Menger Sponge

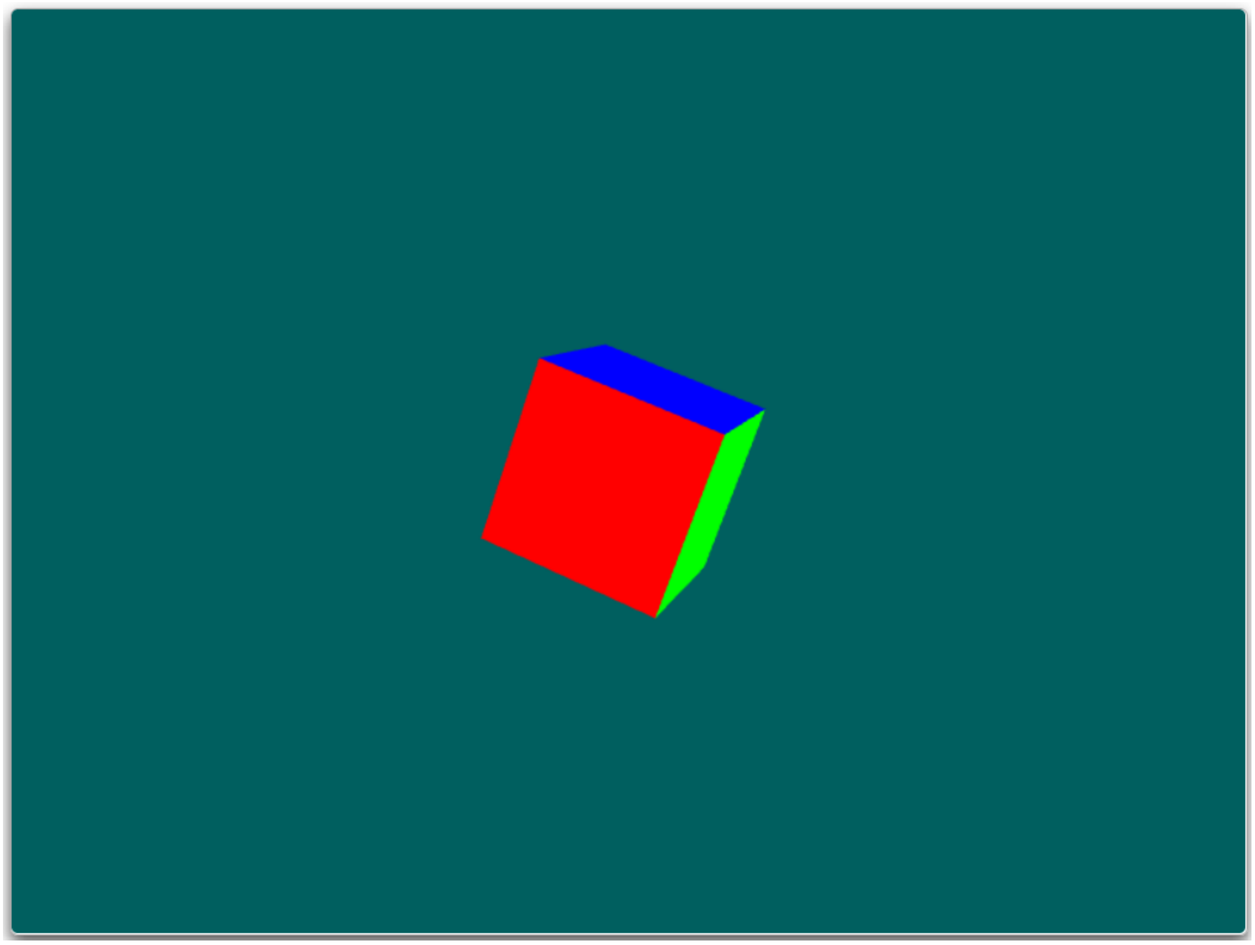
Building MengerSponge

This functionality was implemented in src/menger/mengerSponge.ts
So far only the level 0 cube was implemented. We rendered this cube by specifying 4 vertices per face, 4 normals per face, and 6 indices per face. This renders out a cube. Our original plan was to adapt this code into a make cube method that takes in minX, minY, minZ and maxX, maxY, maxZ parameters. We would use this parameters to select the actual vertex positions of the cube instead of hardcoding it in. We would call the draw cube method in a loop to draw out the current level of the menger sponge.
In pseudo code, this would look something like this:

```
If (this.level == 0) {
    drawCube(-0.5, 0.5) //The cube has a bounding box specified by (-0.5, -0.5, -0.5), (0.5, //
0.5, 0.5)
Else {
    Incr = 1/(this.level * 3)
    For (let i = 0; i < 1; i+= incr) {
        For (int j = 0; j < 1; j += incr) {
            drawCube(-0.5 + i, -0.5 + j);
        }
    }
}
```

We would also not call drawCube in the case of the 7 inner cubes. We do plan on implementing the rest of Menger Sponge for full/partial credit by the end of spring break

Code Snippets and Outputs:



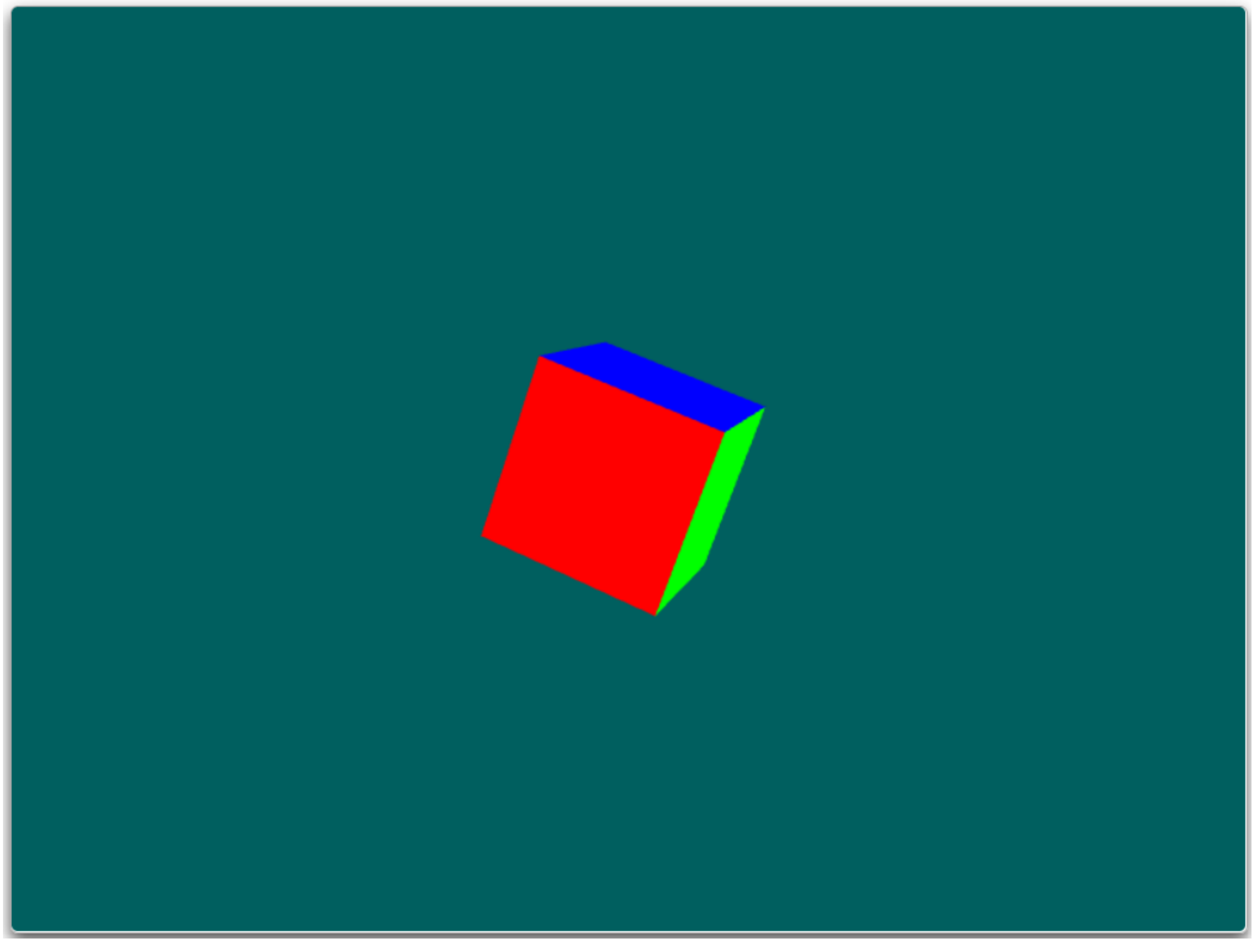
Camera Controls
This functionality was implemented in src/menger/Gui.ts.

Orbital Camera Rotation: This was implemented in the drag method. To do this, we converted the mouse_direction vector to world coordinates by multiplying it by the Inverse of the Projection and inverse of the view matrices. We then rotated the camera around an axis perpendicular this vector and the look vector

Code Snippets and Outputs:

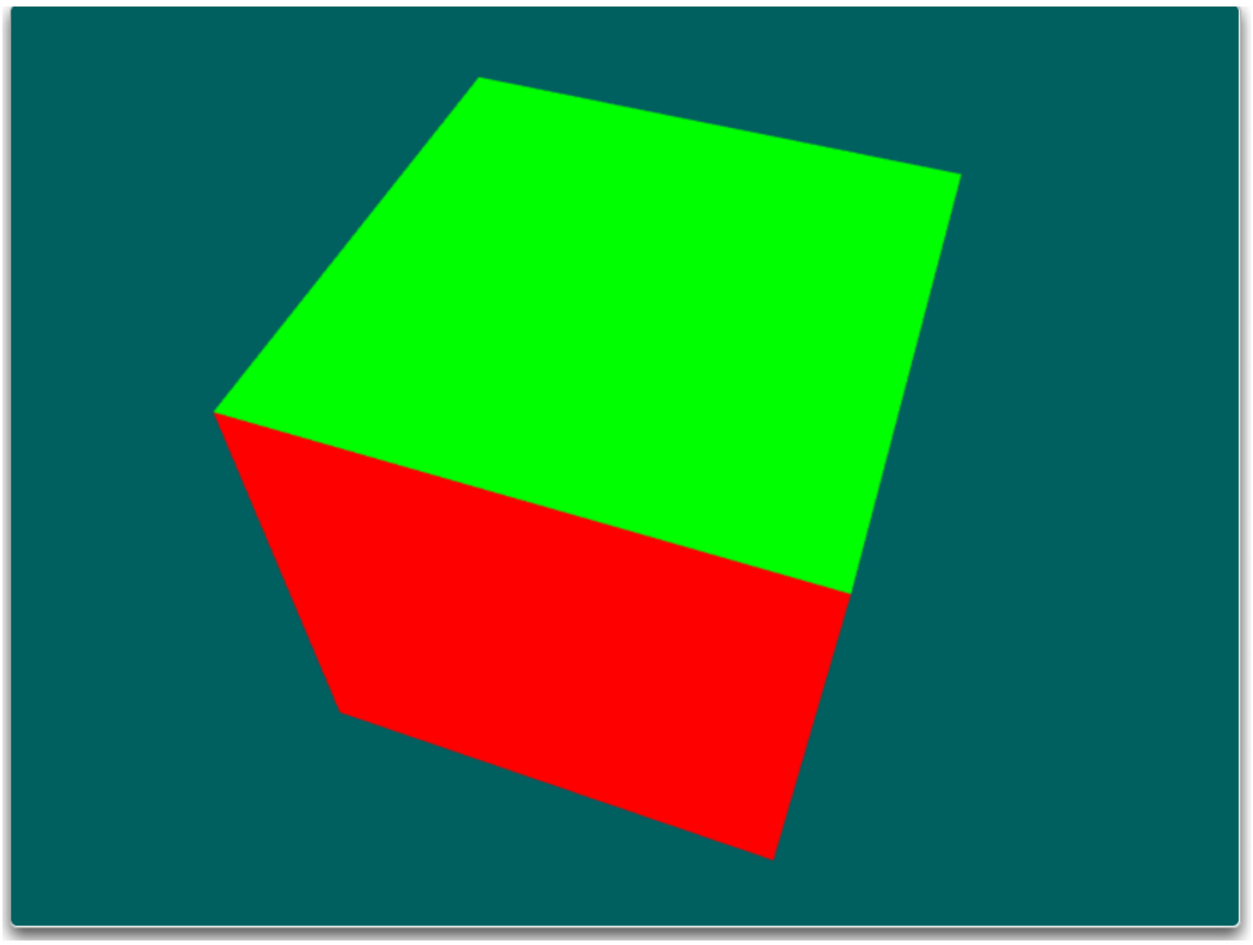
```
        const projMatrixI : Mat3 =
this.camera.projMatrix().toInverseMat3();
        const viewMatrixI : Mat3 =
this.camera.viewMatrix().toInverseMat3();
        const mouse_direction : Vec3 = new Vec3([mouse.screenX -
this.prevX, mouse.screenY - this.prevY, 0]);
        const w_mouse_dir : Vec3 =
viewMatrixI.multiply(projMatrixI).multiplyVec3(mouse_direction);
```

```
        const axis : Vec3 = Vec3.cross(this.camera.forward(),
w_mouse_dir);
        this.camera.rotate(axis, 0.05, this.camera.target());
```



Orbital zoom: this was implemented in drag and with the ‘w’ and ‘s’ keys. This was done using the offsetDistance function in Camera.ts, which reduced/increased the camera_distance

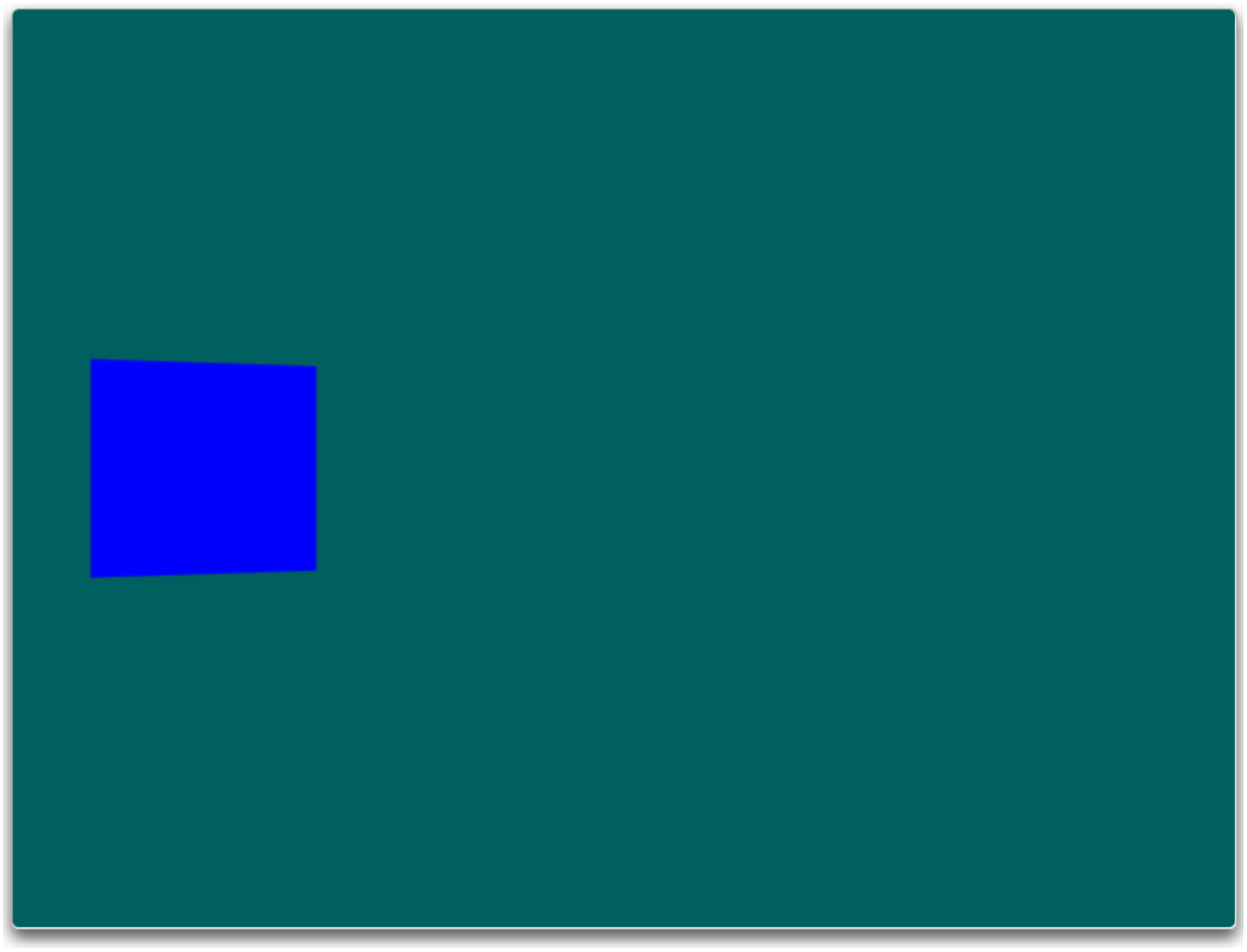
```
          this.camera.offsetDist(-0.1);
```



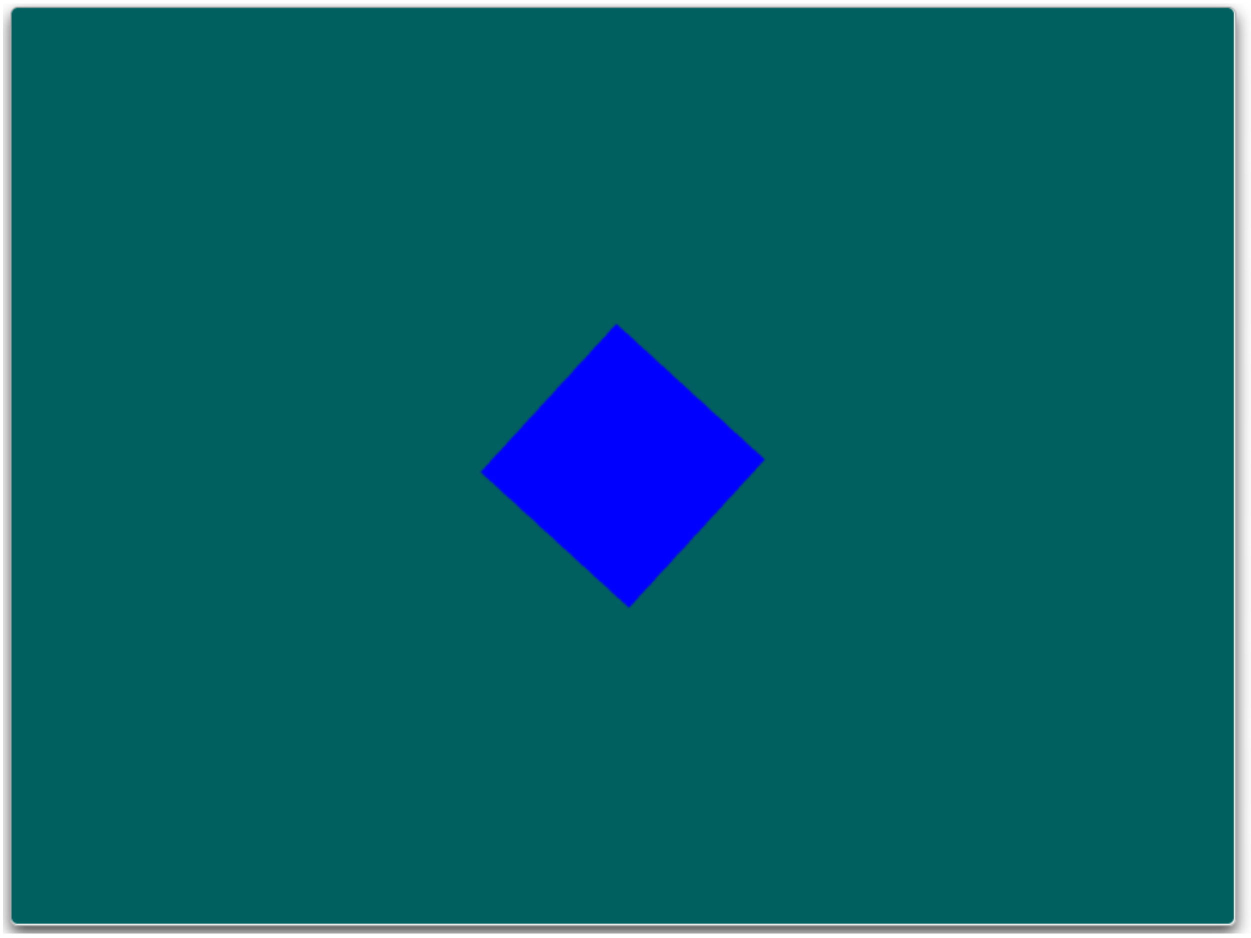
Orbital center shift: this was implemented with the ‘a’ and ‘d’ keys and with the up arrow, down arrow keys. This was done by shifting the target field of the camera by either the tangent or up vector

```
this.camera.setTarget(this.camera.target().add(this.camera.right().scale(0
.1)));


this.camera.setTarget(this.camera.target().subtract(this.camera.up().scale
(0.1)));
```



Rolling the camera: This was implemented with the arrow right and left keys. This was done by calling camera.roll



FPS functionality:
Pressing ‘c’ sets the fps boolean. The ‘w’ and ‘s’ keys causes translation of the eye and center with the look vector

```
this.camera.setPos(this.camera.pos().subtract(this.camera.forward().scale(0.1)));

this.camera.setTarget(this.camera.target().subtract(this.camera.forward().scale(0.1)));
```

‘A’ and ‘d’ causes the camera to move in fps mode

All camera functionality was fully implemented.

Shader

This functionality was implemented in src/menger/Shader.ts. The MengerSponge Fragment Shader was implemented:

```
export let defaultFSText = `
precision mediump float;
varying vec4 lightDir;
varying vec4 normal;


void main () {
    vec3 colors = vec3(normal);
    colors.x = (colors.x * colors.x);
    colors.y = (colors.y * colors.y);
    colors.z = (colors.z * colors.z);
    gl_FragColor = vec4(colors, 1.0);
}
`;
```

This fragment shader works by recognising the following property:
There are 6 possible values of normal:

1) (1.0, 0.0, 0.0, 1.0)
2) (-1.0, 0.0, 0.0, 1.0)
3) (0.0, 1.0, 0.0, 1.0)
4) (0.0, -1.0, 0.0, 1.0)
5) (0.0, 0.0, 1.0, 1.0)
6) (0.0, 0.0, -1.0, 1.0)

We want (1.0, 0.0, 0.0, 1.0) and (-1.0, 0.0, 0.0, 1.0) to be converted to the rgba value (1.0, 0.0, 0.0, 1.0), so in both of these cases, if we square the x, y, and z components we will get the correct rgb value. I.e, 0*0 is still 0, 1*1 is still 1, but -1*-1 is 1

We also have made some changes in App.ts to create a Floor object and render it (using Floor.ts), but we are still working on it.

Issues during implementation

- **Time constraints**

Known Bugs

- **Left clicking the mouse and dragging right quickly causes the cube to disappear for some reason**
- **Camera angle looks a little different from those in the tests, suspect it has something to do with rotation**

Future Work

We will finish the level 1, 2, 3, and 4 cube generation as well as floor rendering and shading incorporating directional lights during Spring Break.