CM20219 – Coursework Part 2

2020/2021 Viewing and analysing 3D Models using WebGL

1. Draw a simple cube

The first requirement was simple to execute and only required 4 lines of code. The components of a cube are the: geometry, material, and mesh.

```
var cube_geometry = new THREE.BoxGeometry(2,2,2);
var cube_material = new THREE.MeshBasicMaterial({ color:
0xffff00 });
cube = new THREE.Mesh(cube_geometry,cube_material);
scene.add(cube);
```

The geometry is what determines the size of the cube and was set to (2,2,2) so that it would have opposite corner points at (−1, −1, −1) and (1, 1, 1). The material is what allows us to be able to see the cube and is where we can apply colour (and later texture) to the cube. The geometry and the material are then combined to produce the mesh. When the cube is added to the scene, the cubes position value is automatically set to be the origin.

(See Figure 1).

2. Draw coordinate system axes

The axes were drawn using coloured lines between pairs of points on each of the axis. The size of the lines was 10 units each, so the coordinate pairs were between -5 to +5 on each axis. Each of the lines used a different line material colour (red, green & blue) depending on which axis (x, y or z).

The lines have the same basic components as the cube:

```
var blue_line_material = new THREE.LineBasicMaterial( { color:
0x0000ff } );
var green_line_material = new THREE.LineBasicMaterial( { color:
0x00ff00 } );
var x_point = [];
var y_point = [];
var z_point = [];

x_point.push( new THREE.Vector3(-5, 0, 0 ) );
x_point.push( new THREE.Vector3( 5, 0, 0 ) );
y_point.push( new THREE.Vector3( 0, -5, 0 ) );
y_point.push( new THREE.Vector3( 0, 5, 0 ) );
z_point.push( new THREE.Vector3( 0, 0, -5) );
z_point.push( new THREE.Vector3( 0, 0, 5 ) );

var xline_geometry = new THREE.BufferGeometry().setFromPoints(
x_point );
var xline = new THREE.Line( xline_geometry, red_line_material );
var yline_geometry = new THREE.BufferGeometry().setFromPoints(
y_point );
var yline = new THREE.Line( yline_geometry, green_line_material );
var zline_geometry = new THREE.BufferGeometry().setFromPoints(
z_point );
var zline = new THREE.Line( zline_geometry, blue_line_material );

scene.add( xline );
scene.add( yline );
scene.add( zline );
```

geometry, material, and mesh, they just use different types. Originally the `GridHelper` (used to visualize the x-z plane) was being rendered after lines were drawn. This made it appear as if the coloured axis lines had disappeared – because it drew the grid on top of the x and z-axis. This problem was solved by ensuring that the coloured lines were added to the scene after the grid.

(See Figure 1).

3. Rotate the cube

For this requirement, a level of separation was created between the key presses and the rotation method by adding a Boolean toggle (each toggle is defined as a global).

```
if(rotatex == true){
    cube.rotateOnAxis(new THREE.Vector3(1,0,0), 0.01);
}if(rotatey == true){
    cube.rotateOnAxis(new THREE.Vector3(0,1,0), 0.01);
}if(rotatez == true){
    cube.rotateOnAxis(new THREE.Vector3(0,0,1), 0.01);
}
```

The cube would only complete the rotate method in the animate function when the toggle was true. Once the key was pressed the cube would continue to rotate relative to its current rotation. The animate function is called every frame to update how the cube looks in response to the transformation. As the frames update so quickly it gives the appearance of the cube rotating. The toggle would be swapped when either the x, y or z key was pressed corresponding to the different axes to rotate around.

```
case 88: //rotate around x axis
    rotatex =! rotatex;
    B_rotatex =! B_rotatex;
    break;
```

(See Figure 2).

4. Different render modes

Each render mode required making new objects of type: points and wireframe based on the geometry of the cube. These new objects were then added to the cube instead of the scene so that when transformations were applied to the cube they would also be applied to the wireframe and vertices.

```
var vertex_material = new THREE.PointsMaterial( { color:
0x0000ff, size:0.2 } );
vertex = new THREE.Points(cube_geometry,vertex_material);

var red_line_material_1 = new THREE.LineBasicMaterial( {
color: 0xff0000 } );
var wireFrame = new
THREE.WireframeGeometry(cube_geometry);
WireFrame_line = new THREE.LineSegments(wireFrame,
red_line_material_1);

cube.add(vertex);
cube.add(WireFrame_line);
```

The different render modes are toggleable so that all of them can be on at the same time – each mode is not exclusive. You can toggle each render mode on and off by pressing the letter keys: v-vertex mode, e-edge mode, and f-face mode. The toggle of the face render mode was implemented by setting the visible property of the cube to be true or false.

```
case 69: // e = edge and wire frame
    if( WireFrame_line.visible == true){
WireFrame_line.visible = false;}
    else{ WireFrame_line.visible = true;}
    break;
```

```
case 70: // f = face
    if(cube.material.visible ==
true){cube.material.visible = false;}
    else{ cube.material.visible = true;}
    break;
```

(See Figure 3).

5. Translate the camera

The camera movement was implemented in a similar way to the rotate functions of the cube, by using a toggle in the animate function for the up, down, left and right (as well as using the corresponding arrow keys). The movement was implemented in the same way the rotations were, in the sense that you don't have to hold down the key to keep the camera moving. Once the key has been pressed it will keep moving in that direction until you hit the same key again. A universal stop button was implemented to halt all camera movements and object rotations, the space bar was used for this.

```
if(left == true){camera.translateX(-0.01);}
if(right == true){camera.translateX(0.01);}
if(stop == true){
up = false; down = false; left = false; right = false;
stop = false; rotatex = false; rotatey = false; rotatez =
false;
```

The forward and backwards movement of the camera was a little more complicated to implement. The mouse scroller was used to control these camera movement. This required creating a new listener function to pick up on the movement of the scroll wheel.

```
document.addEventListener('wheel', zoom);
```

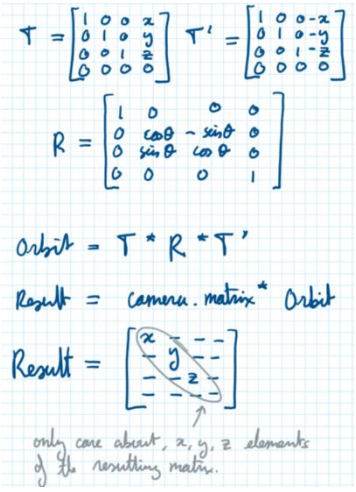
The delta of the event keeps track of the direction the mouse scroller has been moved relative to its previous position. The delta is then used to determine how far the camera should be moved in response to the scroll action.

```
function zoom(event){
        if(event.deltaY < 0){
            camera.translateZ(-0.4);
        }else{camera.translateZ(0.4);}
}
```

(See Figure 4).

6. Orbit camera

The camera orbit controls were implemented using homogeneous matrix transformations. The transformation for orbiting around a point (given as a parameter) was created by multiplying a 3D translation matrix (moving the object to the given point) with a 3D rotation matrix (where the angle ranges between 0 and 2 pi radians in 0.01pi radian increments) then a translation back to the starting position (a negation of the first translation matrix).



Three.js provides methods on the Matrix4 type that has been used to generate the translation and the rotation matrices (1). The resulting transformation matrix '`orbit`' was then multiplied with the matrix of the camera's position vector to make '`resultMatrix`'. Next, extracted the x and y elements from the resulting matrix (which is indexed in column-major order) to work out how far to translate the camera in the x or y direction (relative to its current position).

```
function cameraOrbit(counter, axis, point){
var x, y, z;
x = point[0]; y = point[1]; z = point[2];

var translation = new THREE.Matrix4().makeTranslation(x,
y, z);
var backwardsTransl = new
THREE.Matrix4().makeTranslation(-x, -y, -z);

if(axis == "x"){
    var rotation = new
THREE.Matrix4().makeRotationX(counter * Math.PI);
}
else if(axis == "y"){
    var rotation = new
THREE.Matrix4().makeRotationY(counter * Math.PI);
}

var transformation = rotation.multiply(backwardsTransl);
var orbit = translation.multiply(transformation); // =
T*R*Tn
var cameraMatrix = new THREE.Matrix4();
cameraMatrix = camera.matrix;
var resultMatrix = orbit.multiply(cameraMatrix);

var arrResults = resultMatrix.elements;
if(axis == "x"){camera.translateY(arrResults[5]);}
else if(axis == "y"){camera.translateX(arrResults[0]);};
camera.lookAt(x, y, z);
}
```

Then the camera had the `lookAt` method applied to force the camera to look back at the point it's orbiting around. Sadly, couldn't apply the matrix directly to the camera because three.js doesn't allow for that - probably because it wants to prevent general transformations e.g. to scale or shear the camera. Instead of writing a separate function for the horizontal and vertical orbit (along the x axis and along the y axis) the function was adapted so that the parameter would determine which type of rotation to execute.

The orbit function is called when the r (orbiting vertically in the z-y-plane) or t (orbiting horizontally in the x-z-plane) letter keys are pressed. The function uses global counters to keep track of how far the rotation is around the point. The counters stay within the range of the radians (because that's the angle measurement used for the rotation matrix). The counter gets updated every time the key is pressed.

```
case 82: // r = orbit vertically
    if(counter< 2){counter += 0.01;}
    else{counter = 0;}
    cameraOrbit(counter,"x",[0,0,0]);
    break;
```

If more time had been available, a solution to handle the edge case of the orbit controls could have been included. When completing the vertical orbit around an object and the camera reaches the top of the object and the camera flips perspective by 180 degrees. Essentially the top of the camera is always facing up and when orbiting down the other side of the object it flips itself to never become upside

down. This is likely to be caused by the `lookAt` function and the direction the camera determines as up. A possible solution to this problem is restricting the range of the radian angles then altering the camera's up direction at the edge cases of the counter range.

(See Figure 5)

7. Texture mapping

To complete the texture mapping requirement, 6 different images were applied to texture the cube, one for each face. They were loaded using the `TextureLoader()`.

```
var objloader = new THREE.OBJLoader();
objloader.load('bunny-5000.obj', bunnyLoader);
```

Each of the different images were loaded as `MeshPhongMaterial` into an array.

```
var loader = new THREE.TextureLoader();
var material =
[
    new THREE.MeshPhongMaterial({map: loader.load
('lego1.jpg')}),
    new THREE.MeshPhongMaterial({map: loader.load
('lego2.jpg')}),
    new THREE.MeshPhongMaterial({map: loader.load
('lego3.jpg')}),
    new THREE.MeshPhongMaterial({map: loader.load
('lego4.jpg')}),
    new THREE.MeshPhongMaterial({map: loader.load
('lego5.jpg')}),
    new THREE.MeshPhongMaterial({map: loader.load
('lego6.jpg')})
]
```

Which were then used as the material for the cube's mesh. (3). Could have simply used the MeshBasicMaterial instead of the MeshPhongMaterial. But decide to experiment with the Phong material to see how the lighting would affect the appearance of the cube (2). As expected, the new material gave the cube smooth highlights and shadows where the `DirectionalLight` hit the cube. The Phong material interpolates the colour from the normals of all the points on the surface, not just the normal of the faces or vertices.

```
case 70: // f = face
    if(cube.material.visible == true){
        for(var i = 0; i<6; i++){cube.material[i].visible
= false;}
        cube.material.visible = false;
    }
```

When toggling the textures on and off, each face of the material had to be individually referenced by index to turn each of them off – couldn't just set the cube.material.visible = false.

(See Figure 6).

8. Load a mesh model from .obj

For the mesh loading requirement, the mesh file of the Stanford bunny was used. The file contains a list of all the vertices of the triangle mesh defining the bunny, which is read in by the three.js loader function. The load method used has a callback function argument which was filled with my own `bunnyLoader` function.

The separate bunny loader method that was developed runs a traversal throughout the file and creates a mesh from each of the vertices it reads to build the thousands of individual triangular faces making up the surface. In the traversal, it also applies the Phong material to the mesh so that when the light hits the material the object won't look flat and you can distinguish between the different depths based on the shadows and highlights.

```
function bunnyLoader(BunnyObject){
    var phong = new THREE.MeshPhongMaterial( { color:
0x6a5acd, specular: 0x555555, shininess: 30 } );

    BunnyObject.traverse(function (obj){
        if(obj instanceof THREE.Mesh){
        obj.material = phong;
        bunny = obj;
        }
    })

    var bunny_geometry = bunny.geometry;

    var bunny_vertex_material = new THREE.PointsMaterial(
{ color: 0x0000ff, size:0.1 } );
    bunny_vertex = new
THREE.Points(bunny_geometry,bunny_vertex_material);

    var bunny_line_material = new
THREE.MeshBasicMaterial({color: 'magenta', wireframe:
true});
    var bunny_wireFrame = new
THREE.WireframeGeometry(bunny_geometry);
    bunny_WireFrame_line = new
THREE.LineSegments(bunny_wireFrame, bunny_line_material);
    bunny_WireFrame_line.visible = true;

    bunny.add(bunny_vertex);
    bunny.add(bunny_WireFrame_line);
    bunny.position.x = -5;
    scene.add(bunny);
}
```

The loader function is also responsible for creating the vertex mesh and the wireframe of the bunny inside the loader. (The wireframe and vertices are created in the same way as the cube.) This had to be done inside of the loader because of the order that the callback functions are executed. As the bunnyLoader is a callback function, it was placed at the bottom of the call stack and would be the last function to be executed. Therefore, the bunny will not be defined until the last function is executed. Consequently, the bunny can only be accessed once all the other functions have completed running, which means it can be accessed in the key handler interruption – cannot trigger the key interruptions until all of the initial functions have already been completed.

(See Figure 7).

The function to scale the bunny uniformly to fit perfectly inside the cube can be triggered by pressing the b key (b for box). The scale factor is determined by the current size of the bunny and the size of the cube. The size of the bunny is calculated by creating a box around the bunny object and then using the size of that box. To work out the bunny's scale factor, the max value of all the dimensions of the box was used. In the end, all of the dimension of the bunny have to be scaled by the same value to ensure a uniform scaling (if scaled by the x, y and z dimensions of bunny the individually then it couldn't be guaranteed that the bunny

would be scaled uniformly – but it would fit into the given shape perfectly). The final scale factor is equal to the size of the cube that the bunny is being fitted into divided by the bunny size (the max size of the cube surrounding the bunny).

```
function bunnyScale(originalBunPos){
    var box = new THREE.Box3().setFromObject( bunny );
    var dimensions = new THREE.Vector3();
    box.getSize(dimensions);
    var scaleVal = Math.max(dimensions.x, dimensions.y,
dimensions.z);
    var x = cubeSize/scaleVal;

    bunny.geometry.center();
    bunny_WireFrame_line.geometry.center();
    bunny.position.x = 0;
    bunny.position.y = 0;
    bunny.position.z = 0;

    bunny.scale.set(x,x,x);
}
```

The centre of the bunny needs to be reset to have the same centre as the cube, for the whole of the bunny to be able to fit inside of the cube. This is because the centre of the Stanford bunny mesh is not the geometrical centre of the object, so even though the bunny will be the correct size to fit inside of the cube, its bottom was sticking out because the 'centre' was off on the x axis.

(See Figure 8).

9. Rotate the mesh, render it in different modes

The rotation and different render modes were implemented in the same way as the cube, just applied to the bunny object instead of to the cube. Used the same letter keys to trigger each of the modes. Just had to create a collection of new global variables to be used as toggles which could then be used by the same methods as the cube.

(See Figure 9, Figure 10, and Figure 11).

10. Creative!

I. Intro

For the creative requirement I chose to experiment to see if I could simulate the look of real bouncing balls. The experiment started with copying some code from a stack overflow post on how to implement a very basic infinite bouncing ball (4). I then proceeded to completely change the copied code to make my own new program. The changes were made in steps, that I shall illustrate in my report.

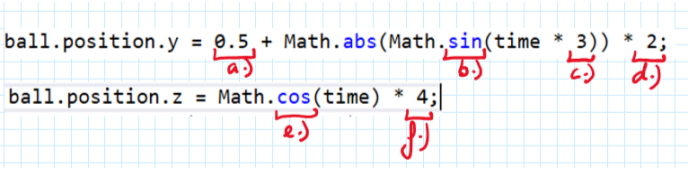
II. Making the ball rotate in response to the direction travelling

The very initial code that was implemented, had the ball bouncing back and forth on a plane following a cosine curve. However regardless of the direction the ball was travelling, it always had the same rotation direction. So, the first addition I made to the code was: to have the ball toggle which direction the ball was rotating based on the direction it was travelling in. This helped make the ball bouncing look much more realistic.

```
function rotateAndBounce(){
        requestAnimationFrame(rotateAndBounce);
        delta = clock.getDelta();
        time += delta;
        ball.position.y = 0.5 + Math.abs(Math.sin(time *
3)) * 2;
        ball.position.z = Math.cos(time) * 4;

        if(ball.position.z > 3.9){
                toggle =! toggle;
        }
        else if(ball.position.z < -3.9){
                toggle =! toggle;
        }
        if(toggle){ball.rotation.x = time * -4;}
        else{ball.rotation.x = time * 4;}

        renderer.render(scene, camera);

}
```

III. Experimented with the bounce patterns

Next, I was intrigued to see how altering the different parts of ball's transformations would affect the animation of the ball. The existing code was copied to create a new ball and new bounce animation function. Then I dissected all the variable parts (underlined in red) of the lines of code which determined the position of the ball in each frame and this is what I discovered:



a) Determines how deep the ball bounces into the plane: 1 = ball hovers over the plane. 0 = bounces through the plane. For the ball to bounce on top of the plane this variable needs to be set to the radius of the sphere
b) Determines which curve the ball bounce follows through. Sin = start the ball on the plane. Cos = starts the ball above the plane.
c) Determines the number of bounces/times it hits the plane on its way back or forth.
d) How high it bounces – the higher the number the higher the bounce and 0 means no bounces, just rolls.
e) Determines the starting direction – is the ball going left or right to begin with.
f) The distance the ball bounces for: 4 is the width of the plane so will bounce across the whole distance of the plane. If greater than 4 the ball will bounce off the sides of the plane. If smaller than 4 will bounce inside the width of the plane.

IV. Exponential bounce

Next, I thought that the infinite bouncing did not look very realistic so decide to implement an exponential bounce that would slow down as the time passed.

This was implemented by using a new variable that would decrease the height of the bounce each time. For the exponential equation I chose the variable to be to the power

of a half because after trialing other fractional values this gave the most satisfying reduction in bounce.

```
var height = Math.pow(0.5,counter);
counter = counter + 0.008;
ball2.position.y = 0.5 + Math.abs(Math.sin(time * 5)) *
height;
```

Next, I worked on trying to implement an exponential bounce travelling from left to right across the screen (rather than bouncing in the same spot). This was implemented very rather simplistically by having the ball move across the scene based on time. To add to the illusion of bouncing movement, the rotation of the ball was added back in. The rotation speed reduced the further it moved towards the left side of the plane.

```
if(4 - time > -4){ball.position.z = 4 - time;}
var spin = height*2;
if(spin>4){spin = 4;}
if(ball.position.z > 3){
        spin = Math.abs(spin - 1);
        spin = 0;
}else{
        ball.rotation.x = time * -4 * spin;
}
```

If I had more time, I would have the ball move slower across the plane as the bounces become shorter and I would have made the ball's rotation stop completely as it hit the left-hand side of the plane.

V. Added lighting, shadows and textures

Once the animation was completed, I moved on trying to make the balls and scene looking more realistic. Firstly, the Wireframes were replaced with solid, coloured materials using the `MeshPhongMaterial`.

```
var light = new THREE.DirectionalLight(0xffffff,1.5);
light.position.set( 1, 1, 1 ).normalize();
light.castShadow = true;
scene.add(light);
```

Then shadows were applied to the objects in the scene. (5) After reading this article, I then added a directional light to the scene and set its `castShadow` property to true. Then for shadows to become visible the object's `castShadow` and `receiveShadow` properties also had to be set to true.

Once the shadows were working, then applied textures to both balls and the plane using the same method as earlier in the coursework. (when applying it to the cube).

```
ball = new THREE.Mesh(new THREE.SphereGeometry(0.5, 16,
8),
new THREE.MeshPhongMaterial({
        map: loader.load ('orange.jpg'),
        shininess: 70,
        specular: 0x555555
}));
```

VI. Added a skybox

Afterwards, I created a sky box to add a background to all direction so the scene. (6)

This was implements by creating a very large box object where the texture was then applied to the inside (instead of the outside) of the box. The plane and balls then fitted within the 'skybox' that was created.

```
var skyBox_geometry = new THREE.BoxGeometry(200, 100,
200);
var skyBox_texture = new THREE.MeshBasicMaterial({
        map: loader.load ('wood.jpg'),
});
skyBox_texture.side = THREE.BackSide;
```

VII. Transparent Ball

Finally, I wanted to experiment further with object materials and made one of the balls transparent. (7.) This was surprisingly simple to implement and only required changing the material properties: `transparent` and `opacity`.

```
var material = new THREE.MeshPhongMaterial({
        map: loader.load ('apple.jpg'),
        //color: 0x0005ff,
                transparent: true,
                opacity: 0.6,
        shininess: 20
});
```

The range of opacity was between 0 and 1 where 0 = completely transparent and 1 = completely opaque.

(See Figure 12).

11. References

1. Cabello R. three.js docs [Internet]. Threejs.org. 2020 [cited 6 December 2020]. Available from: https://threejs.org/docs/#api/en/math/Matrix4
2. Cabello R. three.js docs [Internet]. Threejs.org. 2020 [cited 9 December 2020]. Available from: https://threejs.org/docs/#api/en/materials/MeshPhongMaterial
3. Three.js Textures [Internet]. Threejsfundamentals.org. 2020 [cited 29 November 2020]. Available from: https://threejsfundamentals.org/threejs/lessons/threejs-textures.html
4. 4. How to create simple vertical bounce animation to a sphere [Internet]. Stack Overflow. 2020 [cited 12 December 2020]. Available from: https://stackoverflow.com/questions/51429502/how-to-create-simple-vertical-bounce-animation-to-a-sphere
5. Cabello R. three.js docs [Internet]. Threejs.org. 2020 [cited 13 December 2020]. Available from: https://threejs.org/docs/#api/en/lights/shadows/DirectionalLightShadow.
6. Three.js Backgrounds and Skyboxes [Internet]. Threejsfundamentals.org. 2020 [cited 18 December 2020]. Available from: https://threejsfundamentals.org/threejs/lessons/threejs-backgrounds.html
7. Three.js Transparency [Internet]. Threejsfundamentals.org. 2020 [cited 18 December 2020]. Available from: https://threejsfundamentals.org/threejs/lessons/threejs-transparency.html

12. Figures

Figure 1 – simple cube at the origin with dimensions 2x2

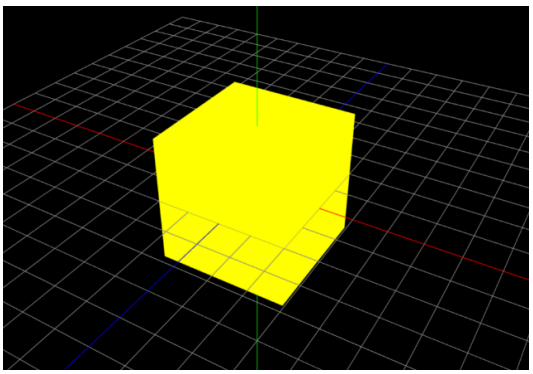


Figure 2 – cube rotated on the x, y and z axis

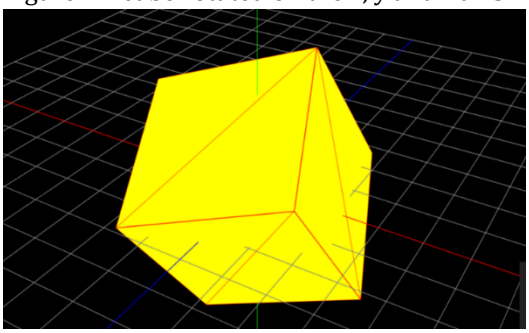


Figure 3 – cube rendered in vertex and edge mode

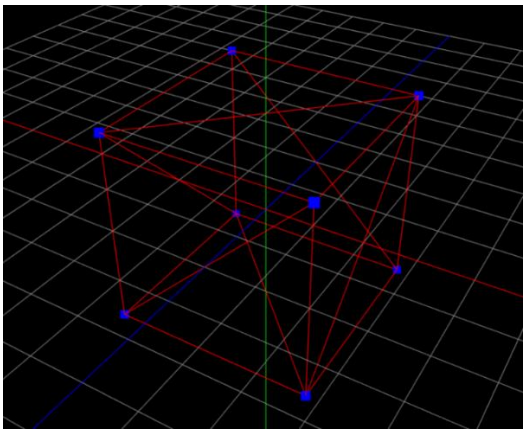
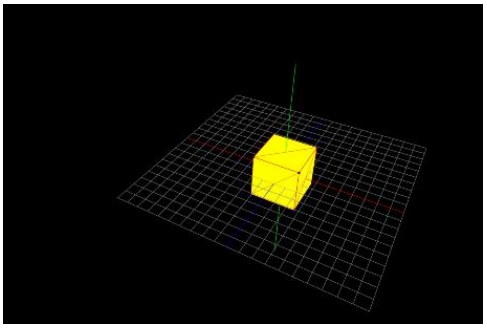


Figure 4 – different camera positions: up, left and zoomed out



Camera positions : right, down and zoomed in.

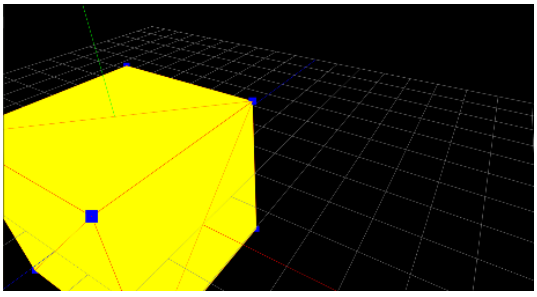
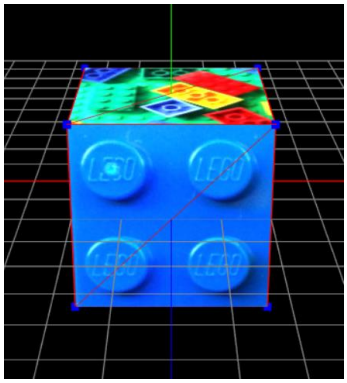
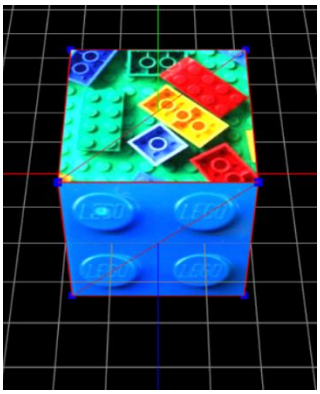
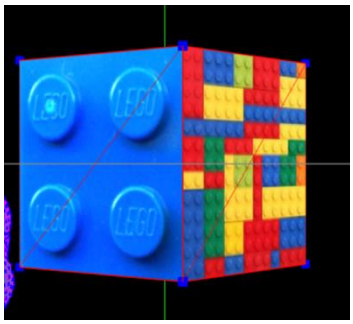


Figure 5 – orbiting vertically up and over the cube





Orbiting horizontally around the cube



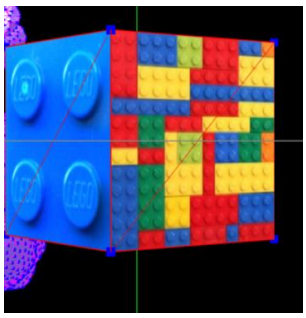
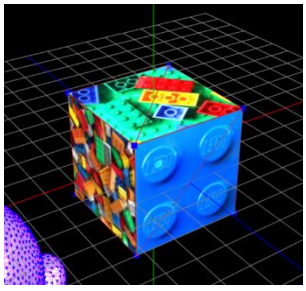


Figure 6 – applying 6 different images to each face



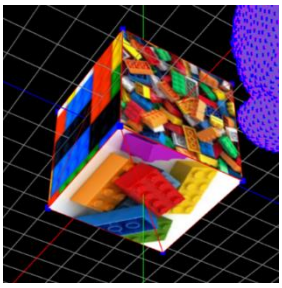


Figure 7 – Stanford bunny mesh with (Phong material)

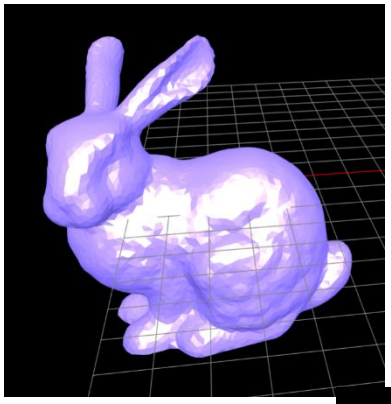


Figure 8 – bunny uniformly transformed to fit inside of the cube (with face mode off so that you can see the bunny inside).

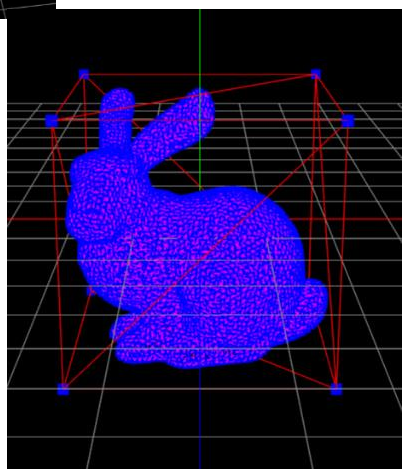


Figure 9 – Bunny in rendered in vector mode

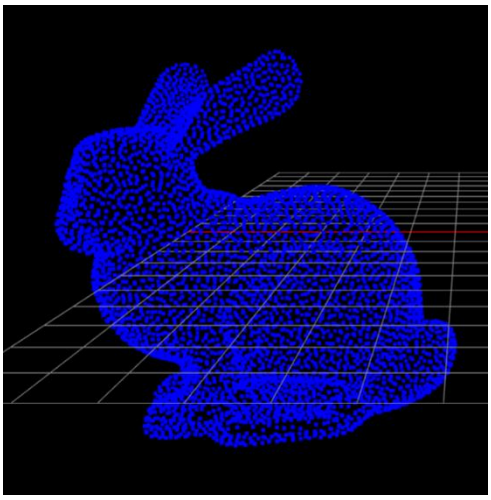


Figure 10 - Bunny in rendered in edge mode

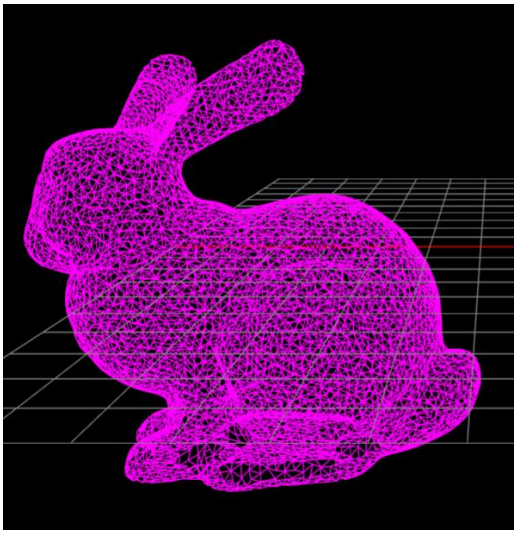


Figure 11 – bunny rotated in the x, y and z axis

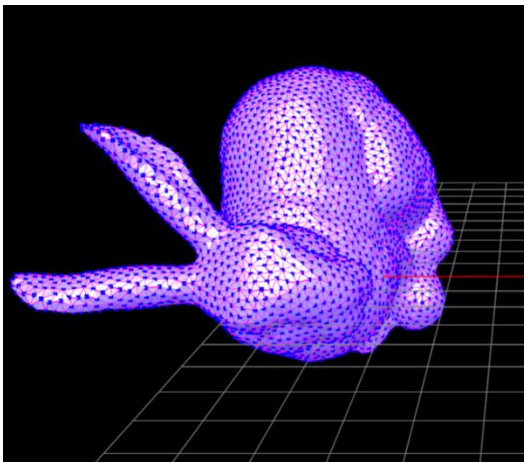


Figure 12 – the balls bouncing past one another