Elliott Spelman
8.12.16

# CS148 Final Project Report

See full project: www.spelman7.github.io

## Abstract/Project Summary

The motivation for this project was to take some skills newly learned in CS148 into the medium of virtual reality. To do that, I decided to use three.js, a Javascript wrapper for WebGL that includes a number of libraries which make VR an easier prospect, especially for a one-person project. Three.js abstracts a lot of features of OpenGL, and I wanted my project to be technically rigorous, so I chose to create a graphically challenging scene. Specifically, I worked to create a dimly lit cave, rendered in real-time from the perspective of someone inside the cave with a headlamp on. The sections below detail how I worked to realize that goal.

## Setting up VR System

### Tying libraries together

I had used three.js before, but never for a virtual reality project. There is an emerging web standard for VR (it's called WebVR, and it's currently a working group within W3C) but it's not fully realized, so tying the requisite pieces of the puzzle still involves some guesswork. Luckily, as a one-man team, most of those pieces have already been built/designed as third-party libraries for three.js. The trick is to tie them all together in a functional way. Here's a list of the key libraries used on this project:

**WebVR.js** *(author: mrdoob / http://mrdoob.com)* - detects if device is VR-compatible.

**VREffect.js** *(authors: dmarcos / https://github.com/dmarcos, mrdoob / http://mrdoob.com)* - scales VR effect to handle FOV of differently supported devices.

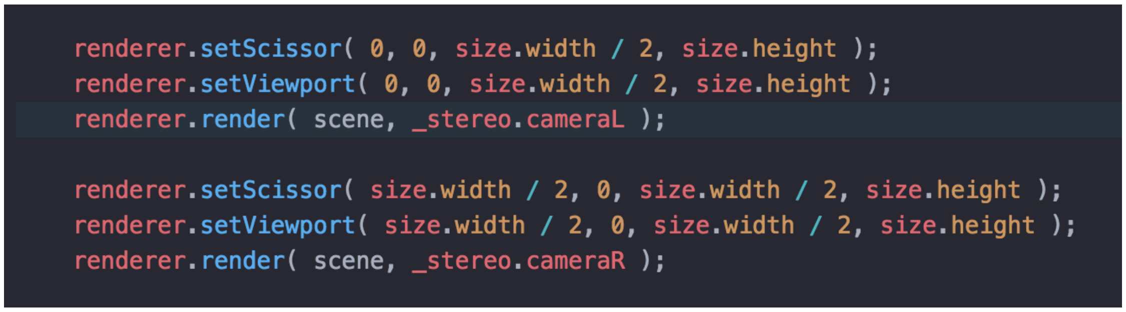
**StereoEffect.js** *(authors: alteredq / http://alteredqualia.com/, mrdoob / http://mrdoob.com/, arodic / http://aleksandarrodic.com/, fonserbc / http://fonserbc.github.io/)* - renders scene twice depending on independent perspective of each eye & pupillary distance.

**VRControls.js** *(authors: dmarcos / https://github.com/dmarcos, mrdoob / http://mrdoob.com)* - communication layer between device and VR libraries.

**DeviceOrientationControls.js** *(authors: richt / http://richt.me, WestLangley / http://github.com/WestLangley)* - computes device orientation dependent on internal accelerometer data and quaternion calculations.
To make the file structure simpler, I put these libraries (along with many other unused three.js libraries into a subfolder within the larger project folder. I included them within my proprietary code – written inside a single script as part of the index.html file.

## Orthographic projection

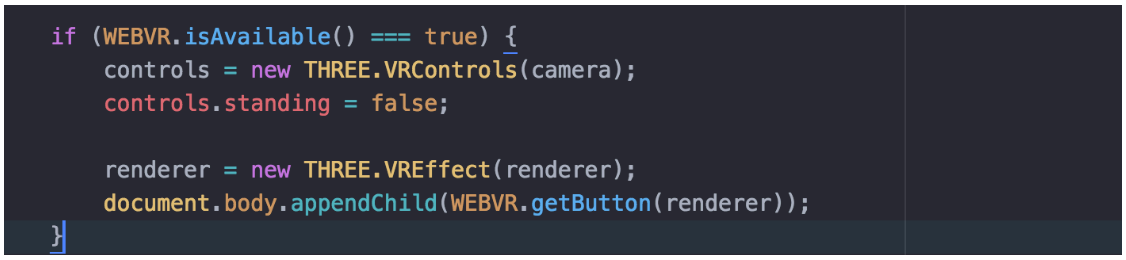
As mentioned in the description of libraries above, the actual orthographic projection work is mostly done within the StereoEffect.js library, where the code snippet below is located. As you can see, the script actually renders twice – once for each eye:



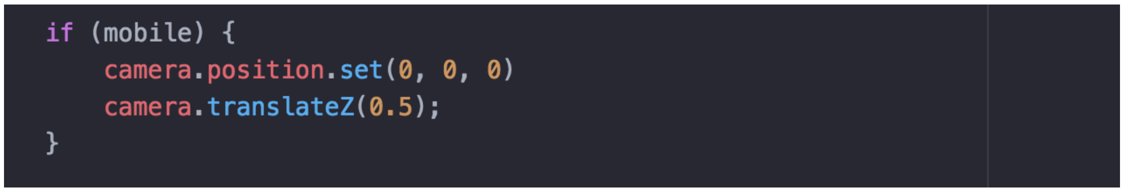
```
renderer.setScissor( 0, 0, size.width / 2, size.height );
renderer.setViewport( 0, 0, size.width / 2, size.height );
renderer.render( scene, _stereo.cameraL );

renderer.setScissor( size.width / 2, 0, size.width / 2, size.height );
renderer.setViewport( size.width / 2, 0, size.width / 2, size.height );
renderer.render( scene, _stereo.cameraR );
```

## Recognizing the device

The libraries handle most of the actual device recognition, but I did have to include a couple of event handlers that changed certain parameters of my scene in case the html detected a VR-capable device:



```
if (WEBVR.isAvailable() === true) {
    controls = new THREE.VRControls(camera);
    controls.standing = false;

    renderer = new THREE.VREffect(renderer);
    document.body.appendChild(WEBVR.getButton(renderer));
}
```



```
if (mobile) {
    camera.position.set(0, 0, 0)
    camera.translateZ(0.5);
}
```

## First Milestone

Here is a list of my stated goals for the first milestone along with short descriptions of if I achieved them and my method for doing so:

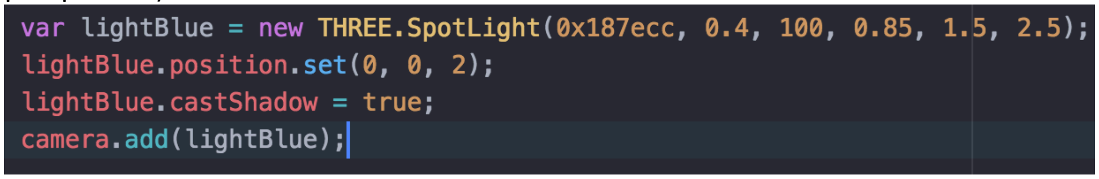
1. demo three.js/VR project hosted on Heroku

   *achieved* – although the project was hosted on Github, not Heroku
2. pull data from phone's accelerometer

   *achieved* – utilizing DeviceOrientationControls.js library
3. change camera perspective to match phone's perspective

   *achieved* – utilizing DeviceOrientationControls.js library
4. create spotlight and move to match phone's movement

   *achieved* – attached spotlight to camera perspective (as opposed to scene perspective)

```
var lightBlue = new THREE.SpotLight(0x187ecc, 0.4, 100, 0.85, 1.5, 2.5);
lightBlue.position.set(0, 0, 2);
lightBlue.castShadow = true;
camera.add(lightBlue);
```

5. rough scene generation

   *achieved* – utilized simple Isocahedron geometries and meshes

Here is a still photo of me playing with the first milestone prototype:

You can see full videos of the working first milestone prototype:

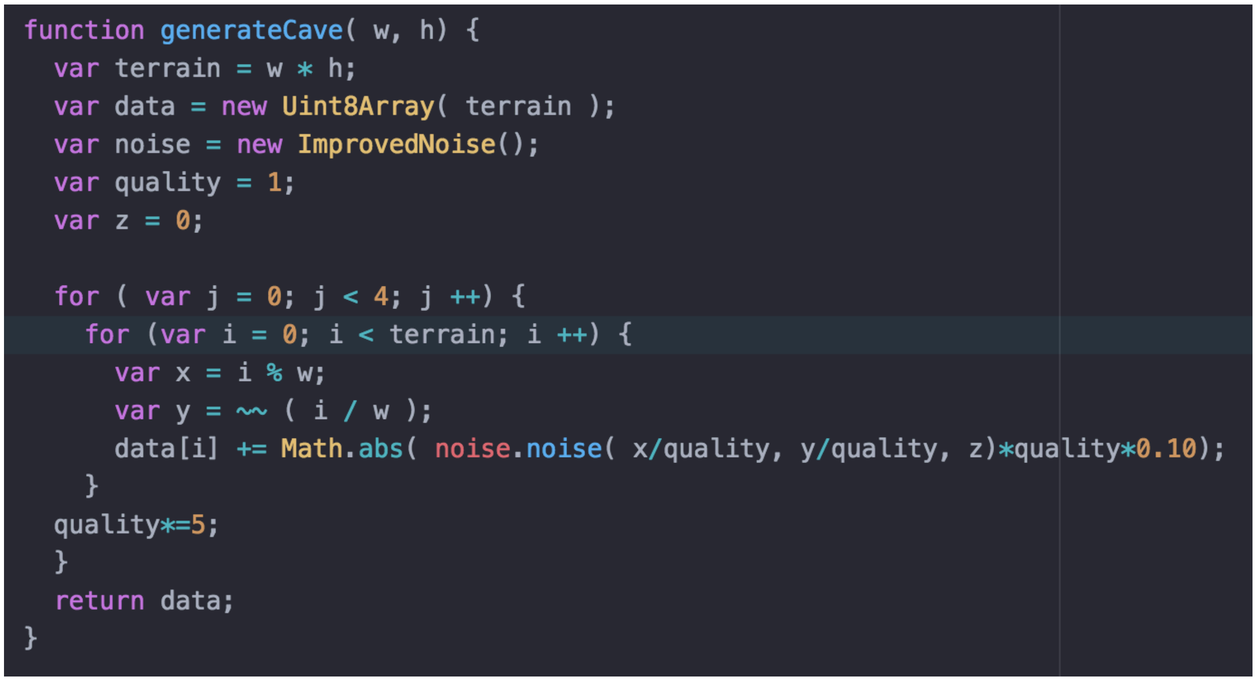
https://streamable.com/wm13
https://streamable.com/7gjc
https://streamable.com/qoug

## Adding Stalagmites/Terrain

Once the first milestone goals were met, most of the infrastructure for an easily distributed VR-based project was built. The challenge over the final week was rendering a scene that genuinely looked and felt like the viewer was inside of a cave. For that, I needed a ‘cavier’ type of terrain – randomized cavern walls and stalagmites/stalactites.

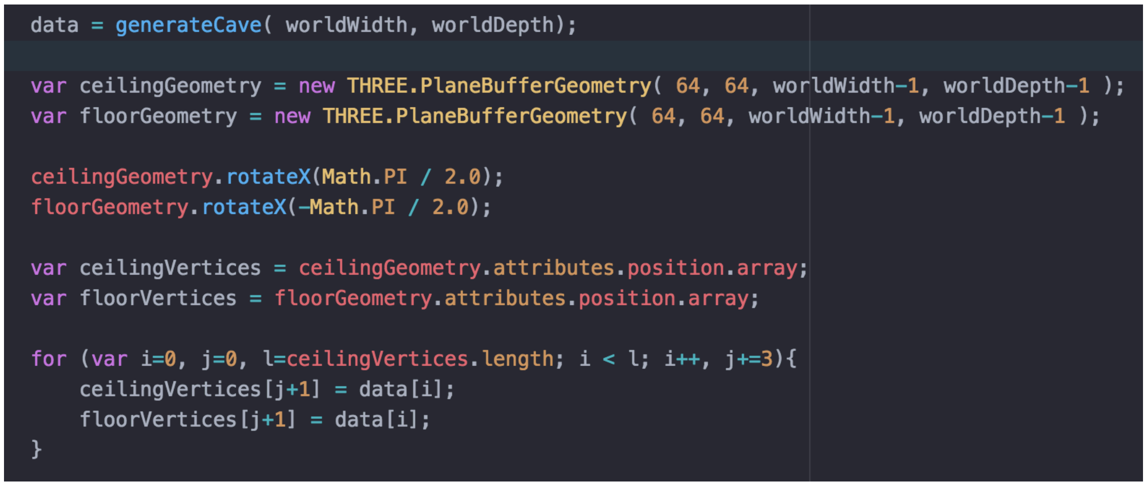
### Cavern Floor/Ceiling

The first step here was to generate a randomized array of terrain height data given the width and depth of the terrain in question. At first I tried randomly moving each terrain vertex an amount up or down, but that proved too noisy and jagged visually. So I tried a new approach that iterated through the z value of each terrain vertex and assigned it a randomly *incremental* change from the previous vertex. This random change was based on a borrowed Perlin noise library. Actual height generation code below:



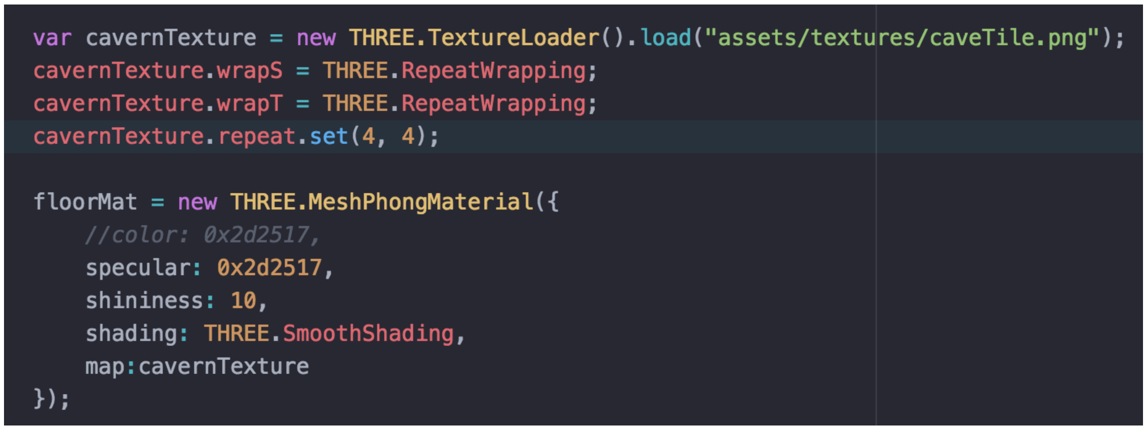
```
function generateCave( w, h) {
  var terrain = w * h;
  var data = new Uint8Array( terrain );
  var noise = new ImprovedNoise();
  var quality = 1;
  var z = 0;

  for ( var j = 0; j < 4; j ++) {
    for (var i = 0; i < terrain; i ++) {
      var x = i % w;
      var y = ~~ ( i / w );
      data[i] += Math.abs( noise.noise( x/quality, y/quality, z)*quality*0.10);
    }
  quality*=5;
  }
  return data;
}
```

The next step was to call the generateCave function, and create plane geometries into which I could pass the generated height data. This actually had to be done twice because there are two terrain surfaces – a floor and a ceiling. I’m using the same height data for each, so they really mirror one another. There was also an intermediate step of pulling out the actual vertex data inside of the PlaneBufferGeometry object, and iterating through it sequentially:



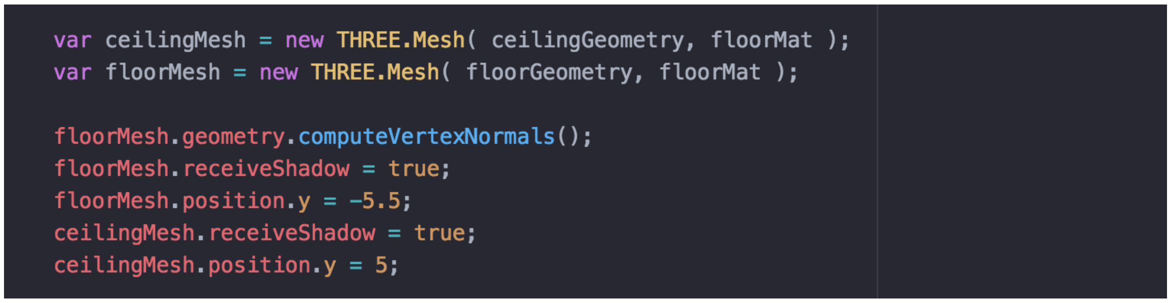
```
data = generateCave( worldWidth, worldDepth);

var ceilingGeometry = new THREE.PlaneBufferGeometry( 64, 64, worldWidth-1, worldDepth-1 );
var floorGeometry = new THREE.PlaneBufferGeometry( 64, 64, worldWidth-1, worldDepth-1 );

ceilingGeometry.rotateX(Math.PI / 2.0);
floorGeometry.rotateX(-Math.PI / 2.0);

var ceilingVertices = ceilingGeometry.attributes.position.array;
var floorVertices = floorGeometry.attributes.position.array;

for (var i=0, j=0, l=ceilingVertices.length; i < l; i++, j+=3){
    ceilingVertices[j+1] = data[i];
    floorVertices[j+1] = data[i];
}
```

Before creating an actual Mesh object into which I could feed the geometry data, I had to design a texture for the mesh-to-be. This involved designing a cavern-like .png image and loading it as a texture into a variable, which could then be mapped onto a MeshMaterial class:



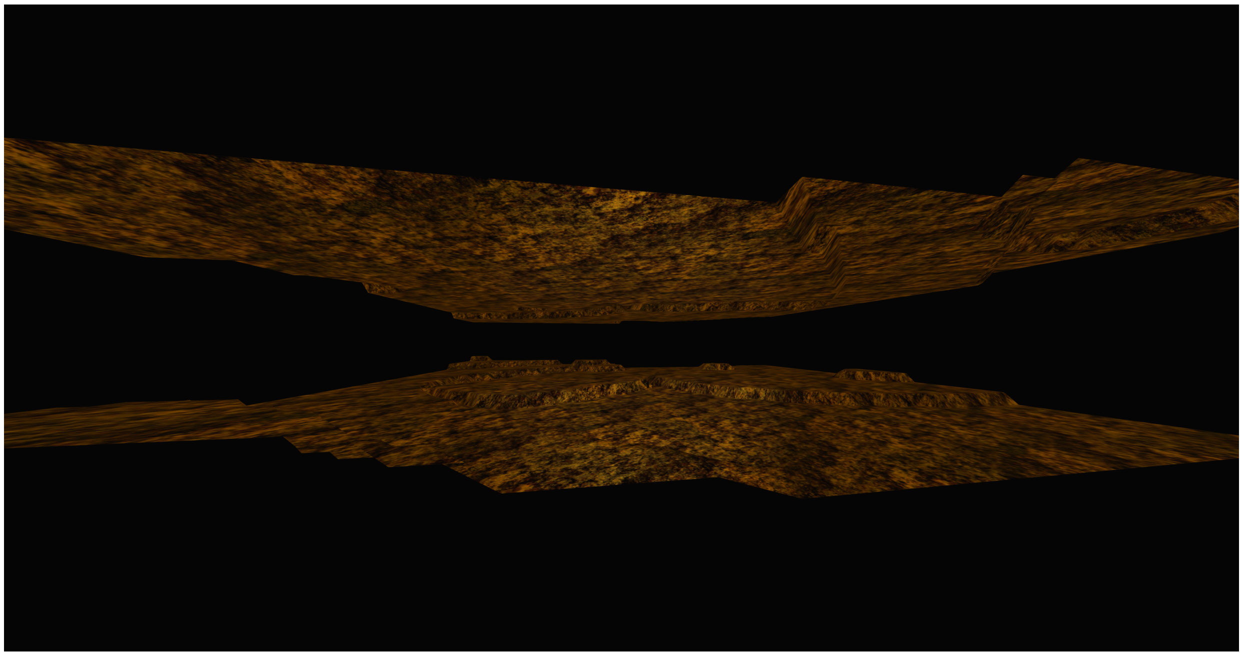
```
var cavernTexture = new THREE.TextureLoader().load("assets/textures/caveTile.png");
cavernTexture.wrapS = THREE.RepeatWrapping;
cavernTexture.wrapT = THREE.RepeatWrapping;
cavernTexture.repeat.set(4, 4);

floorMat = new THREE.MeshPhongMaterial({
    //color: 0x2d2517,
    specular: 0x2d2517,
    shininess: 10,
    shading: THREE.SmoothShading,
    map:cavernTexture
});
```

The final steps were generating a Mesh object from both the geometry and the new MeshMaterial class, translating these new Meshes so they accurately reflected a floor and a ceiling, and then also re-computing the normals of each vertex in the mesh so the lighting would affect them appropriately:



```
var ceilingMesh = new THREE.Mesh( ceilingGeometry, floorMat );
var floorMesh = new THREE.Mesh( floorGeometry, floorMat );

floorMesh.geometry.computeVertexNormals();
floorMesh.receiveShadow = true;
floorMesh.position.y = -5.5;
ceilingMesh.receiveShadow = true;
ceilingMesh.position.y = 5;
```

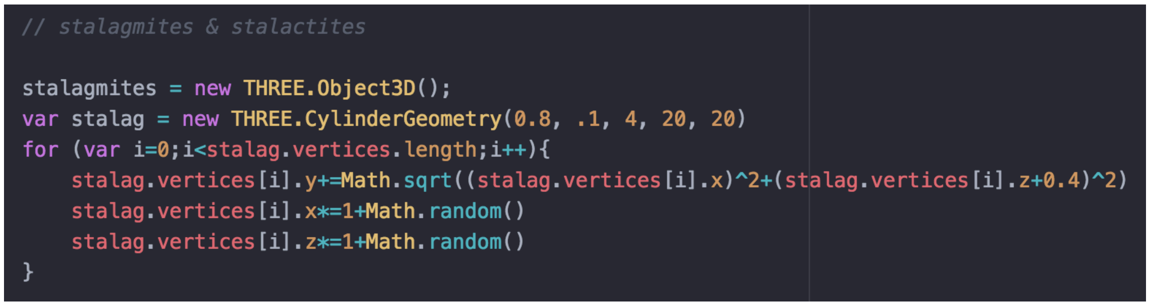
Here's a brightened screenshot of what the ceiling and floor meshes look like when zoomed out of the scene:



## Stalagmites/Stalactites

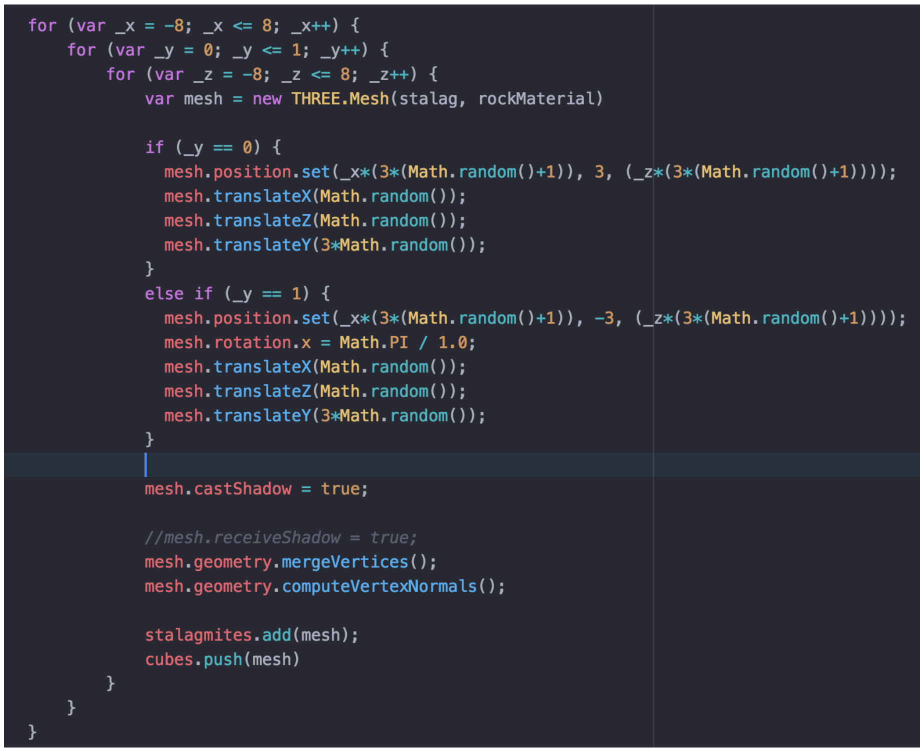
To generate the stalagmites and stalactites, I had to decide what type of pre-built geometry to use. Stalactites are very cone-like, so that's where I started. But unfortunately, cones always end in points, and stalactites have more of a spherical endpoint, generated by lots of liquid buildup/drip, and somewhat reminiscent of shapes that form with regular surface tension (droplets, especially). So I decided to use a modified cylinder geometry instead. Luckily with three.js, you can specify a cylinder geometry and then pass through both radiusTop and radiusBottom arguments. By making the top radius much larger than the bottom, I was able to generate a cone-like shape, but still have enough vertices on the bottom face to avoid the cone-like-tip shape.

Stalagmites are also very bulbous and random looking, so it wasn't enough to keep the smooth cylinder shape. I decided to push each x and z vertex value outward by a randomized amount and then (eventually) merge/smooth vertices that happened to overlap. This still left me with a flat, mesa-like cylinder top, however. To round that out, I decided to translate the y value of each vertex by an amount proportional to that vertex's distance away from the radially symmetric axis of the cylinder.



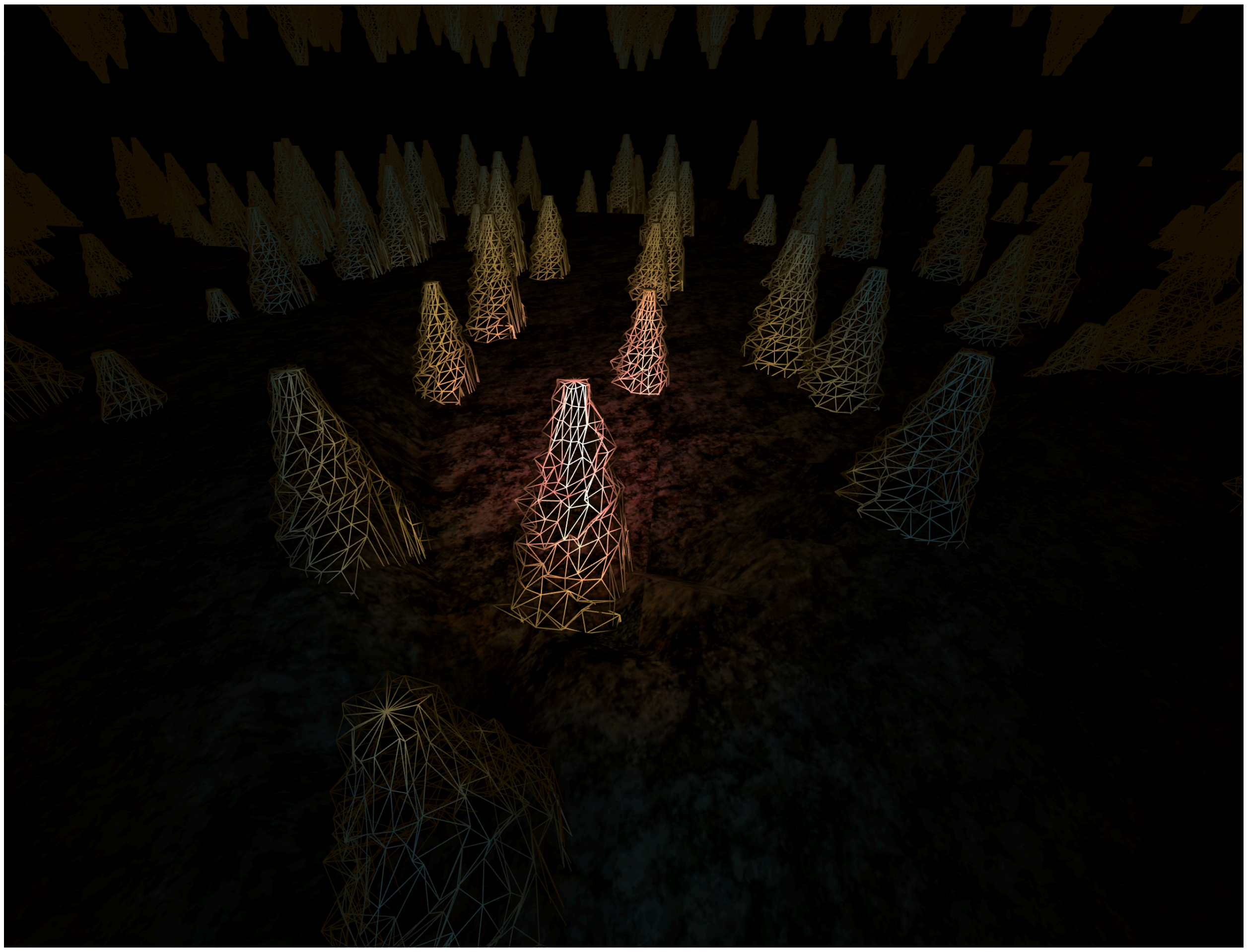
```
// stalagmites & stalactites

stalagmites = new THREE.Object3D();
var stalag = new THREE.CylinderGeometry(0.8, .1, 4, 20, 20)
for (var i=0;i<stalag.vertices.length;i++){
    stalag.vertices[i].y+=Math.sqrt((stalag.vertices[i].x)^2+(stalag.vertices[i].z+0.4)^2)
    stalag.vertices[i].x*=1+Math.random()
    stalag.vertices[i].z*=1+Math.random()
}
```

Once the stalagmite geometry had been generated, I had to position them throughout the scene in a way that felt natural. To do this, I iterated through a 'grid' of positions in the scene, 17 wide by 17 deep. For each position I created both a stalagmite and a stalactite by creating a new Mesh object based on the cylindrical geometry from before. Each Mesh was then translated by a random amount (particularly in the y-dimension, to create non-uniform heights), and rotated depending on whether it was attached to the cavern floor or ceiling. The final steps involve cleaning up the Mesh vertices and computing their normals for lighting purposes.



```
for (var _x = -8; _x <= 8; _x++) {
    for (var _y = 0; _y <= 1; _y++) {
        for (var _z = -8; _z <= 8; _z++) {
            var mesh = new THREE.Mesh(stalag, rockMaterial)

            if (_y == 0) {
              mesh.position.set(_x*(3*(Math.random()+1)), 3, (_z*(3*(Math.random()+1))));
              mesh.translateX(Math.random());
              mesh.translateZ(Math.random());
              mesh.translateY(3*Math.random());
            }
            else if (_y == 1) {
              mesh.position.set(_x*(3*(Math.random()+1)), -3, (_z*(3*(Math.random()+1))));
              mesh.rotation.x = Math.PI / 1.0;
              mesh.translateX(Math.random());
              mesh.translateZ(Math.random());
              mesh.translateY(3*Math.random());
            }

            mesh.castShadow = true;

            //mesh.receiveShadow = true;
            mesh.geometry.mergeVertices();
            mesh.geometry.computeVertexNormals();

            stalagmites.add(mesh);
            cubes.push(mesh)
        }
    }
}
```

Here is a visual of what the stalagmites and stalactites look like without their texture being applied (as wireframes):

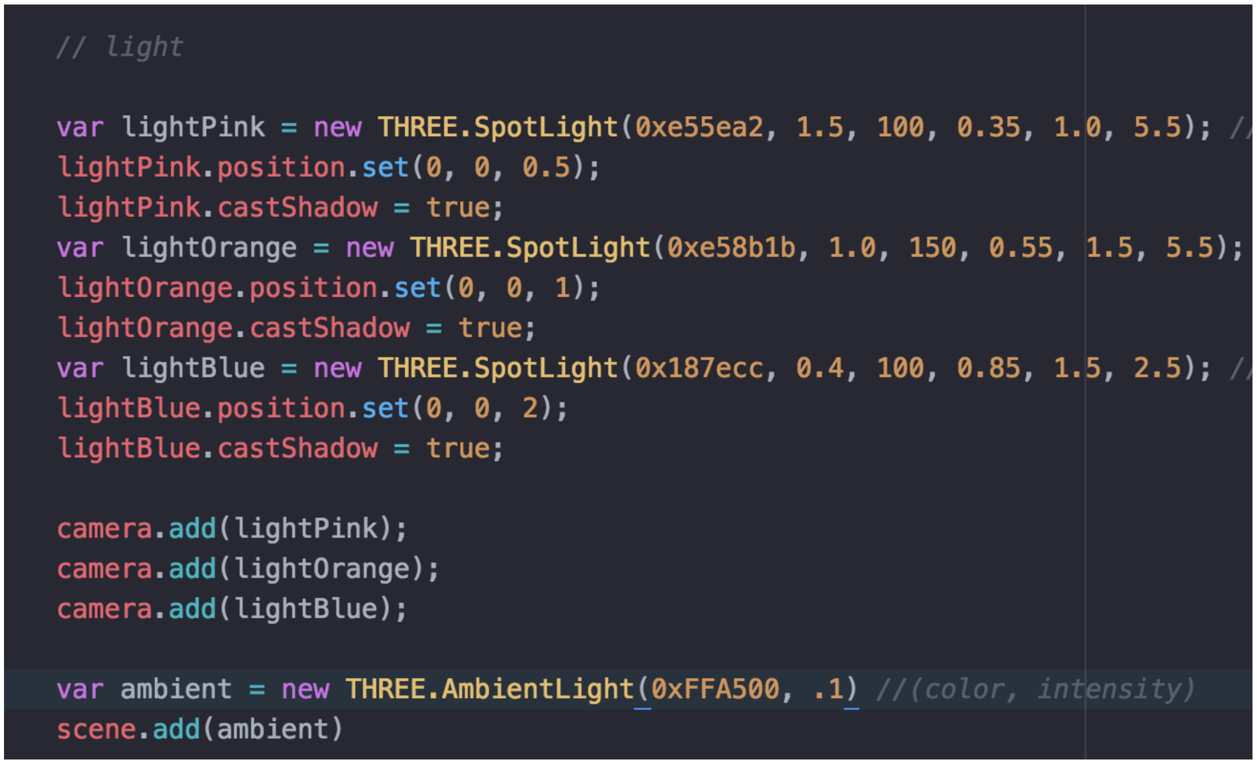


## Lighting

The final major step in making this a finished project was to design a headlamp-style lighting system. The main breakthrough in that effort occurred early, with the realization that you could attach a light to the camera directly, and not just the scene. This made the idea of a ‘headlamp’ very easy to put into practice.

Most of the other lighting work was really guess-and-check fine-tuning. The first problem was that any light centered at the origin (0,0,0 - or in the same place as the camera) will not create realistic shadows, because the eye and the light generation always overlap. The camera sees exactly what the light illuminates and the light illuminates exactly what the camera sees – there’s never any visible shadow because there’s *zero* offset between light and camera. To get around this, I experimented with changing the z-depth of the light source. I also created three spotlights with three different colors, because I liked the visual of a layered color look, and headlamps rarely diffuse super quickly. They kind of fade away into the darkness that surrounds their circle of illumination. My choices of pink, orange, and blue are slightly random, but I think those color are beautiful together, and they create an eerie effect within the scene.

I also had to add some ambient light (a very small amount) in order to increase the reality of the scene, because it feels unnatural to have something *totally* non-illuminated, even if you can't fully perceive the detail behind the illumination.



```
// light

var lightPink = new THREE.SpotLight(0xe55ea2, 1.5, 100, 0.35, 1.0, 5.5); /
lightPink.position.set(0, 0, 0.5);
lightPink.castShadow = true;
var lightOrange = new THREE.SpotLight(0xe58b1b, 1.0, 150, 0.55, 1.5, 5.5);
lightOrange.position.set(0, 0, 1);
lightOrange.castShadow = true;
var lightBlue = new THREE.SpotLight(0x187ecc, 0.4, 100, 0.85, 1.5, 2.5); /
lightBlue.position.set(0, 0, 2);
lightBlue.castShadow = true;

camera.add(lightPink);
camera.add(lightOrange);
camera.add(lightBlue);

var ambient = new THREE.AmbientLight(0xFFA500, .1) //(color, intensity)
scene.add(ambient)
```

## Next Steps

With this project, the most important thing, I think, was the creation of a platform that I can use for any type of fast-prototype VR project. It opens up the question of what type of VR project I'd like to create, because the underlying architecture is very flexible. I'm particularly interested in how to tie additional controllers into the scene. Perhaps a second phone? Or different users within the same scene? For that I would need avatars, or some kind of physicality to the user within the rendered scene.

## Credits (also see WebVR-related three.js libraries above)

three.js - https://github.com/mrdoob/three.js/

Marpi - https://github.com/marpi/worlds

ImprovedNoise.js - http://mrl.nyu.edu/~perlin/noise/