Viewing and Analysing 3D Models Using WebGL

CM20219 – Foundations of Visual Computing, Dr. Christian Richardt

Submission by fn252

Introduction

WebGL is a cross-platform low-level 3D graphics Application Programming Interface that uses *JavaScript* wrapped in HTML5 for bringing plugin-free 3D graphics to websites, and is recognized by major browsers. [1] Support is widespread as to this day, independent libraries and APIs are made that are compatible for use in conjunction with it. One of such interfaces is three.js, which "allows the creation of GPU-accelerated 3D animations using the JavaScript language as part of a website". [2] This report discusses the implementation of three.js with WebGL in terms of my written code and the produced results.

Setup

First and foremost, global variables are initialized for use later in the code. A scene is created so that I can add all my different objects to it, and my camera is set up. In this case, I'm using a perspective camera that uses a field of view angle, an aspect ratio equal to the inner browser window, and the near and far clipping planes as shown below.

```
function setupCamera() {

        camera = new THREE.PerspectiveCamera(45,
        window.innerWidth / window.innerHeight,
        0.1, 1000);

        camera.position.set(3, 4, 5);

        camera.lookAt(new THREE.Vector3(0, 0,
0)); }
```

After the camera is initialized, a grid helper is added to the scene in the x-z plane. Next is the first requirement.

Requirement 1: drawing a cube.

```
function drawCube() {

        var geometry = new THREE.BoxGeometry( 2,
        2, 2 );

        var material = [];

        [... irrelevant code - requirement 7 ...]

        cube = new THREE.Mesh( geometry, material
        );

        [... irrelevant code - requirement 4 ...]

        scene.add(cube); }
```

The function I wrote for requirement 1 creates a cube with several properties that will be discussed later. The geometry of the cube is created to be of length, width, and height of 2 for each as for the cube to be centered at the origin with opposite corner points (-1, -1, -1) and (1, 1, 1). The cube's material is initialized and set next to create a mesh, making the cube by combining the geometry and material. [3]

Requirement 2: creating x, y, and z axes.

The function to make the axes on the plane uses a helper from *three.js*, where it adds orthogonal lines to represent the axes in

```
function drawAxes() {

        var axesHelper = new THREE.AxesHelper( 5
        );

        scene.add( axesHelper );}
```

RGB. The function initializes it, and adds it to the scene as shown below. [4]

Requirement 3: rotating the cube.

To make the cube rotate, I had to first initialise global variables to contain the rotation speed of each axis. The variables are used several times in different functions. The actual rotation happens in the function *animate*(), which calls itself and renders constantly, as shown to the right.

```
function animate() {

        requestAnimationFrame(animate);

        cube.rotation.x += rotationX;

        cube.rotation.y += rotationY;

        cube.rotation.z += rotationZ;

        renderer.render(scene, camera); }
```

At initialization, the rotation variables are set to zero. When the associated button for each of the axes is pressed, the respective rotation variable increases so that the *animate()* function makes the cube rotate and renders it immediately. For example, if the numerical button 1 is pressed for the first time, the cube will start its rotation in increments of 0.01 radians on the x-axis. [3]

If pressed again, it would stop by returning the variable to zero. See example below.

```
case 49:        // 1 = x rotation
        if (rotationX == 0.01){
                rotationX = 0.0; }
        else{ rotationX = 0.01; }
        break;
```

The function *handleKeyDown(event)* oversees handling all buttons by assigning a case to each button's key code that is activated.

Requirement 4: rendering view modes.

The code for rendering the cube's edges and vertices are in the same function as the one that draws the cube. They use the original cube to create their own properties. The variables *edges* and *vertices* are global as they are used elsewhere in the code. The default view mode for the cube is the face view mode. The code below shows the geometry creation of the edges, and the creation of the material for each point or vertex on the cube. [3]

```
function drawCube() {
        [... irr. code - requirements 1, 7 ...]
        var geometryEdges = new
        THREE.EdgesGeometry(geometry);
        edges = new
        THREE.LineSegments(geometryEdges, new
        THREE.LineBasicMaterial({ color:
        0xFFFFFF, linewidth: 2 }));
        var materialVertices = new
        THREE.PointsMaterial({ color: 0xFFFFFF,
        size: 0.2 });
        vertices = new THREE.Points( geometry,
        materialVertices );
        scene.add(cube); }
```

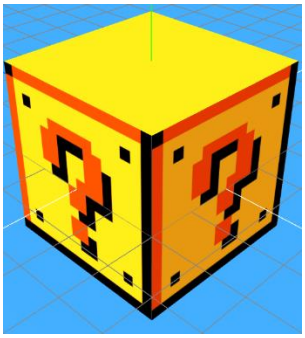
Each of the view modes are triggered by pressing the button that corresponds to the first letter of that mode, i.e. *v* for vertex view mode, *f* for faces, and *e* for edges. Every view mode has a function to handle them. In each function, the appropriate Boolean variable is used to either remove or add the feature.

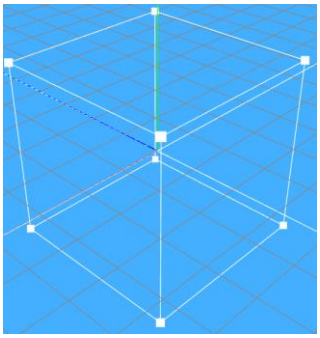
```
function vertexRenderer() {
        if (renderVertex) {
                scene.remove(vertices);
                renderVertex = false;}
        else {
                scene.add(vertices);
                renderVertex = true; }
```

The function *vertexRenderer()* was added as an example. The *renderVertex* Boolean is initially set to false. So when the button *v* is pressed and the function is called, it will add the vertices to the scene and make the Boolean true. If pressed again, vertices are removed and the Boolean would return to false. Each view mode has its own Boolean and they work similarly. All modes can be triggered independently, or simultaneously.

```
function addDirLight() {
        var directionalLight = new
        THREE.DirectionalLight( 0xffffff);
        directionalLight.position.set(1, 1,
        1).normalize();
        scene.add( directionalLight ); }
```

To make the sides of the cube have different shades for the face view mode, I added a directional light that shines on it. It is first initialized in the function and given a colour, white in this case. Its position is set to a point in the 3D space, and it's added to the scene.





The left picture shows the cube rendered in face view mode. On the right, only the cube's edge and vertex mode are rendered.

Requirement 5: translating the camera.

For this requirement, I used a built in translation. It is in fact divided into three, where each one is responsible for every axis. The translation happens in respect to the camera and not the scene itself. As a result, this method still works properly even when orbiting. An example of this is shown below.

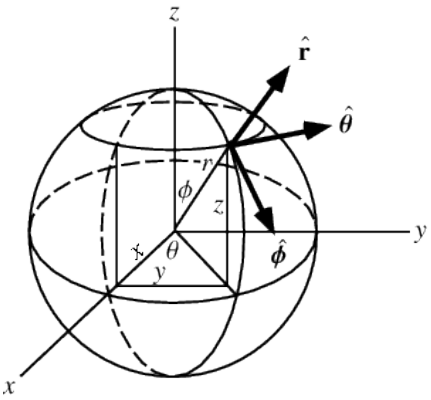
```
case 68:        // d key for move right
        camera.translateX(0.1);
        break;
case 83:        // d key for move right
        camera.translateY(-0.1);
        break;
case 90:        // d key for move right
        camera.translateZ(-0.1);
        break;
```

Requirement 6: orbiting the camera.

To make the camera orbit, three mathematical equations had to be used to convert polar coordinates into 3D cartesian coordinates. These equations were made into the function *cameraOrbit()*. They were derived using trigonometry, where rho is the distance from the origin to the camera, phi is the angle between the positive z-axis and the line from the origin to the camera, and theta is the angle between the positive x-axis and the line from the origin to the other focal point.

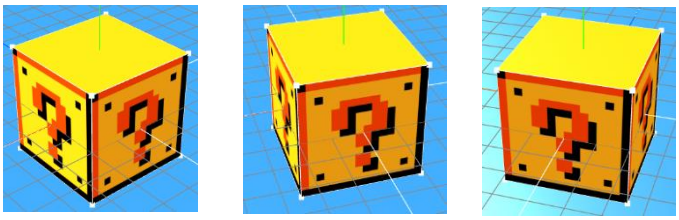
```
function cameraOrbit() {

        rho = camera.position.length();

        var cameraX = rho * Math.sin(phi) *
        Math.cos(theta);

        var cameraZ = rho * Math.sin(phi) *
        Math.sin(theta);

        var cameraY = rho * Math.cos(phi);

        camera.position.set(cameraX, cameraY,
        cameraZ);

        camera.lookAt(0, 0, 0); }
```

A sphere is shown below indicating the variables used in the equations. Below are the equations to get the specific coordinates for rotation:



$$x = \rho sin\varphi cos\theta$$
$$y = \rho sin\varphi sin\theta$$
$$z = \rho cos\varphi$$

However, conventions used in the equations are derived from physics rather than the typical mathematical convention, whereas the roles of theta and phi are reversed. To account for this reversal, the equation for *y* was switched with the equation for *z*, as shown in the code previously shown. [5]



Requirement 7: loading cube textures.

This requirement's code was written in the *drawCube()* function as follows.

```
function drawCube() {

        var geometry = new THREE.BoxGeometry( 2,
        2, 2 );

        var material = [];

        material.push( new
        THREE.MeshPhongMaterial( { map:
        THREE.ImageUtils.loadTexture(
        'qboxside1.png'), overdraw: true } ) );

        material.push(... 'qboxside2.png' ...);

        material.push(...);  //done 6 times total

        cube = new THREE.Mesh( geometry, material
        );

        [... irrelevant code - requirement 4 ...]

        scene.add(cube); }
```

After initializing the *material* array, six images are initialized and pushed into the array to form each side of the cube. When loading external images from a local drive, browsers tend to block the request, and so the page had to be loaded from an external server. The images use a *three.js* function called *loadTexture*, which is mapped onto a material type on each side. [6]

Requirement 8: loading external object into the cube.

An external object loader called *OBJLoader.js* was imported into the code to load the bunny mesh from the file provided. It was placed in the function *initBunny()* which is an initialisation function similar to the one used at the beginning, only it's dedicated for the object. The loader loads the resource object and calls a function afterwards that contains the code for scaling the bunny to fit the cube. [7]

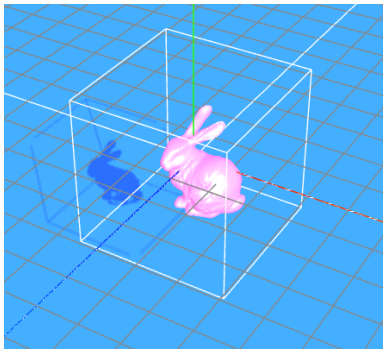
```
function initBunny() {

        var loader = new THREE.OBJLoader();

        loader.load(

                function (object) {

                        [code for scaling]

                        [requirement 9 code ...]

        scene.add(bunny); }
```

The method of scaling I used for the bunny was done using a JavaScipt element called boundingBox, which sets the boundaries of the object in the form of a cube. The function takes all the object's children, which in this case is the bunny's vertices, gets the minimum x, y, and z coordinates and subtracts them from the maximum respective coordinate. The ratios are

then created by dividing one over each of the three sizes, and then used as parameters for the built-in scaling function. The following code is contained in the function in *initBunny()*: [8]

```
function initBunny() {

        ...

        object.children[0].geometry.computeBoundi
        ngBox();

        var sizeX =
        object.children[0].geometry.boundingBox.m
        ax.x -
        object.children[0].geometry.boundingBox.m
        in.x;

        var sizeY =
        object.children[0].geometry.boundingBox.m
        ax.y -
        object.children[0].geometry.boundingBox.m
        in.y;

        var sizeZ =
        object.children[0].geometry.boundingBox.m
        ax.z -
        object.children[0].geometry.boundingBox.m
        in.z;

        object.children[0].geometry.scale(1/sizeX
        , 1/sizeY, 1/sizeZ);

        bunny = object.children[0];

        scene.add(bunny); }
```

The image to the right displays the bunny after being loaded and scaled down to fit inside the cube. The function works in a way that allows for any object to be scaled, and not just the bunny provided.

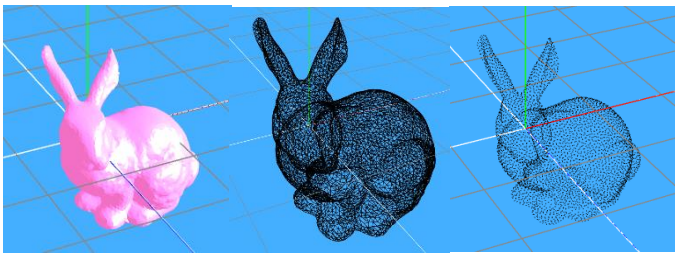


Requirement 9: rotating and rendering modes for bunny.

The code to rotate the bunny mesh and render it in the three different modes uses code almost identical to the one used for the cube. Bunny rotation was implemented the same as in requirement 3 where a global rotation variable was initialized and manipulated using assigned buttons. The function *animateBunny()*, which works identically to *animate()* from requirement 3, is used to constantly animate the rotation of the bunny. The code from requirement 4 was used to make the face, edge, and vertex rendering modes. The geometry of the bunny was used to create lines as edges and points as vertices. All models for the different modes were created when the bunny was loaded and initialized, so this only occurs once. This code is inserted after the bunny the code that scales down the bunny.

```
function initBunny() {

        ...

        bunny.material = new
        THREE.MeshPhongMaterial( { color:
        0xF26FC6, specular: 0x555555, shininess:
        30 } );

        var bunnyGeometry = new
        THREE.EdgesGeometry(bunny.geometry);

        bunnyEdges = new
        THREE.LineSegments(bunnyGeometry, new
        THREE.LineBasicMaterial( { color:
        0x000000, linewidth: 2 } ));

        var pointsMaterial = new
        THREE.PointsMaterial( {size: 0.005,
        color: 0x000000 } );

        bunnyVertices = new
        THREE.Points(bunny.geometry,
        pointsMaterial);

        scene.add(bunny); }
```

The following images show the different rendering modes for the bunny. From left to right, the bunny is in face, edge, and then vertex rendering mode.



The texture on the bunny is just a colour mapped out to it and not loaded from a local image, with the addition of the directional light. How much it reflects the light depends on the *shininess* value when creating the material.

Requirement 10

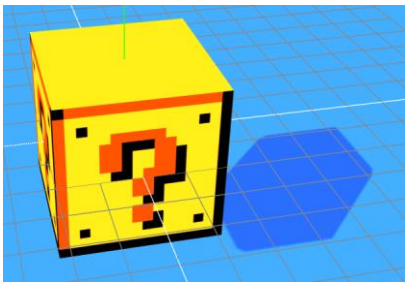
I added a ground plane and casted shadows on it from my objects for this requirement. A separate function was made for its initialization and insertion. It creates the geometry and the material first, and just like the creation of the cube in requirement 1, a mesh combining both is created and added to the scene. Although, when the plane is first created, its default position is vertical and goes through the center. To adjust, I rotated the plane on the x-axis to make it horizontal, and translated it 5 points under the origin – effectively making it a ground plane.

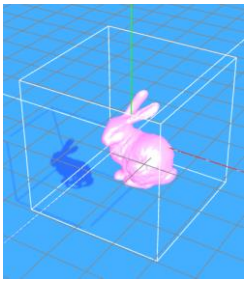
```
function addGroundPlane() {

        var groundGeometry = new
        THREE.PlaneGeometry(48, 48);

        var groundMaterial = new
        THREE.MeshPhongMaterial( { color:
        0x2b6fff, specular: 0x555555, shininess:
        30 } );

        var groundPlane = new
        THREE.Mesh(groundGeometry,
        groundMaterial);

        groundPlane.translateY(-5);

        groundPlane.rotateX(-Math.PI/2);

        groundPlane.receiveShadow = true;

        scene.add(bunny); }
```

For each object to cast a shadow, the castShadow Boolean must be set to true, and the plane's receiveShadow Boolean must be set to true. This must be done before each object's addition to the scene. The lines shown below were added to each object's respective initialization code.

```
cube.castShadow = true;

edges.castShadow = true;

bunny.castShadow = true;

bunnyEdges.castShadow = true;
```

The result produced is as below.





Discussion

Personally, I didn't know about WebGL before this and so I found it to be a really easy way to build 3D environments into websites. The external libraries/APIs were especially more useful and easier to use rather than manipulating everything manually.

The biggest limitation I encountered was using old code instead of renewing my code as I research different functions and methods to complete tasks. At the beginning, my code would initialise the vertices of the cube each time the *animate()* function is called, which caused a significant drop in frame rate when rotation is active. Even toggling different rendering modes took more time than it should. After researching more of course, I realized that there are many ways to make the system more efficient.

As for actual limitations when coding, the only one I faced was when loading textures onto the cube. Web browsers have privacy settings and blocks the use of local files as they are classified as private. To circumvent this issue, I made a local server of the folder with the code and relevant files using Python.

Extensions

The first extension I would do for this project is making another invisible object to serve as the look-at point for the *cameraOrbiting()* function. As it currently stands, attempting to orbit after translating will just reset the look-at point to the origin instead of the new point.

Another possible extension is to create more objects and a full background to make a more complete environment. Animations can be made for these objects as well. Seeing as my loaded textures on the cube represent the question box object from the videogame Super Mario Bros., an appropriate animation would be a coin and a sound triggered by a mouse click on the object.

Conclusion

All in all, WebGL shows much promise as it's plugin-free and uses two well known languages. Its royalty-free and open-source standard has enabled it to grow faster, providing the public with more ways to design and create in 3D environments.

References

[1] WebGL – OpenGL ES for the web. July 19, 2011. Available at: https://www.khronos.org/webgl/

[2] Stackoverflow.com. 2010. Available at: https://stackoverflow.com/tags/three.js/info

[3] three.js documentation. Available at: https://threejs.org/docs/#manual/introduction/Creating-a-scene

[4] three.js documentation. Available at: https://threejs.org/docs/#api/helpers/AxesHelper

[5] Stackoverflow.com. 2010. Available at: Mathworld.wolfram.com. (2017). *Spherical Coordinates -- from Wolfram MathWorld*. [online] Available at: http://mathworld.wolfram.com/SphericalCoordinates.html

[6] three.js documentation. Available at: https://threejs.org/docs/#api/loaders/CubeTextureLoader

[7] three.js documentation. Available at: https://threejs.org/docs/#examples/loaders/OBJLoader

[8] three.js documentation. Available at: https://threejs.org/docs/#api/math/Box3