CMPM163 Final Project ~ Venice Simulation

Jonathan Chuang
jrchuang@ucsc.edu

Eisaku Imura
eimura@ucsc.edu

Matt Loyola
mjloyola@ucsc.edu

ABSTRACT

Creation of a Google-like ground view of the city of Venice using GLSL shaders comprised of a mixture of simple and complex noise functions and mathematical formulas. Heightmaps are generated using open source data to map the real zone of venice to a physical model projected a scene. The overarching goal of this project was to produce a scene that was interactable, visually interesting, and shows off some of the graphics programming practices and techniques taught over the period of this course.

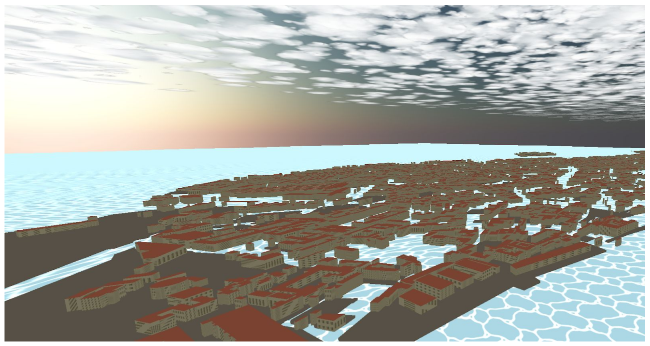
SHADER CONCEPTS

• Procedural Geometry
• Noise Functions
• Buffer Geometry
• Raymarching
• Lighting

KEYWORDS

Clouds, Systems, Cycle, Lighting, Object, Loader, Geometry

1 INTRODUCTION



2 CREATIVE VISION AND TECHNICAL CHALLENGES

2.1 Project Vision

The overall vision of this project was to have a real time simulation of Venice, Italy. Of course due to time constraints and limited experience working with shaders, we tried the best of of our ability to recreate this majestic city within a virtual world. To create Venice, is stating an ambitious task. We would have to take in a lot of factors such as people walking around the streets, boats going through the canals, etc. Again however, those are tasks that are way out of our scope and we decided to focus on three main goals. The three main goals were to have a basic geometry of the city that is textured, a sky that is dynamic with a day and night cycle, and of course the clouds and water that surround the city.

2.2 Challenges in Developing the Shaders

2.2.1 One major challenge in developing the clouds was to resist implementing a system that looked really nice but sacrificed too much performance. I had to keep in mind that the other elements of our larger scene were not yet in place and deciding to go overboard with either too many clouds or higher density clouds (such as the raycasting example from ShaderToy) would have caused performance problems down the line that would have been more of a hindrance to the flow of our scene, even if it *looked* better.

2.2.2 In developing the water texture, we thought about incorporating a reflective/refractive shader similar to the one used in homework 2 and in the fresnel shader used by Mr. Doob. This idea quickly took a back seat when we compared the “labor costs” with what it might realistically add to the scene. Instead we opted for a modified version of the Zelda water shader with noise displacement. There is enough motion in the texture to distract from the fact that the water doesn’t actually reflect the surroundings.

2.2.3 A major problem designing the sky was whether to be lazy and just create a sphere with a light glowing around it or designing a shader that implements rayleigh scattering and mie scattering. The former was tested and completed, but after finishing it in an hour or two, a decision was made to go for the latter since it was decided the former looked really bad.

3 SCENE GEOMETRY ~ Jonathan Chuang

3.2 Parsing OpenStreetMap

In order to build a semi-accurate recreation of Venice, Italy, an accurate data source for building information was necessary. OpenStreetMap (OSM) offers a public API for their database of user-submitted building and region information, which can be easily accessed via an HTTP request for an XML file.

The XML file returned is a blend of many different parts of their public map data - bike paths, neighborhood designations, accessibility ramps, waterways, notable landmarks, and of course, building data. Unfortunately, much of this (particularly building) information is partially labeled, meaning that blacklisting strategies far more effective than whitelisting specific tags. Of the 237 tags used, a total of 23 tags were blacklisted - most often for representing intangible regions or paths.

The parsing process was split into two runs - one for low elevation features (landuse, amenities, leisure, islet/islands) and one for primary features (predominantly buildings).

3.3 Constructive Geometry

Geometry construction has several steps, but for both low elevation geometry and primary feature geometry, almost the same process was utilized, with the exception being that for low-elevation geometry, a lower height was used.

OSM's XML request contains longitudinal and latitudinal data for every major edge (or vertex pair) of every logged building within the requested region. Using this data, a 3D model in OBJ format can be generated in a few steps:

It's important to note that the OBJ format requires a few things for full functionality: a list of vertices, their normals, and corresponding texture uv coordinates, and then subsequently a list of indices for the aforementioned vertices, normals, and texture uv coordinates.

To generate the roof of every building, a random (but proportionally reasonable) height is selected and a ring of vertices at that height are buffered. Using earcut and earclipping strategies, an array of vertex indices can be generated to triangulate the requested rooftop. Being roofs, only a single is needed for all buildings, as non-flat rooftops were not covered in the scope of this work.

To generate the walls for every building, the first step required is to determine whether or not the input list of coordinates is listed in clockwise or counterclockwise order. This is vital to ensuring that all faces face outwards.

This task can be accomplished using the shoelace algorithm, otherwise known as Gauss's area formula, whose output value taken absolutely measures the inner-size of any convex or concave poly. The shoelace algorithm also informs the direction of the input vertices, giving the information needed to have outward facing walls.

Using several cases of the distance formula, tiled texture uv coordinates can be assigned to each unique vertex, which will be later used in texture mapping to generate building windows. Due to the flat, rectangular nature of the geometry, averaging vertex normals does little to improve visual fidelty, Gouraud-like shading was deemed sufficient for the purpose of the work.

After buffering all of the previous data, an obj can be written and then further checked and linted for accuracy.

3.4 Texture Mapping

Texture mapping was approached relatively conventionally - although specific texture coordinates were generated to promote fitted tiling along the walls of every building.

One other strategy employed was using the normal to identify rooftops for a secondary material. Since all rooftops are flat and no other surface and point in a similar direction, a quick normal check in the fragment shader can be employed to identify and render surface-appropriate materials.

3.5 Lighting Approximation

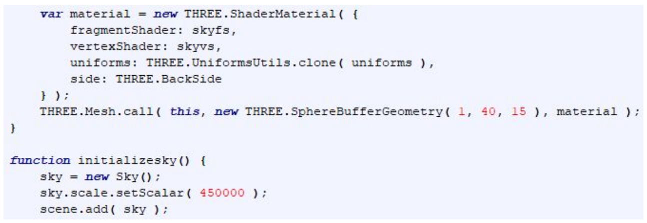
Gouraud shading was implemented to light the various buildings across the environment. In order to very roughly simulate the day/night cycle on the buildings themselves, the positions of the sun and moon are delivered as uniforms to each building. When the sun is "active" (above the horizon), the shading model changes to match the sun's more unique characteristics. The same idea is abstracted to the moon, where a slightly bluer hue is painted directly to each surface.

While not a particularly accurate lighting solution, this works as an effective stop-gap in aligning with the day/night feature.

4 LIGHTING AND DAY/NIGHT CYCLE ~ Eisaku Imura

4.1 Creating the Skydome

Before starting on the sun and moon, I had to debate on whether to use a skydome or a skybox. Both have its pros and cons, and ultimately it came down to how I wanted to implement the sky. The skybox is just a simple box that is very fast to draw with fewer vertices than a skydome. However, its downside is that it can have issues with perspective and corners which can look rather ugly when you want a "grand" atmosphere. On the other hand while a skydome is slower to draw with far more vertices, this method allows me to have more options such as coloring the vertices for different effects and not having to worry about perspective issues.



```
    var material = new THREE.ShaderMaterial( {
        fragmentShader: skyfs,
        vertexShader: skyvs,
        uniforms: THREE.UniformsUtils.clone( uniforms ),
        side: THREE.BackSide
    } );
    THREE.Mesh.call( this, new THREE.SphereBufferGeometry( 1, 40, 15 ), material );
}

function initializesky() {
    sky = new Sky();
    sky.scale.setScalar( 450000 );
    scene.add( sky );
```

Figure 1: Implementing the skydome

At the end of the day, I decided to use the skydome so that there aren't any issues I mentioned before. The skydome is implemented by simply creating a THREE.SphereBufferGeometry, applying the material with the sky shader, and then scaling it to a large size.

4.2 Sun and moon

Now that we decided on using a skydome, it is time to create the sun and moon along with the atmosphere. I initially debated on whether to create a sphere geometry and make it glow or just create the sunlight effect in the shader itself. Ultimately, I chose to just implement an effect called rayleigh scattering and mie scattering so I can use one shader for the skydome.

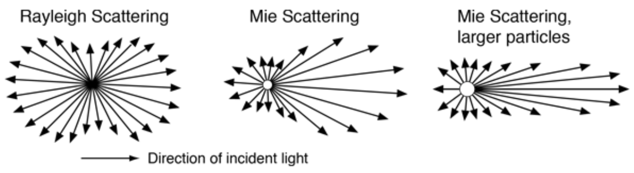
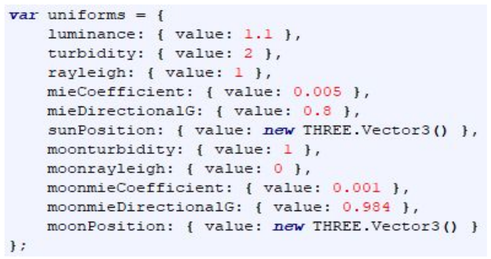


Figure 2: Rayleigh and mie scattering

It simply didn't make sense to make two or three separate shaders for the skydome, sun, and moon respectively. Rayleigh scattering is responsible for giving us the blue color of the sky caused by the scattering of sunlight off the molecules of the atmosphere and mie scattering gives us the white glare around the sun.

Figure 3: Rayleigh and mie scattering in effect

Before I go over the vertex and fragment shader, I also want to mention or put out a reminder that there are two sources of light in this shader which are the sun and moon. Both are implemented at the same time with only a few key minor differences such as the position of the sun and moon being in the opposite side of each other.



```
var uniforms = {
    luminance: { value: 1.1 },
    turbidity: { value: 2 },
    rayleigh: { value: 1 },
    mieCoefficient: { value: 0.005 },
    mieDirectionalG: { value: 0.8 },
    sunPosition: { value: new THREE.Vector3() },
    moonturbidity: { value: 1 },
    moonrayleigh: { value: 0 },
    moonmieCoefficient: { value: 0.001 },
    moonmieDirectionalG: { value: 0.984 },
    moonPosition: { value: new THREE.Vector3() }
};
```

Figure 4: Uniforms for the shader

The major differences besides the position are the rayleigh and mie values that are inputted to the vertex shader. I set the rayleigh value for the moon to be zero so that there is no "light" scattering across the atmosphere giving it a dark sky which is what we want for a nighttime atmosphere. I also set the mie value to be very low so the moon only gives off a slight glow compared to the sun during daytime.

4.3 Vertex and fragment shader

The vertex shader is pretty simple and it only contains a few calculations needed for the fragment shader. First, I got the direction of the sun and moon by normalizing the vectors for the position of the sun and moon. Then I calculated the Euler value for both by calling a function called “sunIntensity”.

```
float sunIntensity( float zenithAngleCos ) {
    zenithAngleCos = clamp( zenithAngleCos, -1.0, 1.0 );
    return EE * max( 0.0, 1.0 - pow( e, -( ( cutoffAngle - acos( zenithAngleCos ) ) / steepness ) ) );
}
```

Figure 5: sunIntensity function

This function basically takes the angle of the sun or moon from the camera and returns a float which will be needed in the fragment shader later on. Next, I calculated the value for when the sun or moon will fade which is given by using Euler’s number with the position of the sun or moon divided by the max height of the skydome. I then clamped that value to from zero or one and subtracted it by one. Afterwards, I got the coefficients for the rayleigh of the sun and moon by subtracting the uniform rayleigh value with the fade value I just calculated. However, that is not the final value we want and we need to do two more steps. I needed to multiply the rayleigh coefficient with the total rayleigh scattering coefficient to get the rayleigh coefficient we want.

$$\beta_m = \frac{8\pi^3(n^2-1)^2}{3N\lambda^4}\left(\frac{6+3p_n}{6-7p_n}\right),$$

Figure 6: Formula for total rayleigh

Finally, I needed to get the mie coefficient we want by getting the uniform turbidity, putting it through a function called “totalMie” which returns a vector, and multiplying it by the uniform mie coefficient.

```
vec3 totalMie( float T ) {
    float c = ( 0.2 * T ) * 10E-18;
    return 0.434 * c * MieConst;
}
```

Figure 7: totalMie function

Now that the vertex shader is done, I will go over the fragment shader. Here, things get a little bit complicated and can be overwhelming with fancy equations to emulate the colors of the sky. I will go over the most important bits of the fragment shader so please refer to the figures for variables that may appear in certain functions or formulas.

First, I would like to talk about the Rayleigh phase function. The rayleigh phase function is a phase function that is applicable to rayleigh scattering of unpolarized radiation. It is basically a function that describes the dependence of scattered radiance on scattering angle. It is commonly expressed as taking the cosine theta to the power of two, adding one, then multiplying it by three over four where theta represents the scattering angle. Cosine theta is determined from the dot product of normalized world position subtracted by the camera position and the direction of the sun or moon. We then take the return value of the rayleigh phase function and multiply it by the rayleigh coefficient we calculated in the vertex shader.

$$F_R(\theta) = \frac{3}{4}(1+\cos^2(\theta))$$

Figure 8: Rayleigh phase function

The next important function is the Henyey-Greenstein phase function and I used this function for mie scattering. The function takes in cosine theta, which we already have, and a uniform value called “mieDirectionalG”. The function should return a float which is then multiplied by the mie coefficient which we also calculated in the vertex shader.

$$p(\theta) = \frac{1}{4\pi}\,\frac{1-g^2}{[1+g^2-2g\cos(\theta)]^{3/2}}$$

Figure 9: Henyey-Greenstein phase function

Finally, we use these two vectors to calculate the total light scattered into the viewing direction as in figure 10.

```
vec3 Lin = pow( vSunE * ( ( betaRTheta + betaMTheta ) /
( vBetaR + vBetaM ) ) * ( 1.0 - Fex ), vec3( 1.5 ) );

Lin *= mix( vec3( 1.0 ), pow( vSunE * ( ( betaRTheta + betaMTheta ) / ( vBetaR + vBetaM ) )
* Fex, vec3( 1.0 / 2.0 ) ), clamp( pow( 1.0 - dot( up, vSunDirection ), 5.0 ), 0.0, 1.0 ) );
```

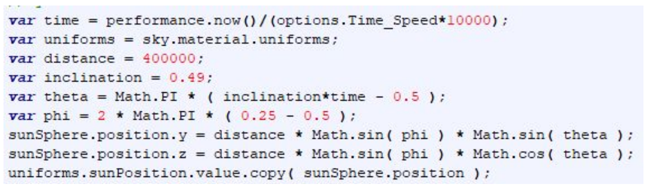
Figure 10: Calculation to find total light scattered

After finding the total light scattered into the viewing direction, we need to find the extinction factor so that we can add with our last value we calculated. The extinction factor is found by getting the Euler number of negative rayleigh coefficient multiplied by the inverse of the rayleigh zenith length and then added by the multiplication of the mie coefficient and the inverse of the mie zenith length.

Finally, the total light scattered and the extinction factor is added to get the color of the sky. Then, I apply some finishing touches such as using a tone mapping from Uncharted 2 to reach the goal of adding together two vectors in gl_FragColor to complete our shader since we have the sun and moon.

4.4 Sun and moon movement

The movement of the sun and moon over time was done by creating a timer in the render function. I made it so that the inclination is multiplied by the time which will give us theta. The theta will give us values that complete a full circle. Theta is applied into the y and z position of the sun and moon which will update and gives us the feature we desire.



```
var time = performance.now()/(options.Time_Speed*10000);
var uniforms = sky.material.uniforms;
var distance = 400000;
var inclination = 0.49;
var theta = Math.PI * ( inclination*time - 0.5 );
var phi = 2 * Math.PI * ( 0.25 - 0.5 );
sunSphere.position.y = distance * Math.sin( phi ) * Math.sin( theta );
sunSphere.position.z = distance * Math.sin( phi ) * Math.cos( theta );
uniforms.sunPosition.value.copy( sunSphere.position );
```

Figure 11: Changing position over time

5 CLOUDS AND WATER ~ Matt Loyola

5.1 Developing a Cloud System

There were a number of ways I could have implemented a cloud system, some looked more visually appealing than others but would have required exponentially more time and debugging to successfully integrate into the Three.js shader environment.

On that note, I ended up settling for an interesting method I read about online which involves drawing individual flat clouds using a basic cloud sprites, running various formulas to vary their size, shape, and position, and then merging them together into one geometry. Evidently, this saves a decently significant amount of GPU processing time and power because each individual element is only being rendered one into the larger, combined texture. The downside to this method is that the each cloud cannot be modified during runtime - only the larger cloud object itself can be moved gradually. Luckily in the case of this implementation, the only runtime movement I required was the ability to slowly move this large cloud plane across the sky to emulate the slow movement of clouds with the wind.

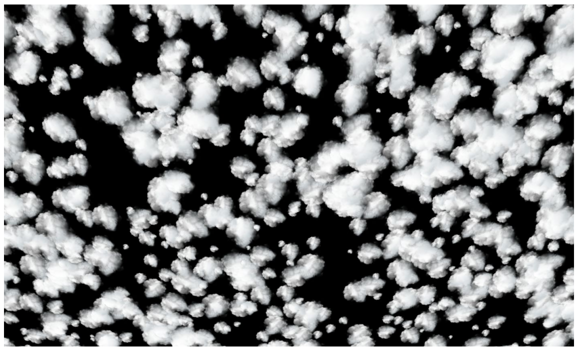
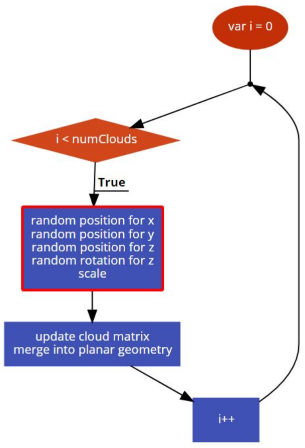


Figure 1: Looking up at the cloud plane.

The shader file for this is very simple and only involves a standard varying vector 2 for the UV. The fragment shader requires a sampler2D specifier for the input texture, in this case a cloud png, which is then run into the texture2D() function and set to the fragment color.

In the initializer script within the main HTML file, the cloud object starts off with a basic setup of a new THREE.ShaderMaterial() method which contains the uniforms (which is just the texture for the map), a link to the vertex and fragment shaders from the external file, and a "true" value for transparency. We want the cloud texture to be transparent so that some of the light from the sun and moon seep through into the scene - also it looks much more natural and similar to the way real clouds look.

The bulk of the code is in the for-loop and the variables which control what it does. Specifically, you have a this loop running from some counter variable until it hits the maximum set number of clouds that you've decided upon. For each cloud in this set, positional data (x, y, z) are set using Math.random() to achieve a natural sense of spacing and scattering of the sprites. Scale is also modified because of course not all clouds are the same size. Then, the cloud is merged into a plane geometry created outside the loop using Three.js's merge() method which takes the geometry data



and the matrix data in order to translate that one cloud into the plane. Exiting the loop, all that's left to do is create a mesh for the cloud which is made up of the merged plane geometry and the materials defined above, and then add the cloud to the scene.

5.2 Wave Motion in a Water Shader

Similar to the cloud system, there are many ways to do a water shader with a general fact being that the more realistic and attractive you want your water to look, the more time and processing power is required. Although I aimed for doing reflection and refraction in my proposal presentation, I elected to focus on getting a displacement place in place using a semi-transparent water texture and noise to mimic the motion of real waves.

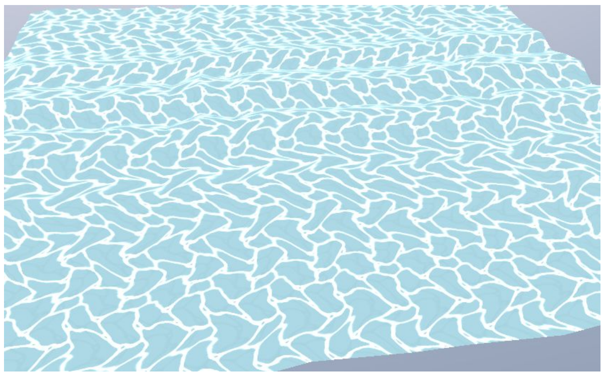


Figure 2: Water plane with displacement/wave motion.

The vertex shader for this effect begins with the same standard formula but has an important function called calculateSurface() which takes as input two floats, one for the x coordinate and one for the z. The y is left out because the purpose for the function is to calculate the y which in this case is the height of the waves. This involves a combination of sin functions with the basic format of:

```
y += (sin(x SCALE + uTime) + sin(x SCALE + uTime) + sin(x SCALE + uTime)) / 3.0
```

Before scale and time you can enter floats which allow for greater control of the way the surface reacts and at the end of the function this y value is returned. Lastly in the main() function there is the addition of a z position value which takes the calculated surface y value and multiplies it by a "strength" float (which controls how powerful the wave effect will be.

The fragment shader differences include a fragNoise() function which takes a vec2 position and runs a set noise calculations very similar to the above function involving sin. The purpose of this function is to generate the noise pattern which will decide how the "waves" (plane) vertices are displaced to create the effect. The main() function includes two texture variables, both using texture2D and incorporating the semi-transparent water texture for the map and the uv which will be used in gl_fragColor.

The code in the main HTML file is very similar to the clouds in setup, however the main difference is that you want to access the texture map from the uniforms and set wrapS and wrapT equal to THREE.RepeatWrapping. This tells the renderer that we want our water texture tile to wrap around the entire plane object and repeat itself, with no breaks or seams, ensure smooth transitions between the repeated texture. The only other thing to do is have the time and strength uniforms update in the render loop to keep the waves flowing each frame and allowing for GUI adjustment of the strength of the displacement effect.

6 CONCLUSIONS

Stuff--

A HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the appendix environment, the command section is used to indicate the start of each Appendix, with alphabetic order designation (i.e., the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure within an Appendix, start with subsection as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 CREATIVE VISION AND TECHNICAL CHALLENGES

A.2.1 Project Vision

A.2.2 Challenges in Developing the Shaders

A.3 Scene Geometry ~ Jonathan Chuang

A.3.1 Parsing OpenStreetMap

A.3.2 Constructive Geometery

A.3.3 Texture Mapping

A.3.4 Lighting Approximation

A.4 Lighting and Day/Night Cycle ~ Eisaku Imura

A.4.1 Creating the Skydome

A.4.2 Sun and Moon

A.4.3 Vertex and Fragment Shader

A.4.4 Sun and Moon Movement

A.5 Clouds and Water ~ Matt Loyola

A.5.1 Developing a Cloud System

A.5.2 Wave Motion in a Water Shader

A.6 Conclusions

A.7 References

ACKNOWLEDGMENTS

Professor Angus Forbes and Assistant Lucas Ferreira for their excellent lectures and strong willingness to help things make sense.

7 REFERENCES

[1 http://mrdoob.com/#/131/clouds

[2 https://github.com/mattatz/THREE.Cloud

[3 Week 9 Water Shader from Lab

[4 https://www.shadertoy.com/view/XslGRr

[5 http://gpupro.blogspot.com/2015/01/gpu-pro-6-real-time-rendering-of.html

[6 http://cgg.mff.cuni.cz/projects/SkylightModelling/HosekWilkie_SkylightModel_SIGGRAPH2012_Preprint_lowres.pdf

[7 http://www.cs.utah.edu/~shirley/papers/sunsky/sunsky.pdf

[8 http://hyperphysics.phy-astr.gsu.edu/hbase/atmos/blusky.html

[9 https://www.cosy.sbg.ac.at/~held/projects/triang/triang.html

[10 https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf