CS 411 – Computer Graphics

Assignment 4 – Surface Rendering

Mayank Bansal – mbansal5@hawk.iit.edu

November 6, 2017

1. Problem Description

In this project, we were to write a WebGL program that satisfied the needs below:

a) ***Input file format:*** The text file describes on 3D object. Each line in that file describes either a vertex or a face.

b) ***Object Loading:*** Read vertex lines should be stored in a vertex array based on their order in the input file. Read face lines should be stored in a face array. Assume that all the faces are triangles. A face is specified by the indexes od three vertices in the vertex array.

c) ***Object Normalization:*** The coordinates of the loaded object should be scaled so that the larger side of its bounding box will have a length of one. he coordinates should be normalized so that the center of mass of the object will be placed at the origin. This may be achieved using a translation and scale transformation.

d) ***Vertex Normals:*** in order to obtain correct lighting of the object, for each drawn face (polygon) a normal vector should be specified.

e) ***Moving the Camera:*** after rendering the object, the user should be able to move up/down/left/right, zoom in/out

f) ***Normal Direction:*** The vertex winding order needs to be consistent.

g) Computing Vertex Normals: Replace face normal with average vertex normal

Support code, and a skeleton program was given to help start with the program.

2. Method and Problems Incurred

2.1 Input File Format

The template code given already parses the object file in ***obj_loader.js*** therefore no additional code or modifications were required to be made.

2.2 Object Loading

The object was also loaded using the code available in the template code in ***obj_loader.js***, so no code was required to be written here as well.

```
extractOBJFileArrays(model);
```

2.3 Object Normalization

The object needs to be normalized in such a way that the longest side of the bounding box has a scale factor of one uniform across the different objects. After that the center of mass of the object needs to be moved to the origin, i.e., (0,0,0). For this, we have written the ***normalizeObject()*** function.

First, we initialize our variables:

```
// Calculate Center of Mass of Object
cMass = {
    x: 0,
    y: 0,
    z: 0
};

// Calculate Bounding Box
boundingBox = {
    xMax: -9999999,
    xMin: 99999999,
    yMax: -9999999,
    yMin: 99999999,
    zMax: -9999999,
    zMin: 99999999
};
```

Then we loop through all the vertices and calculate the ***boundingBox*** and ***cMass***

```
for (var i = 0; i < model.arrays.vertices.length / 3; i++) {
    var point = {
        x: model.arrays.vertices[i * 3 + 0],
        y: model.arrays.vertices[i * 3 + 1],
        z: model.arrays.vertices[i * 3 + 2]
    };

    cMass.x += point.x;
    cMass.y += point.y;
    cMass.z += point.z;

    // Find bounding box
    if (point.x < boundingBox.xMin) boundingBox.xMin = point.x;
    if (point.x > boundingBox.xMax) boundingBox.xMax = point.x;
    if (point.y < boundingBox.yMin) boundingBox.yMin = point.y;
    if (point.y > boundingBox.yMax) boundingBox.yMax = point.y;
    if (point.z < boundingBox.zMin) boundingBox.zMin = point.z;
    if (point.z > boundingBox.zMax) boundingBox.zMax = point.z;
```

```
}

console.log(boundingBox);

cMass.x /= (model.arrays.vertices.length / 3);
cMass.y /= (model.arrays.vertices.length / 3);
cMass.z /= (model.arrays.vertices.length / 3);
```

Now that we have the bounding box and center of mass, we need to find the longest side of the bounding box and scale by that value:

```
// Find largest side of bounding box
var initScaleFactor = Math.max(
    boundingBox.xMax - boundingBox.xMin,
    boundingBox.yMax - boundingBox.yMin,
    boundingBox.zMax - boundingBox.zMin
);

mvMatrix.scale(100 / initScaleFactor, 100 / initScaleFactor, 100 /
initScaleFactor);
```

To be able to see the object, here I have scaled by a factor of 100. Even then all the loaded objects will be set to the same normalized reference scale.

After this we need to translate the object's center of mass to the origin. This can be done as follows:

```
if (cMass.x !== 0 || cMass.y !== 0 || cMass.z !== 0) {
    mvMatrix.translate(-cMass.x, -cMass.y, -cMass.z);
    cMass = {x: 0, y: 0, z: 0};
}
```

Here we check if the center of mass isn't at the origin and change it accordingly.

2.4 Vertex Normals

To obtain the correct lighting of the objects, we need to average the normal for each face. For that we have the ***averageNormals()*** function:

```
function averageNormals() {

    // Reset all the normals
    for (var i = 0; i < model.arrays.normals.length; i++)
        model.arrays.normals[i] = 0;

    // Loop through all the faces
    for (var i = 0; i < model.arrays.indices.length / 3; i++) {

        var p0 = [], p1 = [], p2 = [];

        for (var j = 0; j < 3; j++)  // vertices of face
            for (var k = 0; k < 3; k++) {  // x,y,z of vertex
```

```
                var point = model.arrays.vertices[3 * model.arrays.indices[3 *
i + j] + k];
                if (j === 0)
                    p0.push(point);
                else if (j === 1)
                    p1.push(point);
                else
                    p2.push(point);
            }

        var newNormal = calcNormal(p0, p1, p2);

        // Change Normals
        for (var j = 0; j < 3; j++)
            for (var j = 0; j < 3; j++)
                model.arrays.normals[3 * model.arrays.indices[3 * i + j] + k]
+= newNormal[k];
    }

    console.log('Normals Averaged');
}
```

Here we initially set all the normal to 0, loop through all the faces, find the vertices, average them by calculating new normal, and add it to the normal for that vertex. Since the way these objects were rendered, this function makes no change on the output.

2.5 Moving the Camera

For moving the camera, we can just move the objects instead. The functions written below zoom in/out and move up/down/left/right by simply multiplying the ***mvMatrix*** by the desired rotation.

```
function turnLeft() {
    tmpRot.set(leftRot);
    tmpRot.multiply(curRot);
    curRot.set(tmpRot);
    mvMatrix.multiply(leftRot);
}

function turnRight() {
    tmpRot.set(rightRot);
    tmpRot.multiply(curRot);
    curRot.set(tmpRot);
    mvMatrix.multiply(rightRot);
}

function turnUp() {
    tmpRot.set(upRot);
    tmpRot.multiply(curRot);
    curRot.set(tmpRot);
    mvMatrix.multiply(upRot);
}

function turnDown() {
    tmpRot.set(downRot);
    tmpRot.multiply(curRot);
```

```
    curRot.set(tmpRot);
    mvMatrix.multiply(downRot);
}

function zoomIn() {
    if (camZ < 1) camZ = 1;
    camZ += 0.3;
    console.log("CamZ: ", camZ);
    mvMatrix.scale(camZ, camZ, camZ);
    camZ = 1;
}

function zoomOut() {
    if (camZ > 1) camZ = 1;
    camZ = (camZ <= 0.1) ? camZ : camZ - 0.3;
    console.log("CamZ: ", camZ);
    mvMatrix.scale(camZ, camZ, camZ);
    camZ = 1;
}
```

2.6 Normal Direction

We need to compute the vertex winding order of the vertices to determine whether the normal are facing inside or outside the object. Since the object is already checked for winding order, here we just verify by calculating the decision parameter R = N.(C-X) where C is the center of mass, X is a random point on the surface (taken as v0 in this case) and N is the surface normal. The following code calculates the dot-product and inverts the normals if required:

```
var N, X, R = {value: null, set: false};

// check vertex windings
for (var i = 0; i < model.arrays.vertices.length / 3; i += 3) {

    var v0 = [
        model.arrays.vertices[i * 3 + 0],
        model.arrays.vertices[i * 3 + 1],
        model.arrays.vertices[i * 3 + 2]
    ];

    var v1 = [
        model.arrays.vertices[(i + 1) * 3 + 0],
        model.arrays.vertices[(i + 1) * 3 + 1],
        model.arrays.vertices[(i + 1) * 3 + 2]
    ];

    var v2 = [
        model.arrays.vertices[(i + 2) * 3 + 0],
        model.arrays.vertices[(i + 2) * 3 + 1],
        model.arrays.vertices[(i + 2) * 3 + 2]
    ];

    // Calculate Average Normal N = (v2-v1)x(v0-v1)
    N = calcNormal(v0, v1, v2);
```

```
    // test point v0 with normal N for winding order
    X = v0;

    // calculate X-C
    X[0] -= cMass.x;
    X[1] -= cMass.y;
    X[2] -= cMass.z;

    // calculate winding variable check R = N.(X-C)
    R.value = N[0] * X[0] + N[1] * X[1] + N[2] * X[2];

    // set R flag if R < 0
    if (R.value < 0 && !R.set)
        R.set = true;

    // print only a few Average Vertex Normals
    if (i < 36)
        console.log("Average Vertex Normal: Face(v" + i + ",v" + (i + 1) +
",v" + (i + 2) + ") ", N);

    // replace vertex normals with average
    for (var j = 0; j < 3; j++)
        for (var k = 0; k < 3; k++)
            model.arrays.normals[(i + j) * 3 + k] = N[k];
}

// check if R flag is set to invert normals
if (R.set) {
    for (var i = 0; i < model.arrays.vertices.length; i++)
        for (var j = 0; j < 3; j++)
            model.arrays.normals[i * 3 + j] *= -1;
}
```

A button was added to invert all the normals as well if the user wished to:

2.7 Computing Vertex Normals

The vertices are averaged using N = (v2-v1)X(v0-v1) in the code written in the previous section. We find the 3 points of the face, and then subject it to the function that calculates the value of N. The are averaged before using the winding order decision parameter to invert all the normals.

```
function invertNormals() {
    invertNorm = !invertNorm;

    for (var i = 0; i < model.arrays.vertices.length; i++)
        for (var j = 0; j < 3; j++)
            model.arrays.normals[i * 3 + j] *= -1;

    assignVertexBuffersData(gl, buffers, model);
}
```

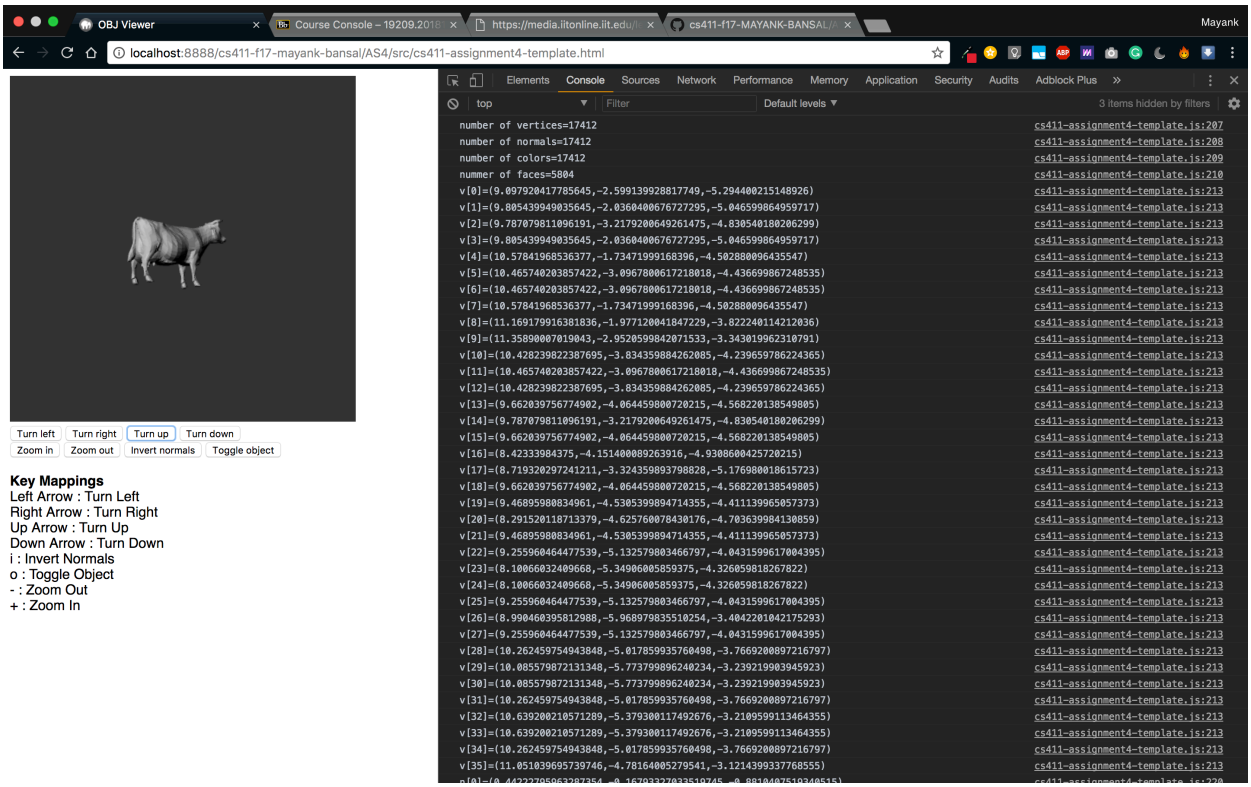
3. Results and Evaluation

Below there are the screenshot and notes for different required outputs mentioned in the problem statement:

3.1 Input File Format

The file was read according to the specifications

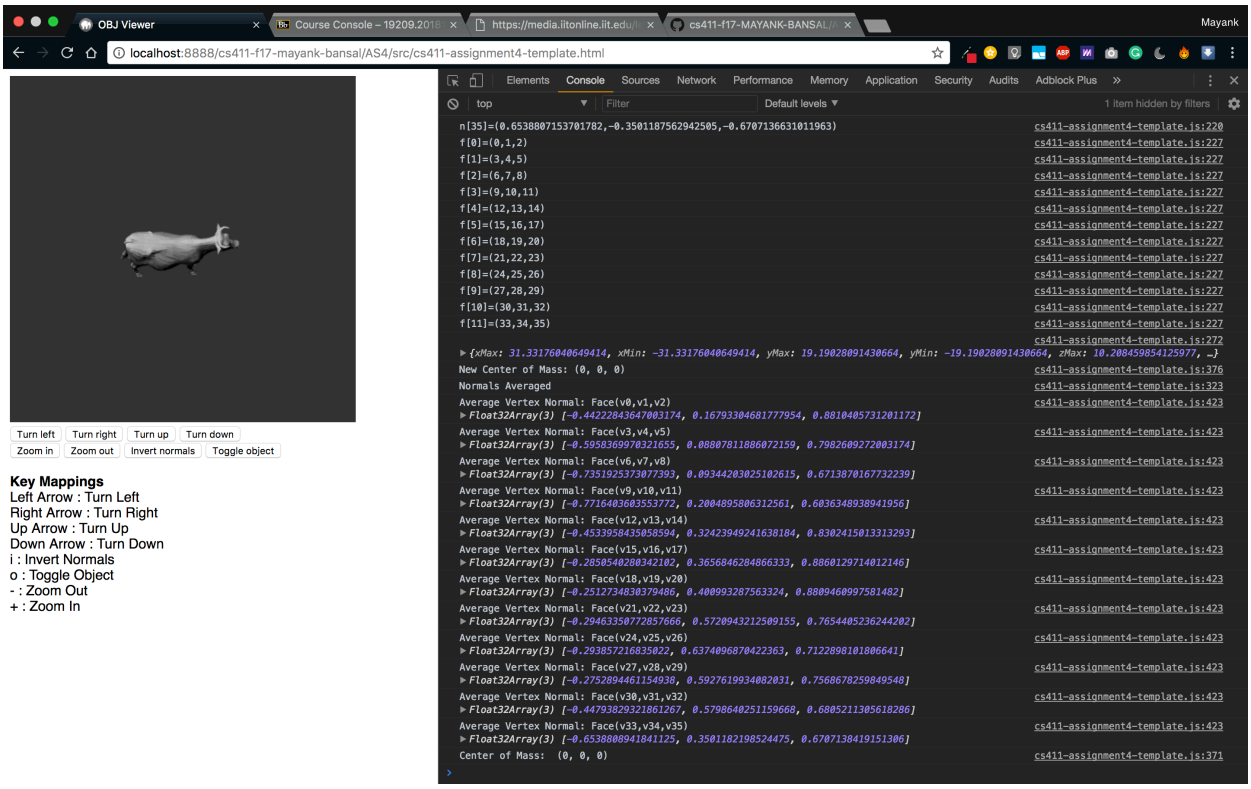
3.2 Object Loading

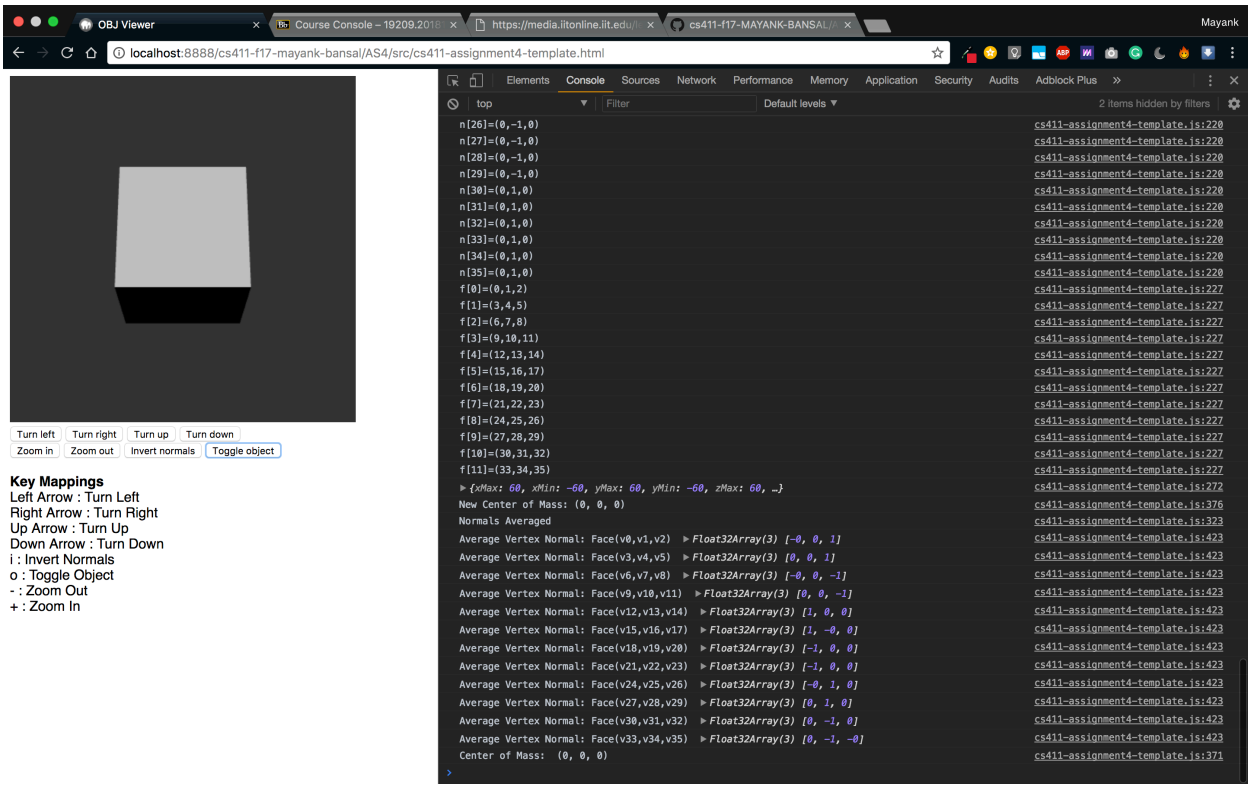


The console has information about the loaded object

3.3 Object Normalization

Here both the given objects roughly of the same size and have the center of mass shifted to (0,0,0) as shown in the console.





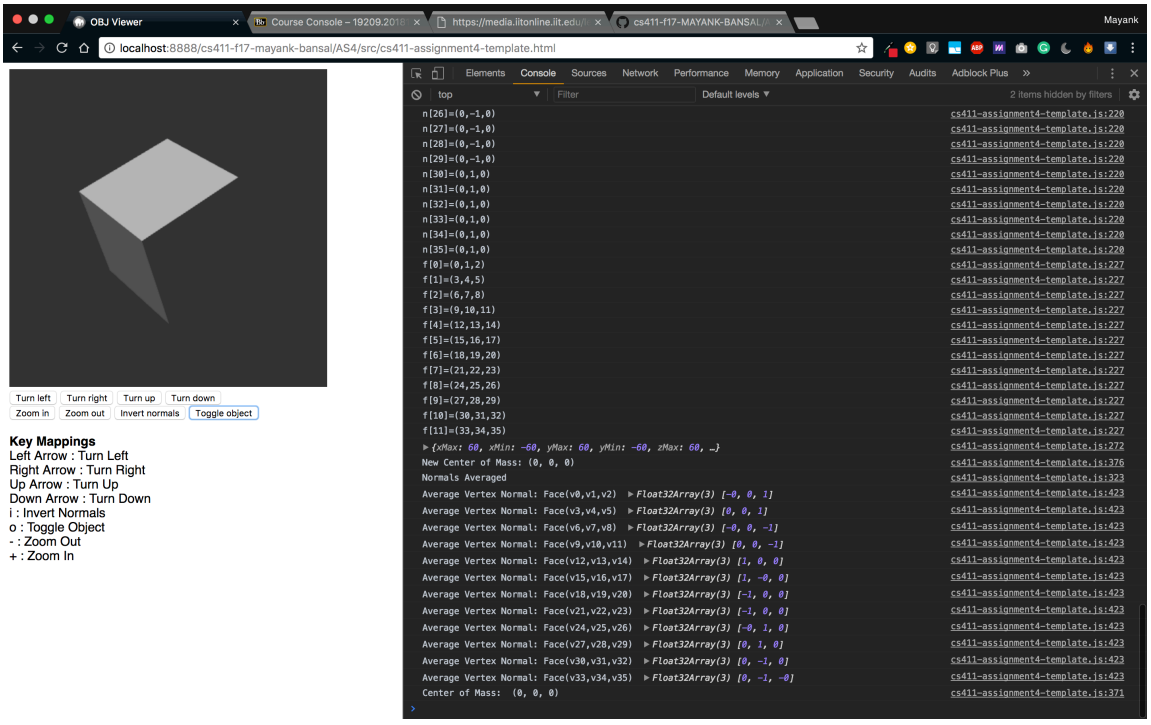
Here the longest side of the bounding box is normalized to the scale, and the new center of mass is shifted to (0,0,0)

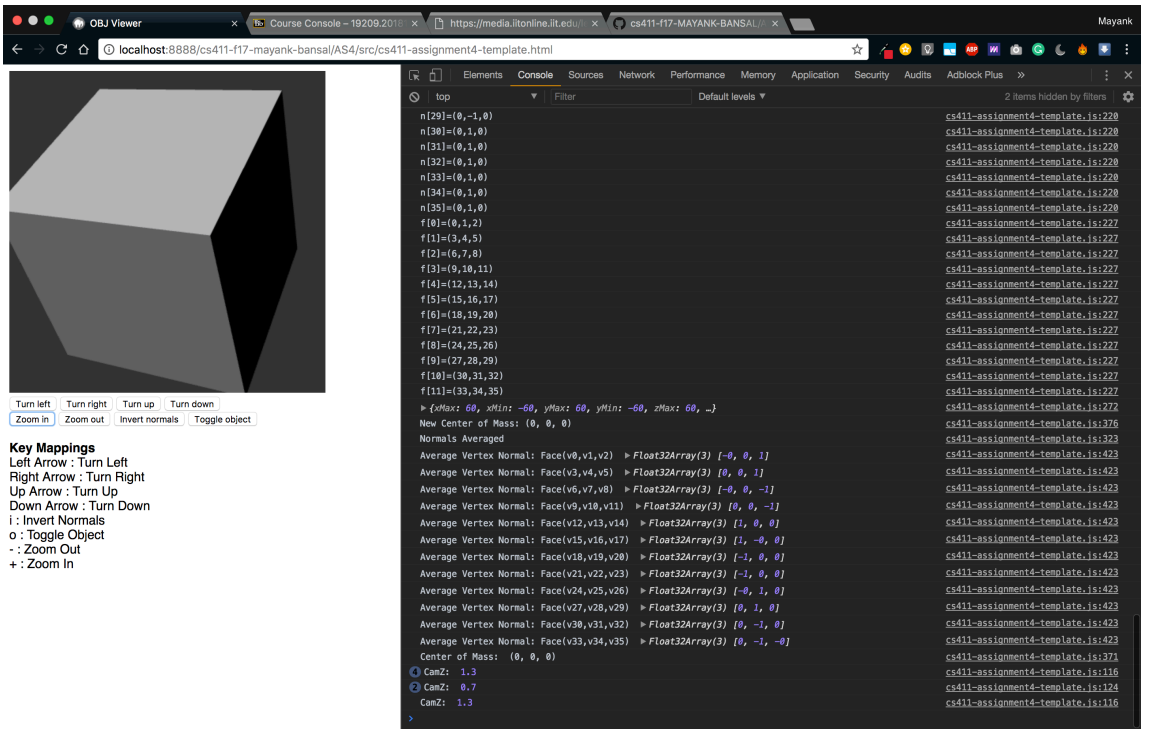
3.4 Vertex Normals

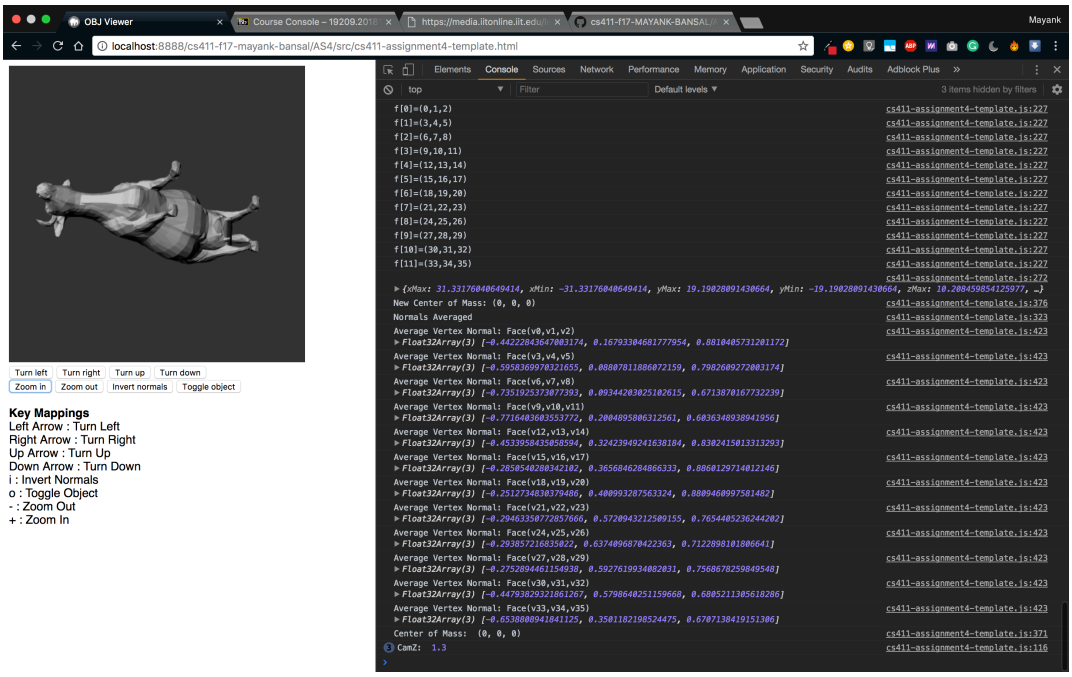
Since the objects were loaded in a different manner, the normalization of the vertices made no difference on the output.

3.5 Moving the Camera

Rotations, zoom in/ zoom out all were working as desired.





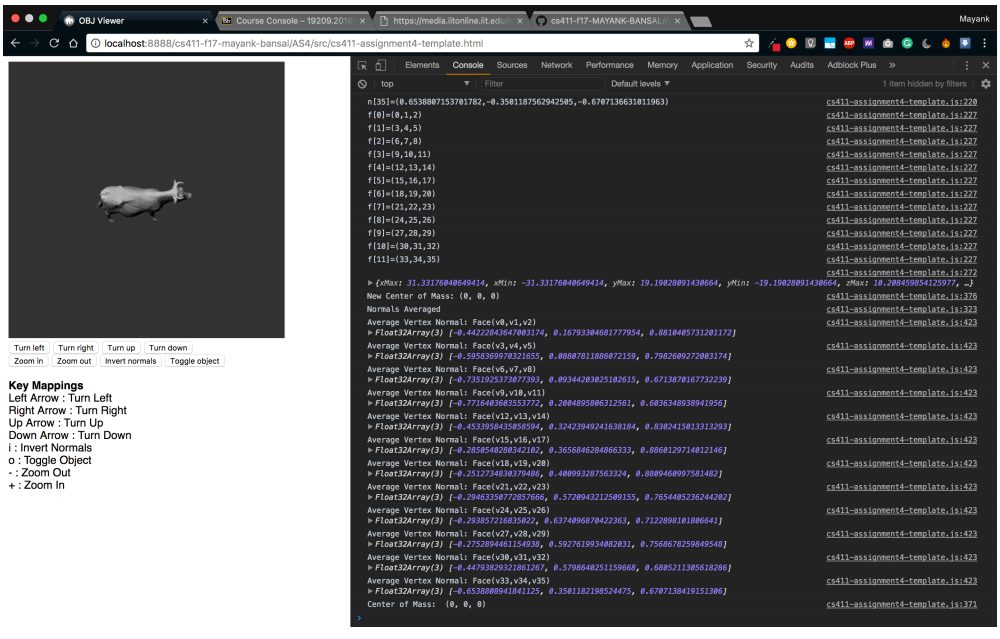


The user is also given the functionality of using the arrow keys to move the object.

3.6 Normal Direction

Since the files had consistent normal directions, the output was unchanged from the previous section.

3.7 Computing Vertex Normals



Here the vertex normal were calculated as shown in the console on the right.

4. Conclusion

The outputs shown in the previous section mimic the reference output given to us with no documented problems or missed cases.

5. Environment & Organization

This project was run and tested on ***Google Chrome Version 61.0.3163.100 (Official Build) (64-bit)***. Run out of a MAMP stack server. Simply pull the repository and run out of the SRC file.

The SRC folder contains 1 file:

cs411-assignment4-template.html

Contains solutions to the programming WebGL problem set, written in JavaScript.