)float floorDist = 2.0 * near * far / (far + near – (2.0 * depth - 1.0) * (far – near));
5)depth = gl_FragCoord.z;
6)waterDist = 2.0 * near * far / (far + near – (2.0 * depth - 1.0) * (far – near));
7)waterDepth = floorDist – waterDist;

…

8)gl_FragColor.a = clamp(waterDepth/ 8.0, 0.0, 1.0);

After this you should be done with water that reflects/ refracts, ripples, shimmers, and softly blends into the sand.

3 ROBERT GAINES: ACTUALIZING SAND

The sand map was made by layering a height map for the oasis pool and two more to represent flowing sand dunes. Each are added together to give the final height for each vertex, then textures matching each moving sand part of the height map are applied.

3.1 Vertex Shader

The vertex shader for the sand takes in a 256 by 256 plane and changes the heights of the sand by taking in the combined intensity of the color channels of three different textures. The first is a static texture which acts as a base to make the center of the map deeper than the edges for the oasis. The second and third maps are both repeating textures to represent the actual sand moving across the map. Each is a bump map version of a texture passed into the fragment shader so they match when rendering. The second map makes up the large waves in the scene, and the third map simulates smaller lumps of sand making their way through the map more quickly, as if the wind is blowing.

Each sand's texture's UV mapping is offset by a variable called sandSpeed, which is incremented upward each frame from the sand object file, so the UV map shifts. The vertex's height value (which is conveniently Y in Three.js, and Z in GLSL) is then offset by the combined RGB value of the closest pixel on the bump map. The height added from each texture is weighted, with the base heightmap and the dunes modifying the height significantly more than the fast moving sand lumps. Then the total offset is added to the actual position of the terrain plane in the scene. Finally, the positions of the vertices are multiplied by the model matrix, view matrix, and projection matrix, which respectively apply world coordinates to the map, rotate the map towards the camera, and decides how map the scene to the screen.

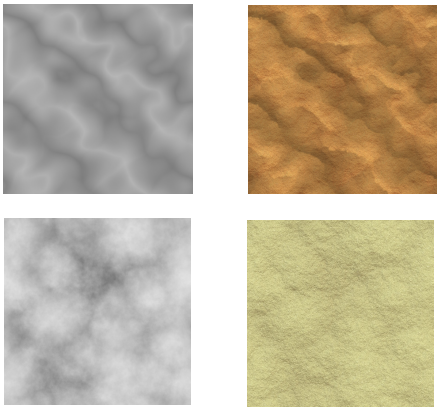

Figure 4: Images on the left are bump maps for vertex shader, versions on the right are the actual textures people will see.

3.2 Fragment Shader

The fragment shader is much simpler than the vertex shader and applies the same movement transformations as for the second and third textures for the vertex shader so the sand looks like it moves across the scene. Then each texture is mixed together, with the large dune texture adding depth to the scene with the peaks of the dunes being lighter than the pits. The default dune texture is actually black and white, so I mixed it together with a repeating sand texture in a photo editor and gave it a reddish tinge to make the scene more interesting. The end result of mixing the dune and repeating sand texture makes it look as if the sand is flowing, while still accentuating the dunes themselves.

4 TRENTEN KAUFELDT: MAKING CLOUDS

The clouds were a product of multiple attempts to make noise clouds. First using simplex noise, then Perlin nose and finally making use of a pre-rendered file in Photoshop with multiple different maps to try out multiple kinds of clouds. The clouds are intended to slowly drift around overhead.

4.1 Fragment Shader

The clouds created make use of a noise map pre-rendered in Photoshop. This cloud map is simply culled through the usage of the following assignment:

gl_FragColor.a = gl_FragColor.r;

This single line creates an alpha map. An Alpha map is a map used for making textures vary in transparency across themselves instead having a uniform level of transparency. Such effect is used frequently in game software- one example being chain link fence textures. To simulate the illusion of seamless (if looping) movement, the cloud map was made to slowly rotate around the world from an off-center fixed point. The vertex Shader does very little beyond reading in the UV coordinates for the texture map while the fragment Shader applies the texture map and tweaks the output to make darker colors disappear. It does this linearly and is wholly reliant on the r value of RGB. This is why a monochromatic map is needed- to maximize control over how opaque the texture can get. The cloud texture was also made two sided so that they can be looked at from above or below.

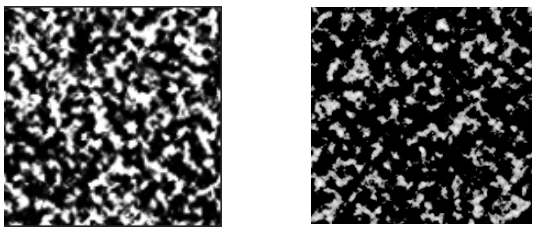

Figure 5: The texture above are two pre-rendered noise maps used for the clouds. The one on the right was made to deal with the issue of the clouds appearing to bright.

4.2 Pre-Rendering

The process for creating the noise clouds in Photoshop is very quick once you know what to do. First, I made a large 4096x4096 pixel document, then I made sure to make the document monochromatic. With the monochromatic document, I than rendered the noise clouds. Initially they don't look that cloud-like, so additional brushing up is done. Mainly, using Photoshop's levels function to make more obvious contrast between the clouds and sky. It also gives the clouds a bit of a fluffy, organic look. Doing this gives the noise a much more cloud-like appearance than it otherwise would have. The pre-rendered noise is than imported as a texture directly, applied to a two sided plane. It is a very simple procedure.

5 CONCLUSIONS

In summary, we worked toward creating a desert oasis using shaders and three.js. We attempted to create water, sand, and clouds that would fit in a video game setting. Through our work we learned more about different techniques in creating graphics with shaders. We learned how to solve our problems by finding resources online and asking for help from other classmates.

ACKNOWLEDGMENTS

We would like to thank Professor Forbes and our TA Lucas Ferreira for helping us along the way. Their advice and sample code helped us in getting started with the project and solving problems that arose.

REFERENCES

[1] "ThinMatrix." YouTube. YouTube, n.d. Web. 20 Mar. 2018.

[2] Vlachos, Alex. "Water Flow in Portal 2." Valve Software, www.valvesoftware.com/publications/2010/siggraph2010_vlachos_waterflow.pdf.