Visual Report

Requirement 4 – Different render modes

For my implementation of requirement 4, being rendering the cube in different ways, I used different key presses for each of the different render modes, so you could change to any one of them with just one key press.

When rendering the cube in wireframe mode, I had to change the material of the cube, so that there was no shadow when rotating the cube. I did this by changing cube.material to a MeshBasicMaterial as opposed to MeshPhongMaterial, which allowed for shadows (See fig. 1 and 2).

```
basicMaterial = new THREE.MeshBasicMaterial({color: 0xff0000});
cube.material = basicMaterial;
```

To make the wireframe, the program will take the coordinates of each of the vertices of the cube and render the triangle primitives that make up the cube. Each time the animate function is called, the values of the vertices will be updated and the triangles will also be updated according to the changes in vertex positions. To change the render mode to the wireframe, I just set the wireframe property of the cube to true, and update the render.

```
cube.material.wireframe = true;
```

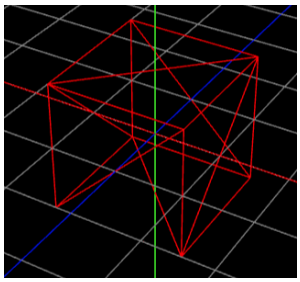


Fig. 1: Image of the wireframe model

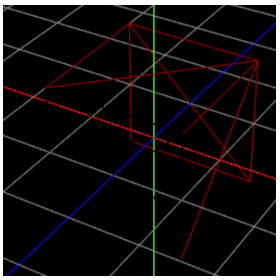
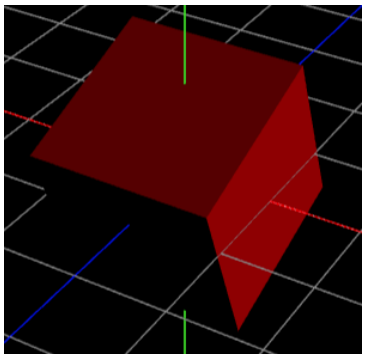


Fig. 2: Wireframe model when the material has not been changed

The material used for the face rendering mode is a Phong material, which means that there will be lighting applied to it, which will depend on which direction the light is coming from (in this case, it is coming from behind the cube). A Phong material will use the diffuse lighting, specular lighting and ambient lighting. The ambient lighting refers to if light was spread equally across the entire model. The specular parts will be closer to the light, so they will be brighter than the rest of the model. The diffuse lighting refers to the loss of light in the model as it moves further away from the light, or has an opaque part of the model between it and the light, blocking it. If you add all of these parts together, you will get the Phong lighting for the cube:



As you can see in the picture, the parts closer to the light are brighter, and the faces that are more blocked are a lot darker.

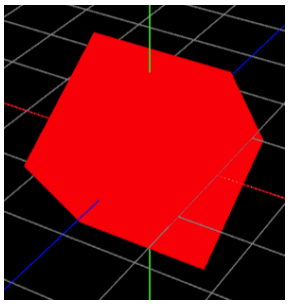
Fig. 3: Face model with shading

The full equation for the Phong lighting model is:

$$I = k_a i_a + k_d i_d \cos\theta + k_s i_s \cos^{\alpha}\phi$$

Where k is reflectivity, i is intesity, θ is the reflection angle and φ is the angle between the reflected ray and the viewing ray.

Initially, with just a basic material and no lighting, the cube looked like this:



As you can see, the light is spread evenly throughout the entire cube, meaning it is just the ambient light that is used, as opposed to using the diffuse and specular lightings as well.

Fig. 4: Face model without shading

To make the vertex model for the cube, I had to make an entirely new object with exactly the same dimensions as the initial cube and rotated at the same time as the cube. When I press the key to show the vertex model, I replace the cube with the vertex model, which will be in the exact same place, creating the illusion that they are the same model.

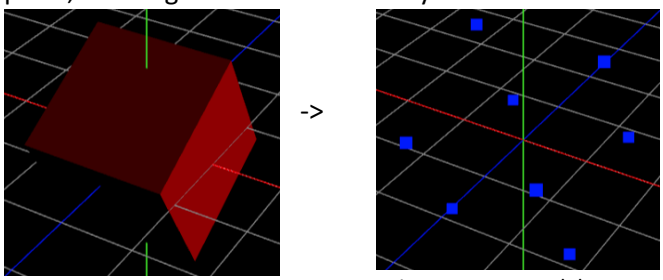


Fig. 5: Vertex model

These are in exactly the same position and just replace each other when the render mode is changed. I check to see if the vertex model is in the scene by giving the vertex model the name: 'Points' and use scene.getObjectByName('Points') to look for it. This makes sure I don't try to remove something that isn't there when changing render modes.

```
points.name = 'points';
if (scene.getObjectByName('points')){
    scene.remove(points);
    scene.add(cube);
}
```

Requirement 7 – Texture mapping

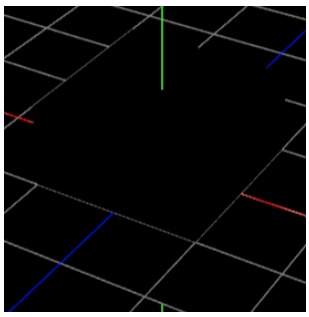
To add the image textures to the cube, instead of using one texture image and wrapping it around the cube, I had to find 6 separate png files that I was to use for the 6 different faces of the cube. I then made a new material that could be added to the cube that incorporated these 6 images:

```
textureMaterials = [
    new THREE.MeshBasicMaterial({ map:
loader.load('http://localhost:8000/Sid.png') }),
    new THREE.MeshBasicMaterial({ map:
loader.load('http://localhost:8000/Sasuke.png')}),
    new THREE.MeshBasicMaterial({ map:
loader.load('http://localhost:8000/Poker.png')}),
```

```
    new THREE.MeshBasicMaterial({ map:
loader.load('http://localhost:8000/Blackjack.png')}),
    new THREE.MeshBasicMaterial({ map:
loader.load('http://localhost:8000/Prequel.png')}),
    new THREE.MeshBasicMaterial({ map:
loader.load('http://localhost:8000/Duck.png')}),
];
```

This is a list of the separate images that correspond to the 6 faces of the cube, so that each image would only span one face, instead of leaking into other faces.

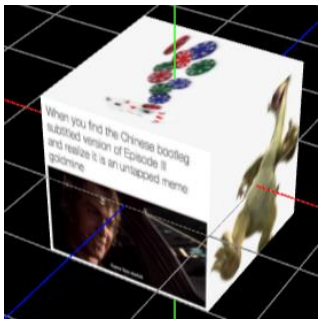
In order to get this to work, I needed to make sure each of the 6 images was exactly square and had exactly the same dimensions as each other, otherwise none of the images would render and the cube just shows up black in the scene:



This also occurs when trying to load the textures and html file straight from the file explorer, without hosting a server for the images and html file. This is because the security settings of the browser block the loading of the images. This is why, in the code, the image files are gotten from localhost:8000, as this is the server I was hosting the images on.

Fig. 6: Texture model when it cannot load the textures

It only allows square images to be used because the texture mapping uses the vertices of the cube and the vertices of the image to scale the image so that it perfectly fits within the face of the cube. It takes each vertex of the image and the corresponding vertex of the cube, and fits the texture over it. If the image were not square, the loading function would be unable to scale the image to fit the cube perfectly. Once I managed to scale the images to be the same size as each other, the cube looked like this:



As you can see, all of the images span exactly their own face

Fig. 7: Texture model when textures can be loaded

Requirement 9 – Rotate the mesh, render it in different modes

Once the bunny model has been rendered using OBJLoader() and has been scaled to fit inside the cube, we can then transform it further by rotating it. In my implementation, I use the same function as I do for rotating the cube which, every frame, adds 0.05 to bunny.rotation.(x, y or z depending on which way I want to rotate it). If, for example, I was adding 0.05 to bunny.rotation.x, what the function actually does is multiplies every vertex in the bunny model by the matrix:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

Where θ = 0.05 radians

Once it has done this for every vertex, all of the other parts of the model are updated to accommodate the change to all of the vertices. This means it needs to update all of the vertex normal and primitives that make up the model, so that it still has its overall shape.

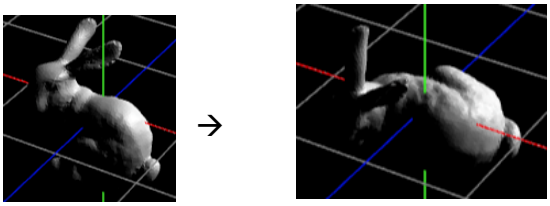


Fig. 8: initial bunny model Fig. 9: Rotated bunny model about x axis

```
if (scene.getObjectByName('bunny')) {
   bunny.rotation.x += 0.05;
}
```

If I wanted to make the bunny do a full rotation in a certain number of frames, for example 60 frames, I would make θ = 2π/60 since θ is measured in radians, and 2π radians is a full rotation, so it would finish a full rotation after 60 iterations(frames).

If I want to rotate the bunny through the y or z axis, I would use bunny.rotation.y or bunny.rotation.z respectively. The matrices that the vertices are multiplied by in these cases are:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad \text{And} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

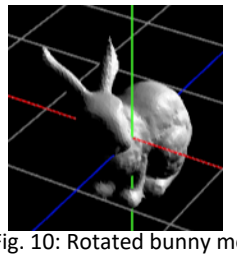


Fig. 10: Rotated bunny model about y axis

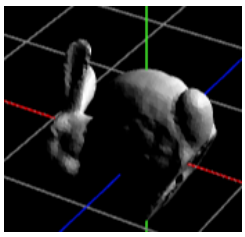


Fig. 11: Rotated bunny about z axis

When rendering the bunny in different modes, since I was working with a .obj file, I had to do it slightly differently than I did while working with the cube (See requirement 4). In my implementation for the bunny, I made a function as a parameter for the OBJLoader function (https://threejs.org/docs/#examples/en/loaders/OBJLoader) that traverses the 'children' of the object (in this case it is just the bunny model). In this function I add wireframe and points models to the child and initialise them as being invisible. Then, when you press the key to toggle the different render modes, it makes the next one visible and the other invisible.

```
wireframeMaterial = new THREE.LineBasicMaterial({color: 0xff00ff});
pointMaterial = new THREE.PointsMaterial({ color: 0x0000ff, size : 0.02 });
wireframe = new THREE.LineSegments(wireframeGeometry, wireframeMaterial);
point = new THREE.Points(child.geometry, pointMaterial);
child.add(wireframe);
child.add(point);
```

The way the wireframe and vertex models of the bunny work are the same as they work for the cube, just on a larger scale (5000 vertices as opposed to 8). For the wireframe, the program draws

the outline of all the triangle primitives, without any textures, using the vertices and connecting them to each other with lines. This shows very clearly that the bunny is entirely modelled from triangles that are connected together via shared vertices and edges, instead of just being one smooth model. For the vertex model, the obj files already has access to the positions of each of the 5000 vertices, so the function can just access these through the OBJLoader. It then just highlights these vertices and removes all of the connections between them and any textures, leaving just the vertices.

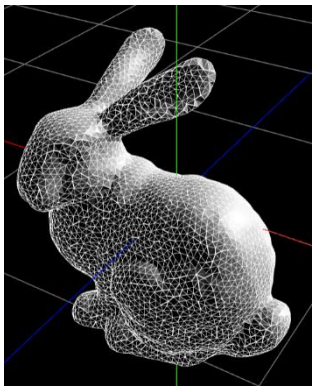


Fig. 12: Wireframe model of bunny

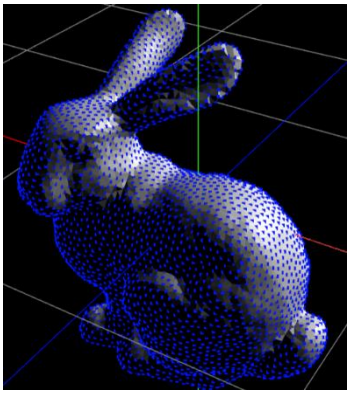


Fig. 13: Vertex model of bunny

As you can see from the screenshots, I was only able to overlay the different render modes over the existing model, since I could not find a way to make the rest of it invisible. If I were to make this again, I would find a way to make the scene only show the render mode that I want, instead of just overlaying the wireframe and vertex models on top of the face model.

Requirement 10 – Be Creative

For my extension task, I decided to make a short animation of someone picking up and folding a poker hand. This animation showed multiple forms of transformations, being translation and rotation. I also made a plane to act as the poker table and added textures to the cards.

Since the cards were not cubes, I had to change the images so that they were not square, but I still had to make them exactly the same dimensions as each other, otherwise the textures would not be loaded in and all that would appear would be a black texture (see Fig. 6).

I used the same method of placing the textures onto the model as I did for the cube (see requirement 7), by making a material that is a list of images and making them the faces of the model:

```
cardGeometry1 = new THREE.BoxGeometry(1.125, 0.03, 1.75);
cardMaterial1 = [
   new THREE.MeshBasicMaterial({map:
loader.load('http://localhost:8000/Black.png')}),
   new THREE.MeshBasicMaterial({map:
loader.load('http://localhost:8000/Black.png')}),
   new THREE.MeshBasicMaterial({map:
loader.load('http://localhost:8000/CardBack.png')}),
   new THREE.MeshBasicMaterial({map:
loader.load('http://localhost:8000/Ah.png')}),
   new THREE.MeshBasicMaterial({map:
loader.load('http://localhost:8000/Black.png')}),
   new THREE.MeshBasicMaterial({map:
loader.load('http://localhost:8000/Black.png')}),
```

```
]
card1 = new THREE.Mesh(cardGeometry1, cardMaterial1);
```

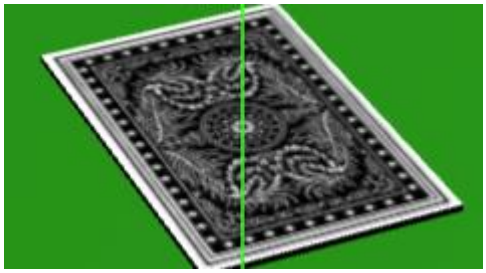


Fig. 14: Back of the playing cards

Fig. 15: Front of the playing cards.

For the table (which you can see in the background of figures 14 and 15), I made a plane that fit over the entire grid in the scene:

```
const planeGeometry = new THREE.PlaneGeometry(10, 10);
const planeMaterial = new THREE.MeshPhongMaterial( {color: 0x00ff00, side:
THREE.DoubleSide} );
plane = new THREE.Mesh(planeGeometry, planeMaterial);
```

For my implementation of the animation, I had to use Boolean toggles and a counter for the number of frames each part of the animation would take as, since the animate function is called every frame, my animation would take too long to run through until the function is called again, so I had to run through each part of the animation one at a time to ensure it would be completed, resetting the counter after each section of the animation was completed. During the part of the animation where the card values are being shown to the user, I had to translate the back card away from the user, otherwise the two cards would overlap and the textures would end up overlaying each other, creating a flickering effect.

The rotation of the cards in each part of the animation works exactly the same way as it does for the cube and the bunny (see requirement 9 for the exact rotation matrices), where each vertex position for the model is multiplied by the rotation matrix, rotating the model by θ radians. To ensure I rotated the models the correct amount for each part of the animation (I wanted to rotate by π/2 radians), I made θ = (π/2)/no. frames for that part of the animation (in this case 60).

For translation, if, for example, I wanted to translate the model in the x direction by 3, the program would use this equation for each vertex position:

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

https://www.gpp7.org.in/wp-content/uploads/sites/22/2020/04/file_5e9df44854704.pdf

where $T_x = 3$ and $T_y = T_z = 0$.

If I were to make this animation again in the future, for the final part of the animation, I would give the cards' path more of an arc instead of just a straight line towards the plane, so it would seem more realistic. I would also have tried to make an actual model for the poker table, making it look more like a table than just a coloured plane to give it an extra bit of realism.