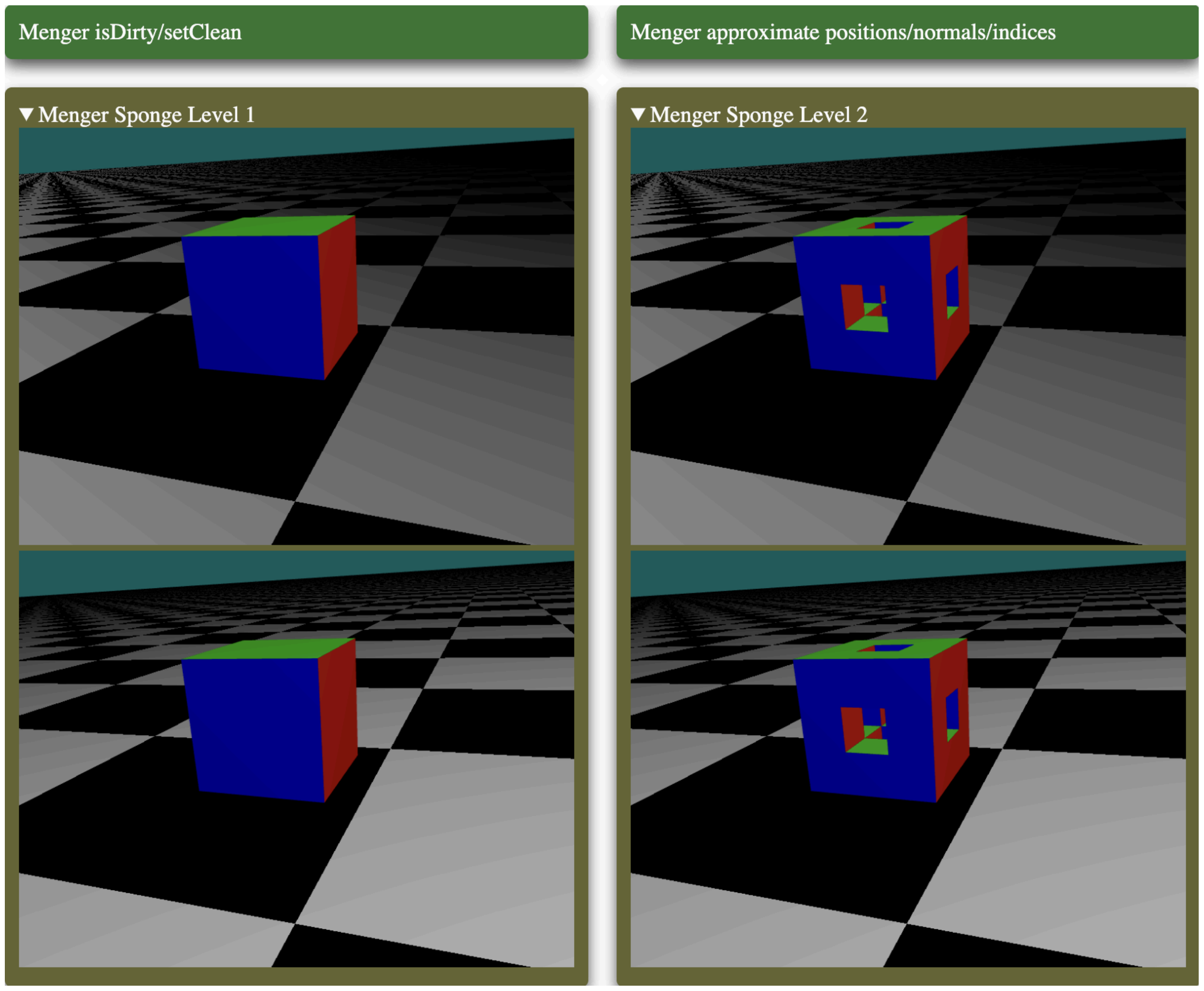
https://gitlab.com/raymww/menger-sponge

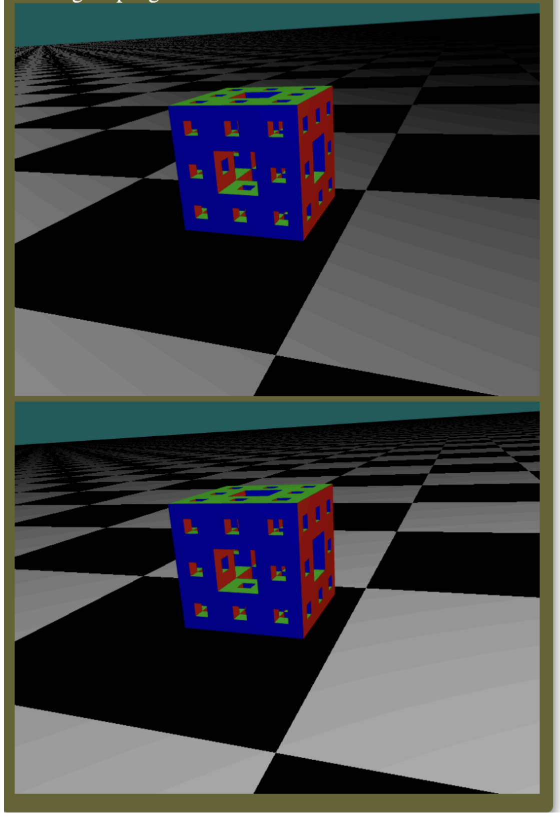
Writeup for A3-Menger Sponge:

We, Raymond Wang and Max Hartfield, have fully completed what was required for the Menger Sponge assignment, Menger Sponge Implementation, Camera Controls, and Sponge and Floor Shaders.

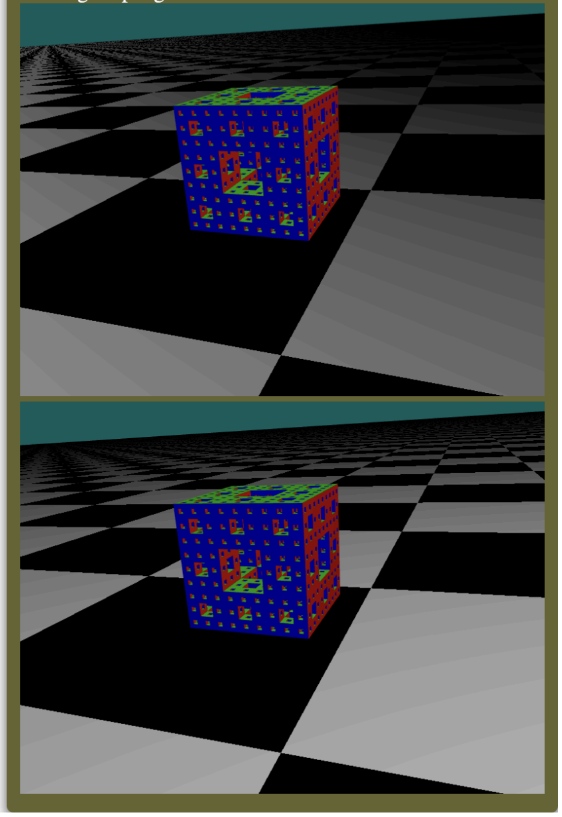
Here are some example images of our Menger Sponge compared to the reference's Menger Sponge in the test suite:



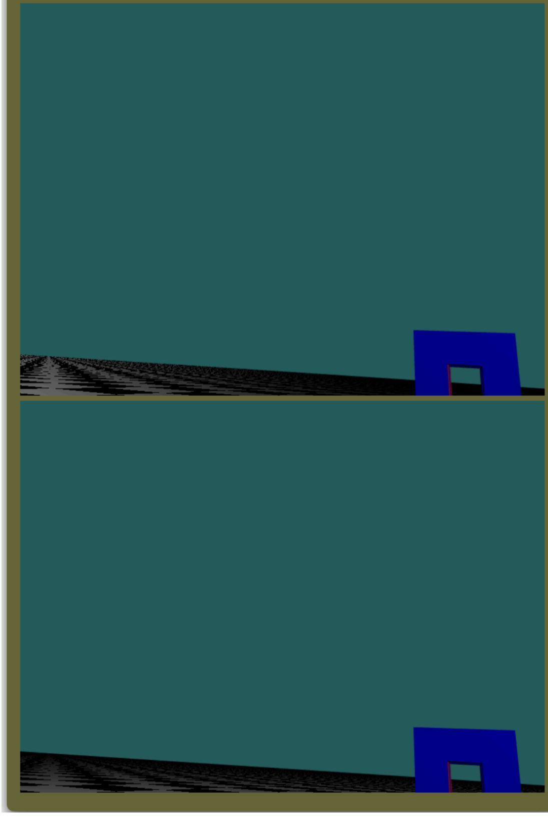
▼ Menger Sponge Level 3



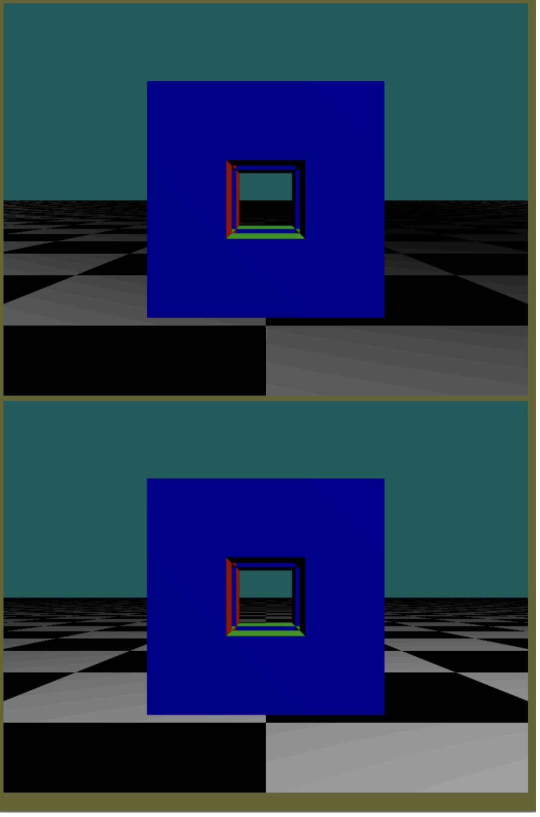
▼ Menger Sponge Level 4

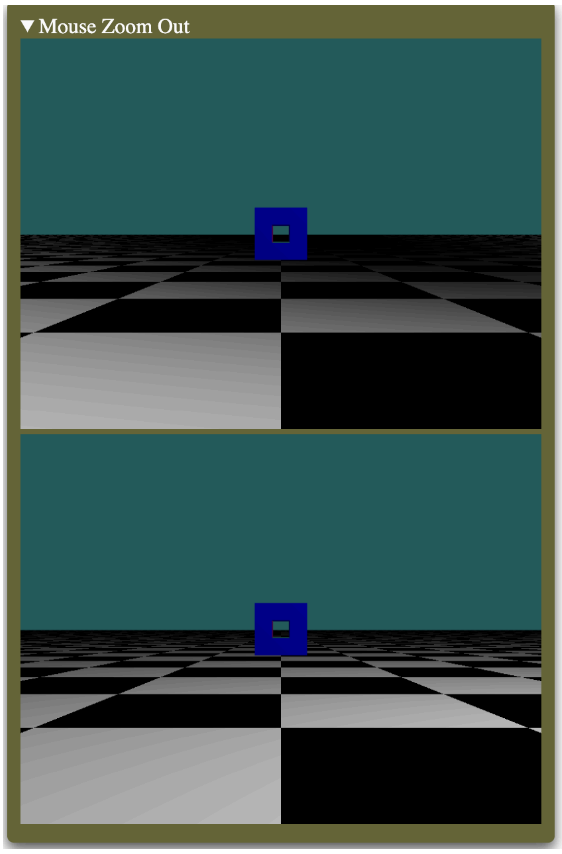


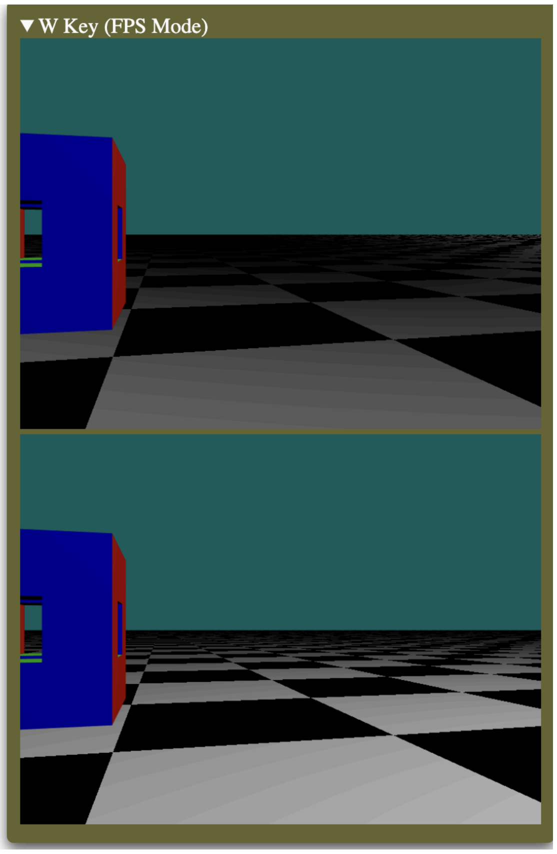
▼ Mouse FPS Rotation

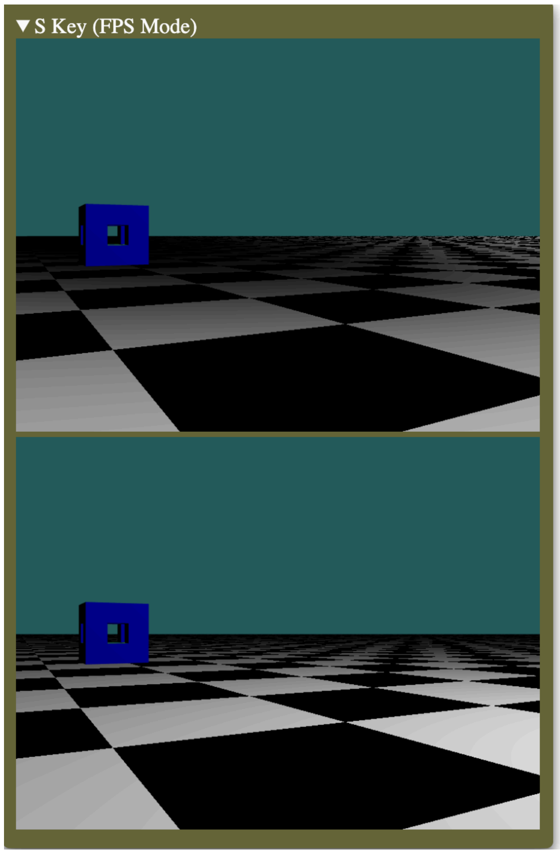


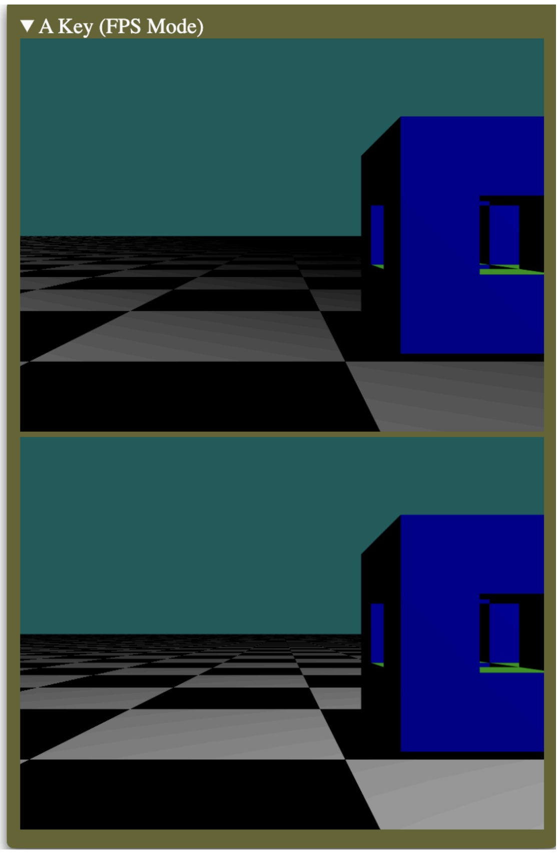
▼ Mouse Zoom In

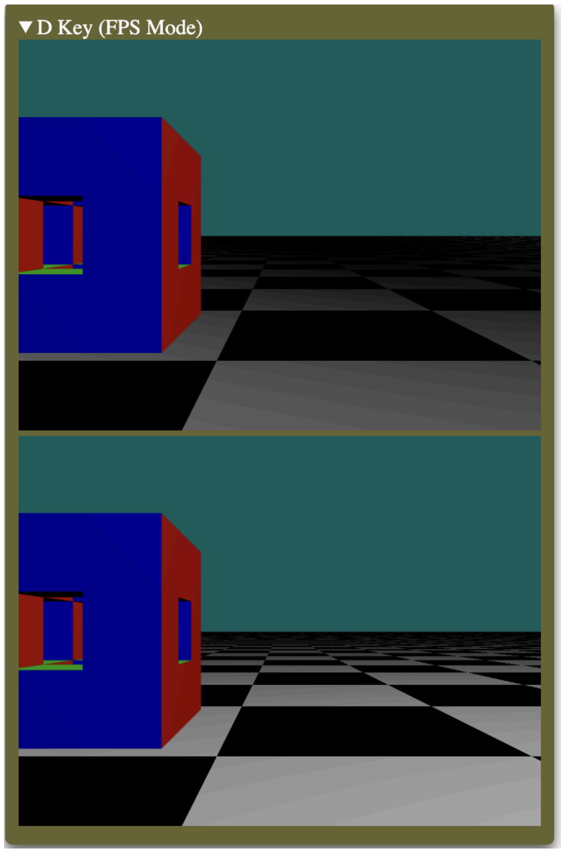


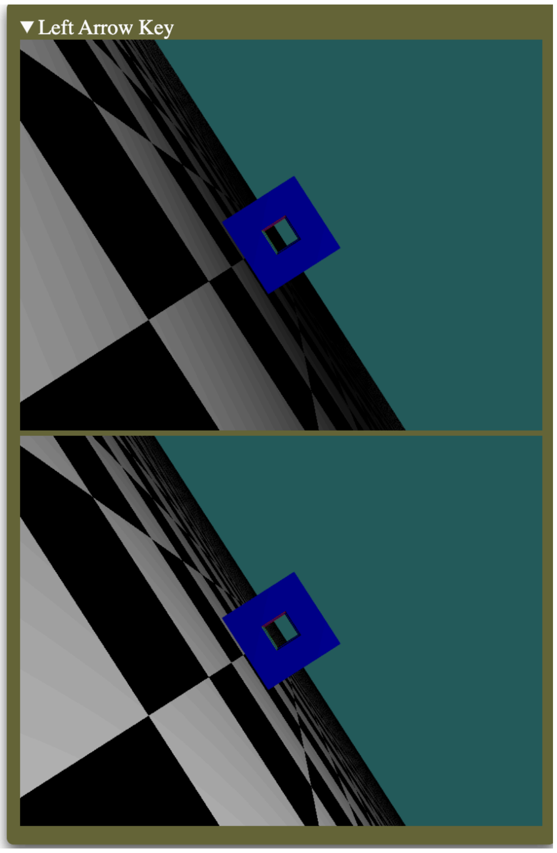

▼ Mouse Zoom Out

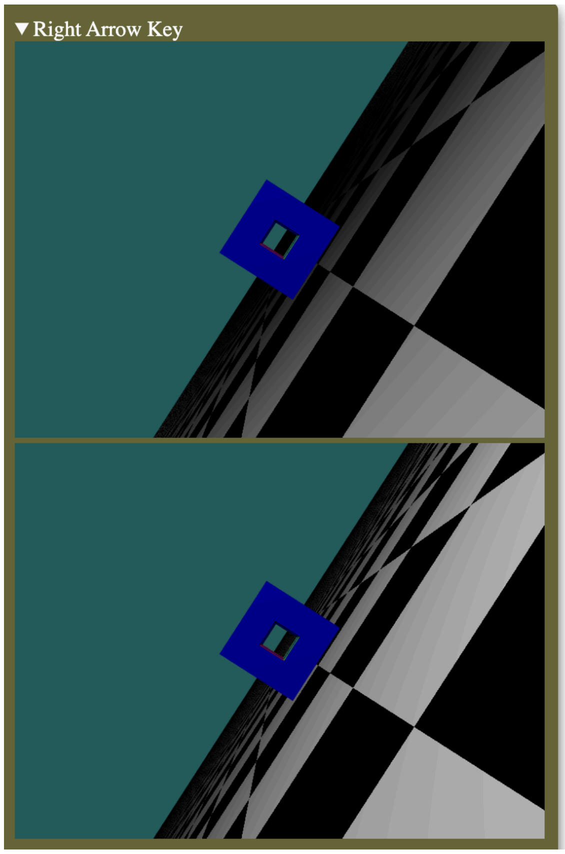

▼ W Key (FPS Mode)

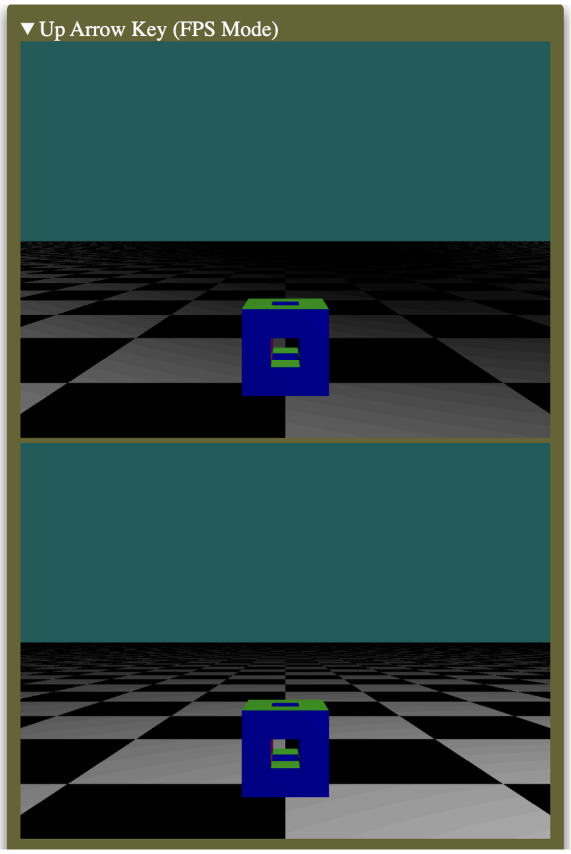

▼ S Key (FPS Mode)

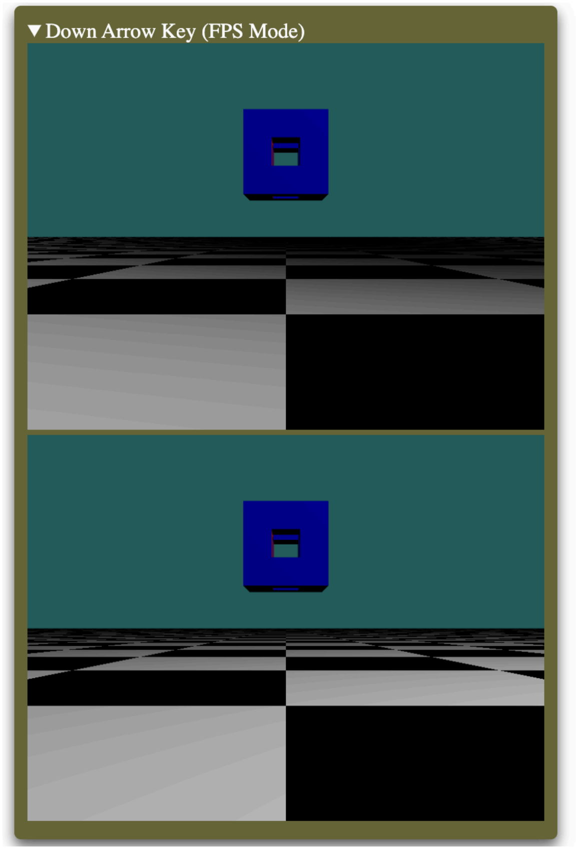

▼ A Key (FPS Mode)


▼ D Key (FPS Mode)


▼ Left Arrow Key


▼ Right Arrow Key


▼ Up Arrow Key (FPS Mode)



Menger Sponge Implementation:
Creating the Menger Sponge was fairly easy. We first figured out how to render a cube and then implemented a recursive algorithm for the levels. To render the cube, we first rendered a single face using two triangles. This consisted of six vertex positions, six normals facing in the direction the face is facing, and six consecutive indices. We drew the vertices out on paper to visualize and get the values for each vertex position on each face. We then trial and error'd permutations until the face actually showed up. We repeated this for the rest of the 5 faces. We also colored each face based on the abs of its normal and hard coded a different position for the camera so we can better see the different faces and ensure everything was working. For the recursive algorithm, we had the bounding area of the cube as the recursive state, and we subdivided this area into 27 subsections, 3 by 3 by 3. We then called the recursive algorithm for these new subdivisions, making sure to skip the ones in the middle. Once the level reached zero, we would stop recursing and add the individual cube we previously made based on each bounding area.

```
/**
* Represents a Menger Sponge
*/
export class MengerSponge implements IMengerSponge {

 // TODO: sponge data structures
 positions: number[];
 indices: number[];
 normals: number[];
 currIdx: number;
 first: boolean;
 dirty: boolean;

 constructor(level: number) {
   this.positions = [];
   this.indices = [];
   this.normals = [];
   this.currIdx = 0;
   this.first = true;
   this.setLevel(level);
   // TODO: other initialization
 }

 /**
  * Returns true if the sponge has changed.
  */
 public isDirty(): boolean {
      return this.dirty;
 }

 public setClean(): void {
   this.dirty = false;
 }
  public helper(level: number = 0, xm: number = -0.5, ym: number = -0.5, zm: number =
-0.5,
   length: number = 1) {
     // console.log("level*********" + level + " positions: " + this.positions.length
+ " indices: " + this.indices.length + " normals: " + this.normals.length);

   // TODO: initialize the cube
   if(level == 0) {
     let xM: number = xm + length;
     let yM: number = ym + length;
     let zM: number = zm + length;
     this.positions.push(
         //facing +z
         xm, ym, zM, 1.0,
         xM, ym, zM, 1.0,
         xM, yM, zM, 1.0,
         //facing +z
         xm, ym, zM, 1.0,
         xM, yM, zM, 1.0,
         xm, yM, zM, 1.0,
         //facing -z
         xM, ym, zm, 1.0,
         xm, ym, zm, 1.0,
         xM, yM, zm, 1.0,
         //facing -z
         xM, yM, zm, 1.0,
         xm, ym, zm, 1.0,
         xm, yM, zm, 1.0,
         //facing +y
         xm, yM, zM, 1.0,
         xM, yM, zM, 1.0,
         xM, yM, zm, 1.0,
         //facing +y
         xm, yM, zm, 1.0,
         xm, yM, zM, 1.0,
         xM, yM, zm, 1.0,
         //facing -y
         xM, ym, zM, 1.0,
         xm, ym, zM, 1.0,
         xM, ym, zm, 1.0,
         //facing -y
         xm, ym, zM, 1.0,
         xm, ym, zm, 1.0,
         xM, ym, zm, 1.0,
         //facing +x
         xM, ym, zM, 1.0,
         xM, ym, zm, 1.0,
         xM, yM, zM, 1.0,
         //facing +x
         xM, yM, zM, 1.0,
         xM, ym, zm, 1.0,
         xM, yM, zm, 1.0,
         //facing -x
         xm, ym, zm, 1.0,
         xm, ym, zM, 1.0,
         xm, yM, zM, 1.0,
         //facing -x
         xm, ym, zm, 1.0,
         xm, yM, zM, 1.0,
         xm, yM, zm, 1.0
     );
     for (let i = this.currIdx; i <= this.currIdx + 35; i++) {
       this.indices.push(i);
     }
     this.currIdx += 36
     for(var i: number = 0; i < 6; i++) {
       this.normals.push(0.0, 0.0, 1.0, 0.0)
     }
     for(var i: number = 0; i < 6; i++) {
       this.normals.push(0.0, 0.0, -1.0, 0.0)
     }
     for(var i: number = 0; i < 6; i++) {
       this.normals.push(0.0, 1.0, 0.0, 0.0)
     }
     for(var i: number = 0; i < 6; i++) {
       this.normals.push(0.0, -1.0, 0.0, 0.0)
     }
     for(var i: number = 0; i < 6; i++) {
       this.normals.push(1.0, 0.0, 0.0, 0.0)
     }
     for(var i: number = 0; i < 6; i++) {
       this.normals.push(-1.0, 0.0, 0.0, 0.0)
     }
   } else {
     let cnt: number = 0;
     for(var i: number = 0; i < 3; i++) {
       for(var j: number = 0; j < 3; j++) {
         for(var k: number = 0; k < 3; k++) {
           if (i % 2 + j % 2 + k % 2 < 2) {
             let newLength: number = length / 3.0
             cnt += 1;
             this.helper(level - 1, xm + i * newLength, ym + j * newLength, zm + k *
 newLength,
                 newLength);
           }
         }
       }
     }
     console.log("at level " + level + ", drew " + cnt + " sub-cubes for next level");
   }
   }

 public setLevel(level: number)
 {
   this.positions = [];
   this.indices = [];
   this.normals = [];
   this.currIdx = 0;
   this.helper(level);
   if(!this.first){
     this.first = false;
   } else {
     this.dirty = true;
   }
 }

 /* Returns a flat Float32Array of the sponge's vertex positions */
 public positionsFlat(): Float32Array {
   // TODO: right now this makes a single triangle. Make the cube fractal instead.
   return new Float32Array(this.positions);
 }

 /**
  * Returns a flat Uint32Array of the sponge's face indices
  */
 public indicesFlat(): Uint32Array {
   // TODO: right now this makes a single triangle. Make the cube fractal instead.
   return new Uint32Array(this.indices);
 }

 /**
  * Returns a flat Float32Array of the sponge's normals
  */
 public normalsFlat(): Float32Array {
   // TODO: right now this makes a single triangle. Make the cube fractal instead.
   return new Float32Array(this.normals);
 }
```

Camera Controls:

Creating the camera controls was also relatively easy, as the actual moving of the camera and most of the keyboard functions just required the use of a predefined function in the camera class. The challenging part was just figuring out which function to call in which situation, and finding where to insert the code and the camera's built in functions themselves. We implemented WASD by calling the offset function, which moves the camera and target at the same time, which results in translation of the camera. We passed in forward or right vectors to move left / right or forward / backward.

Then, we implemented the arrow keys in a similar fashion, by calling either roll for the left and right rolling of the camera or offset again for the up and down translation of the camera, passing in the correct vector and predefined speed constant.

Finally, we implemented the mouse drag rotation and zoom. This was done by creating the two events by checking mouse.buttons for left (1) or right (2) click events. Then, for the rotation, this was first done by tracking how much the mouse had changed. Then, we calculated the corresponding vector for this screen change by multiplying the up and right vectors by the amount that had changed and adding it together to get the vector version of the mouse movement. Finally, we crossed this vector with the forward vector to get the axis of rotation to rotate around. In order to implement the mouse zoom, it was done in a similar way, except we only tracked mouse y movement and zoomed in by the amount of movement. Additionally, we fixed the issue of zooming in too far inverting the world by placing a bound on the distance of the camera's target to not be able to zoom in into the camera.

Left and right click drag:

```
public drag(mouse: MouseEvent): void {
   // TODO: Your code here for left and right mouse drag
   if(mouse.buttons == 1){
     let changex: number = mouse.screenX - this.prevX;
     let changey: number = mouse.screenY - this.prevY;
     let xmul: Vec3 = new Vec3([-1 * changex, -1 * changex, -1 * changex]);
     let ymul: Vec3 = new Vec3([changey, changey, changey])
     let axis: Vec3 = Vec3.cross(this.camera.forward(),
this.camera.up().multiply(ymul).add(this.camera.right().multiply(xmul)).normalize());
     this.camera.rotate(axis.normalize(), GUI.rotationSpeed);
   } else if(mouse.buttons == 2){
     let changey: number = mouse.screenY - this.prevY;
     this.prevY = mouse.screenY;
     if(changey > 0 || this.camera.distance() > 0){
       this.camera.offsetDist(changey * GUI.zoomSpeed);
     }
   }
 }
```

Fps mode / arrow keys:

```
switch (key.code) {
     case "KeyW": {
       this.camera.offset(this.camera.forward().negate(), GUI.zoomSpeed, true);
       break;
     }
     case "KeyA": {
       this.camera.offset(this.camera.right().negate() , GUI.zoomSpeed, true);
       break;
     }
     case "KeyS": {
       this.camera.offset(this.camera.forward(), GUI.zoomSpeed, true);
       break;
     }
     case "KeyD": {
       this.camera.offset(this.camera.right(), GUI.zoomSpeed, true);
       break;
     }
     case "KeyR": {
       this.setCamera(new Vec3([0, 0, -6]), new Vec3([0, 0, 0]), new Vec3([0, 1, 0]),
45, this.width / this.height, 0.1, 1000.0);
       break;
     }
     case "ArrowLeft": {
       this.camera.roll(GUI.rollSpeed, false);
       break;
     }
     case "ArrowRight": {
       this.camera.roll(GUI.rollSpeed, true)
       break;
     }
     case "ArrowUp": {
       this.camera.offset(this.camera.up(), GUI.zoomSpeed, true);
       break;
     }
     case "ArrowDown": {
       this.camera.offset(this.camera.up().negate(), GUI.panSpeed, true);
       break;
     }
     case "Digit1": {
       this.sponge.setLevel(0);
       break;
```

```
    }
    case "Digit2": {
      this.sponge.setLevel(1);
      break;
    }
    case "Digit3": {
      this.sponge.setLevel(2);
      break;
    }
    case "Digit4": {
      this.sponge.setLevel(3);
      break;
    }
    default: {
      console.log("Key : '", key.code, "' was pressed.");
      break;
    }
  }
```

~~~

Sponge and Floor Shaders:
To shade the sponge, we calculated the light intensity by taking the dot product of the normalized light direction and the normal. We then clamped the intensity value to make sure it was bounded between 0 and 1. Finally we multiplied the intensity by the color calculated by the absolute of the normals and applied it to gl_FragColor.

To render the floor, we copied the code for rendering the sponge in App.ts. We then thought of the four triangles for the floor. We already had some familiarity with this when creating the sponge so this was quick and easy to do. We copied the vertex shader from the cube and made the fragment shader simply set the color red to see if the floor showed up. We surprisingly got the floor to render first try. We were unsure about a lot of what we copied and pasted in App.ts so we were happy to see that it ended up working. To create the checkerboard pattern we passed the world position from the vertex shader to the fragment shader. We then calculated whether the position should be colored white or black based on the world positions x and z values. This was done by flooring the x and z positions / 5.0 (each square was 5 by 5) and doing mod 2. The intensity should have been the same calculation as the sponge shader but we encountered a bug so we had to do a fix by multiplying the dot product by -1.0 and multiplying the color by 1 - intensity.

```
// TODO: Write the fragment shader

export let defaultFSText = `
```
~~~

```
   precision mediump float;
   varying vec4 lightDir;
   varying vec4 normal;
   void main () {
       float intensity = dot(normalize(lightDir.xyz), normal.xyz);
       intensity = clamp(intensity, 0.0, 1.0);
       gl_FragColor = vec4(intensity * abs(normal.xyz), 1.0);
   }
`;

// TODO: floor shaders

export let floorVSText = `
   precision mediump float;

   attribute vec3 vertPosition;
   attribute vec3 vertColor;
   attribute vec4 aNorm;

   varying vec4 lightDir;
   varying vec4 normal;
   varying vec3 worldPosition;

   uniform vec4 lightPosition;
   uniform mat4 mWorld;
   uniform mat4 mView;
   uniform mat4 mProj;

   void main () {
       //  Convert vertex to camera coordinates and the NDC
       gl_Position = mProj * mView * mWorld * vec4 (vertPosition, 1.0);

       //  Compute light direction (world coordinates)
       lightDir = lightPosition - vec4(vertPosition, 1.0);

       worldPosition = vertPosition;

       //  Pass along the vertex normal (world coordinates)
       normal = aNorm;
   }
`;
```

```
export let floorFSText = `
   precision mediump float;
   varying vec4 lightDir;
   varying vec4 normal;
   varying vec3 worldPosition;
   void main () {
       vec3 color;
       if (mod(floor(worldPosition.x / 5.0) + floor(worldPosition.z / 5.0), 2.0) == 0.0) {
           color = vec3(0.0, 0.0, 0.0);
       } else {
           color = vec3(1.0, 1.0, 1.0);
       }
       float intensity = -dot(normalize(lightDir.xyz), normal.xyz);
       intensity = clamp(intensity, 0.0, 1.0);
       gl_FragColor = vec4((1. - intensity) * color, 1.0);
   }
`;
```

```
/* Returns a flat Float32Array of the floor's vertex positions */
public floorPositionsFlat(): Float32Array {
  // TODO: right now this makes a single triangle. Make the cube fractal instead.
  return new Float32Array([
    -1000.0, -2.0, -1000.0, 1.0,
    -1000.0, -2.0, 1000.0, 1.0,
    1000.0, -2.0, -1000.0, 1.0,

    -1000.0, -2.0, 1000.0, 1.0,
    1000.0, -2.0, 1000.0, 1.0,
    1000.0, -2.0, -1000.0, 1.0,
  ]);
}

/**
 * Returns a flat Uint32Array of the floor's face indices
 */
public floorIndicesFlat(): Uint32Array {
  // TODO: right now this makes a single triangle. Make the cube fractal instead.
  return new Uint32Array([
    0, 1, 2, 3, 4, 5
  ]);
}
```

```
/**
 * Returns a flat Float32Array of the floor's normals
 */
public floorNormalsFlat(): Float32Array {
  // TODO: right now this makes a single triangle. Make the cube fractal instead.
  return new Float32Array([
    0.0, 1.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0
  ]);
}
```

Issues we encountered and fixed:
Menger Sponge Implementation: We had issues rendering some triangles. This was fixed by trial and error of the permutations of the vertex positions. We also had some issues with the recursive calls due to how the index values were being passed through to the recursive steps. This was fixed by making the index values global.

Camera Controls: The main problems that we had with camera controls were with the mouse drag rotation and zoom. One of the biggest challenges was figuring out how to get it to go only on the click, which was solved through some online research found to be mouse.buttons == 1 / 2. Additionally, it was very confusing to get the rotation right, due to the tracking the mouse and making a direction vector out of that and then taking the correct cross product, but after diagramming it out, we managed to get a working implementation, that however was rotating the wrong way on along the y axis but correctly on the x. After playing around with the ordering and negating the x change vector, we were able to fix this.

Sponge and Floor Shaders: At first we thought our shading implementation was wrong because it didn't look like the image on the assignment instructions. After discussing on discord we realized that we should only compare to the reference images on WebGL and that the picture on the assignment was misleading. It turns out we had the correct implementation the whole time. For the floor, for some reason only half was checkerboard and the other half was black. We don't know what caused this issue but by messing around with the intensity value we were able to fix it, but it was still slightly off (see below).

Known Bugs:
The shading on the floor seems a bit darker in our implementation than the reference picture. There are not any other known bugs.

Future Work:
We have completely finished this assignment and have no future work.