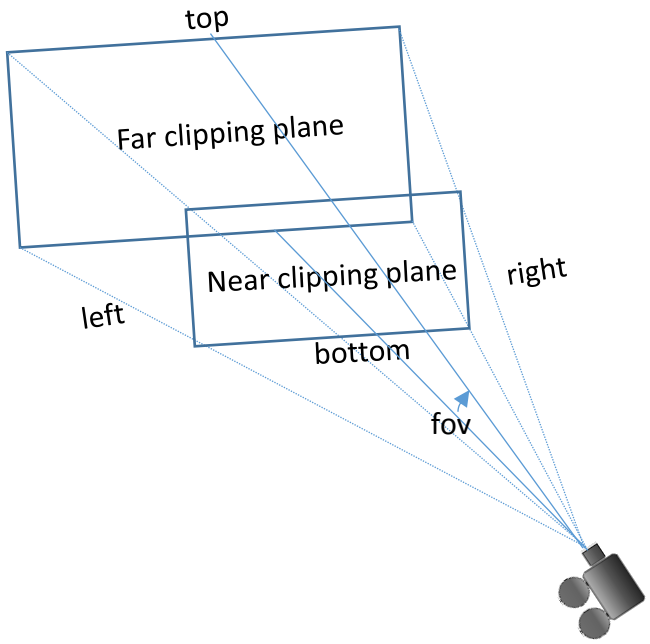
**Using Three.js: Hints and Programming Notes**

This document provides some tips, hints and examples that may help you understand the Three.js demo examples and move forward with your programming project related to Three.js.

The Three.js reference documentation and API is an excellent resource for additional examples and starter code. This reference is available at this URL:

https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene

The following diagram should be used as a reference for understanding the 3D geometry. If you are having trouble viewing your objects, more than likely, the parameters you are using need to be tweaked to allow the objects to be viewed.




Note: you can and should experiment with all of the parameters and objects noted below in the modeling-starter.html file provided in the Three.js demos.

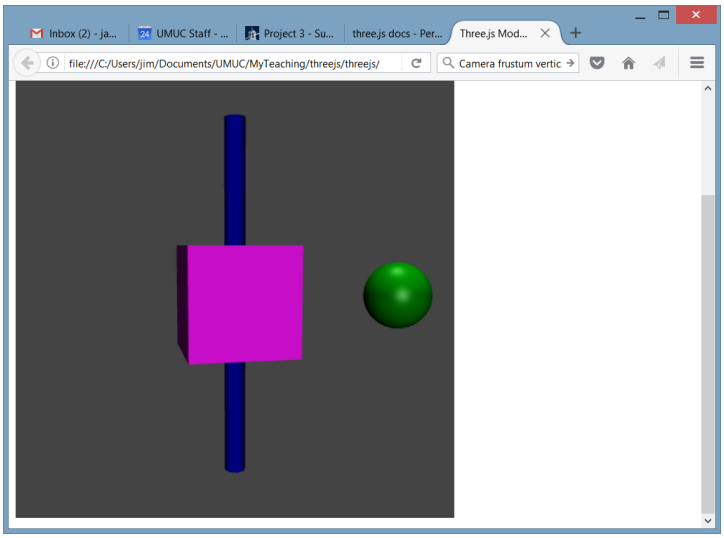
Key techniques for successfully using Three.js include:

1. Controlling the Viewing Frustrum with THREE.PerspectiveCamera(fov, aspect, near, far);

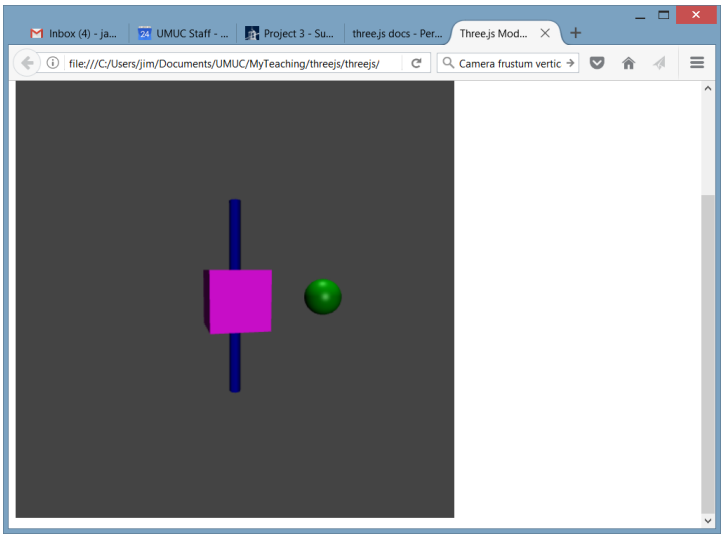
   This method has 4 parameters including fov, aspect, near and far as described below.

**fov** = Camera frustum vertical field of view. This is the angle subtended by the lines converging from the top and bottom centers of the screen. Increasing the fov has the perceived results of moving further away from the scene whereas decreasing the fov results in the scene appearing closer.

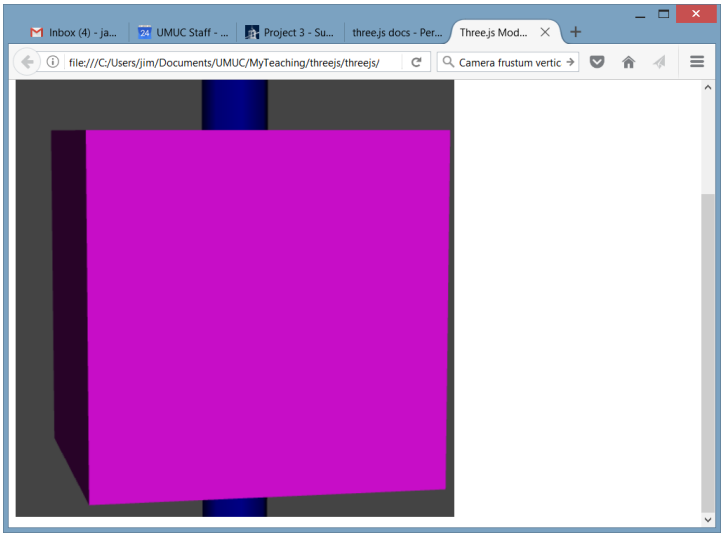
Using the modeling-start.html file, here is a screen capture when the FOV is set at 45 degrees.




Here is a screen capture with the fov is set to 75 degrees.




Finally, here is a screen capture with the fov is set to 15 degrees.



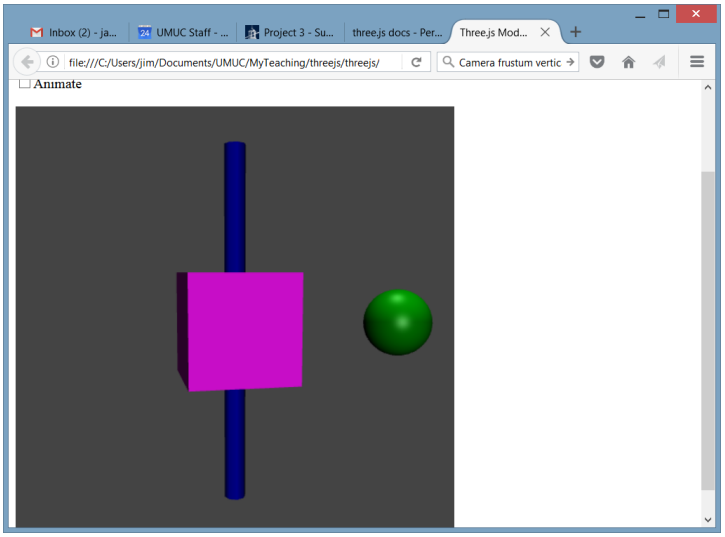

**aspect** - Camera frustum aspect ratio. This is usually the canvas width / canvas height. Note

The canvas width and height are set in the html portion of the code:

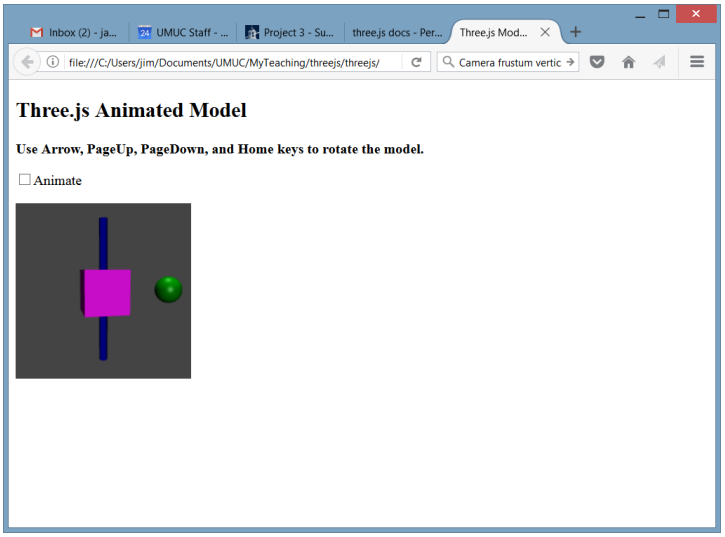
<canvas id="glcanvas" width="300" height="300"></canvas>. You should experiment by changing

The width and height.

Here is a screen capture of the modeling-starter.html file when the height and width are set to 500.




Here is a screen capture of the same html file when the height and width are set to 200.




**Near** - Camera frustum near plane. To be visible (rendered in the scene), an object must be between the near and far planes. The valid range is greater than 0 and less than the current value of the far plane.

**Far** - Camera frustum far plane. To be visible (rendered in the scene), an object must be between the near and far planes. The valid range is between the current value of the near plane and infinity.

Decreasing the Far plane may impact the scene. For example, changing the Far plane from 30 to 15 in the modeling-starter.html results in the sphere disappearing as it begins to rotate in the back plane. Experiment with changing these values and note their effect.

camera = new THREE.PerspectiveCamera(45, canvas.width/canvas.height, 1, 30);

camera = new THREE.PerspectiveCamera(45, canvas.width/canvas.height, 1, 15);

2. Adding Objects to your scene

Adding objects to your scene involves constructing the objects, positioning, then adding them to your scene. Often, you create a Object3D model, add objects to the model and then finally add the model to the scene.

Overall, these steps are part of a separate function (e.g. createWorld() ) within the JavaScript section of your code. The steps include:

a. Create the scene – Use the THREE.Scene() function to create a scene as follows

   scene = new THREE.Scene();

   A scene is required and is where you add your objects.

b. Create the camera – The camera sets-up the view for the scene. A camera is not part of the scene. It is just an observer. The parameters of the camera were described in the previous section.

Here is an example call with the fov set to 45 degrees, the aspect set to the ratio of the width and height of the canvas, the near field set to 1, and far plane set to 30.

```
camera = new THREE.PerspectiveCamera(45, canvas.width/canvas.height, 1, 30);
```

c. Create lights and add to the scene– Lights add life and color to your scene. Multiple lighting types exist including Directional. The following code adds a direction light with an intensity of 0.5 to the scene.

```
scene.add( new THREE.DirectionalLight( 0xffffff, 0.5 ) );
```

This light can cast shadows as the direction is calculated as pointing from the light's position to the target's position.

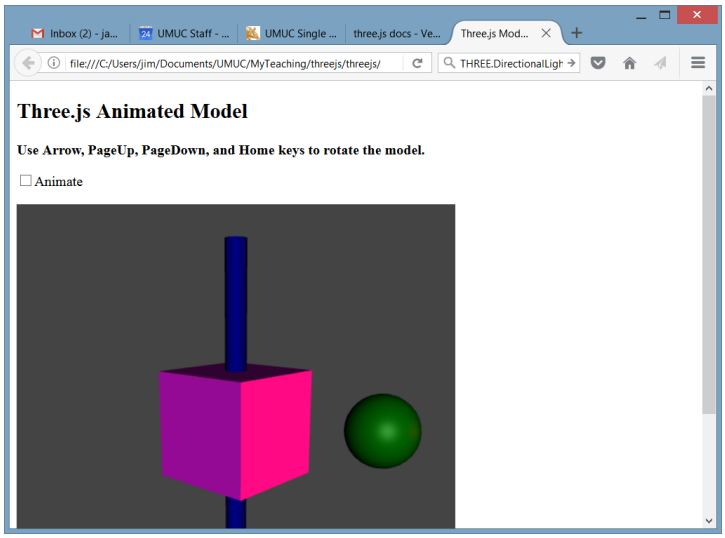
The hexadecimal values represents the color of the light. The float value represents the intensity. Intensity values ranges from 0 to 1 with 1 being the brightest. The color is Hexadecimal RGB (Red, Blue, Green) with 0xff0000 being red, 0x00f00 as green and 0X0000ff as blue. Hexadecimal color look-up values are available online for those looking for more interesting sets of color combinations.

The default light position is (0,1,0) which is looking down the Y-axis.

To change this you can construct the light and the set the light position:

```
var dlight = new THREE.DirectionalLight(0xff0000, 0.7);

dlight.position.set(1,0,0); // Look down the x-axis

scene.add(dlight);
```

This results in the direction looking down the X-axis as shown in this screen capture.



Note the above graphic was generated by using up/down/left/right arrows to allow the x-axis to be slightly turned towards the view to demonstrate the incoming red directional light source.

d. Create the model – The Object3D is used to create a model. Using this object provides a set of properties and methods for manipulating all objects in 3D space.

For example, after creating an Object3D model, methods for translating, rotating and scaling can be called that impact all objects within the model.

First, we create the model

model = new THREE.Object3D();

Then, after we have created objects and added them to the model (see step e below), we can use the available methods to manipulate and transform all objects in the model.

The following code rotates the model by 0.2 x and 0.2 y radians.

model.rotation.set(0.2,0.2,0);

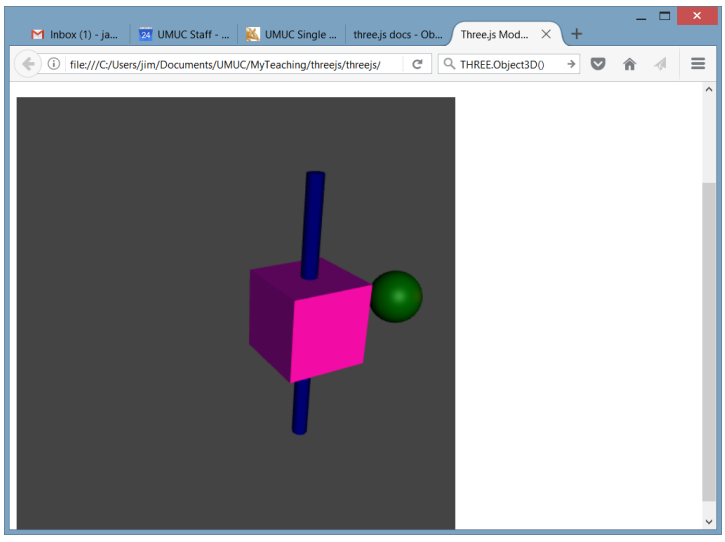
In addition, the following code positions (translates) the objects by 2 units in the x direction.

model.position.set(2,0,0);

Finally, the following code scales all objects in the model by to 80% of the original size in x,y and z.

model.scale.set(0.8,0.8,0.8);

Running each of the sets of transformations on the modeling-starter.html would result in the following output.




e. Create objects and add them to the model.

Creating objects involved constructing your specific objects you want displayed in the scene. Often, you use the existing geometric shapes such as the boxes, cylinders, spheres, torus and other shapes available to you.

Here are a couple object creation examples to consider:
To create a green cube at position (1,1,4) and add it to the model

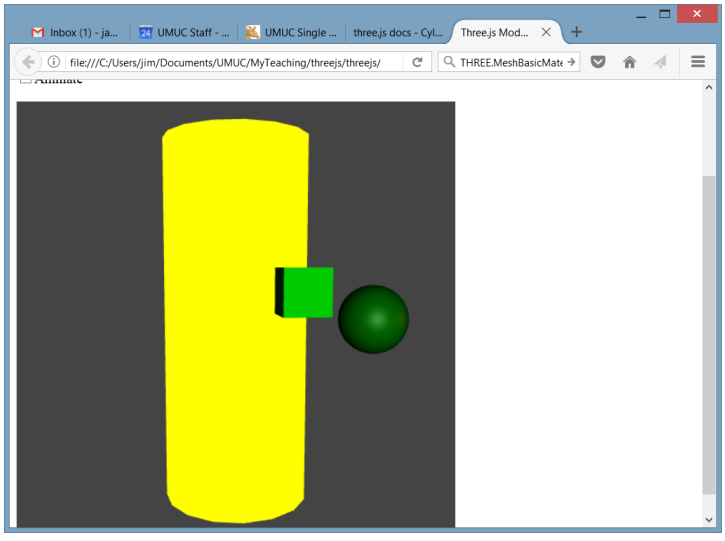
```
var geo1 = new THREE.BoxGeometry( 1, 1, 1 );
var mat1 = new THREE.MeshLambertMaterial( {color: 0x00ff00} );
var cube1 = new THREE.Mesh( geo1, mat1 );
cube1.position.set(1,1,4);

model.add( cube1 );
```

You should experiment with the many materials available for you. MeshLambertMaterials are affected by light sources where as MeshBasicMaterials are not.

To create a yellow cylinder and it to the model:

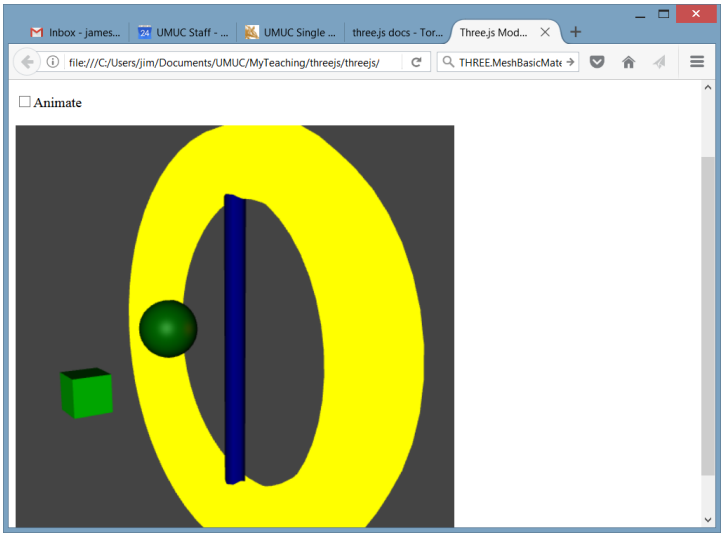
```
var geometry = new THREE.CylinderGeometry( 2, 2, 10, 16 );
var material = new THREE.MeshBasicMaterial( {color: 0xffff00} );
var cylinder = new THREE.Mesh( geometry, material );
model.add( cylinder );
```

Adding this code to the previous results will render a yellow cylinder as shown in the following screen capture:




To add a yellow torus to the model:

```
var geometry = new THREE.TorusGeometry( 5, 1, 8, 50 );
var material = new THREE.MeshBasicMaterial( { color: 0xffff00 } );
var torus = new THREE.Mesh( geometry, material );
model.add( torus );
```




Note the torus screen capture was taken after the up/down/left/right arrow keys were used to better show the torus structure.

f. Add the model to the scene

Once the objects have been created and added to the model, you add the model to the scene with this line of code:

scene.add(model);

3. Rendering and additional methods

The createWorld() method contains most of the code associated with building the objects and associated scene. This section provides brief descriptions of the remaining methods and structure in the modeling-starter.html file

a. HTML and JavaScript

JavaScript is part of the HTML file. A template for HTML is shown below:

```
<!DOCTYPE html>
<html>
  <head>
      <meta charset=utf-8>
          <title>My first three.js app</title>
           <script src="js/three.js"></script>
            <script>
                 // Our Javascript will go here.
            </script>
    </head>
    <body>
    </body>
</html>
```

Note that all of the JavaScript code you write will go between the <script> and </script> tags. Inside of the body tags will include all the HTML specific code such as paragraphs, headers, forms and other special tags.

b. Javascript Init()

The init() method is called when the html file is first loaded into the browser. It includes creating the renderer, making sure WebGL is available, checking the values of the check box, and calling the createWorld() and render() methods.

Although there isn't much code in the init() method, the code is quite powerful. For example, the reason why we can rotate via the up/down/right/left keys is because the calls associated with doKey method. Each time a key is pressed, an event is triggered that captures which key was pressed and performs specific functionality as listed in the doKey() method.

In addition, an onChange handler is set-up such that at any point in time if the animate check box value is changed, the doAnimateCheckBox() method is called.

Most of this code can be copied as is, but if you want to add additional radio buttons or functionality, more lines of code would need to be added to handle those events.

c. doKey()

The doKey() method is called anytime a keyboard key is pressed by the user. Based on the key selected, a switch statement is used to take appropriate action. Since we are using the Model3d, we can easily use the rotate (or other methods for scale, position) to change the all objects in the model.

For example, if the user selects the right arrow, the model will rotate 0.03 radians for each key stroke with this line of code:

```
case 39: model.rotation.y +=  0.03; break;    // right arrow
```

d. doFrame() and doAnimateCheckbox() methods

These methods work side-by-side to update the frames based on if the animating checkbox has been selected. Each calls render() or updateForFrame() as appropriate.

e. render()

The render() method calls the render method based on the scene and camera setting

f. updateForFrame()

The updateForFrame() method is where you would place all of your specific object transformation calls. For example, the following would update the sphere object by rotating about the y-axis 0.03 radians from the current value.

```
sphereRotator.rotation.y += 0.03;
```

Increasing the radians would have the net impact of increasing the speed of the rotation about the y-axis.

g. requestAnimationFrame()

This is a method that is part of the JavaScript library that we didn’t have to write. This method helps smooth the animation. This essentially assures animation code is called when the user's computer is ready to make changes to the screen.

**Summary**

This walk-through hopefully has helped you understand the Three.js modeling-starter.html code as well as provide a pathway to completing your unique project using the Three.js libraries. It is highly recommended to take the modeling-starter.html code and experiment with each line of code to better understand the functionality and limitations of the calls. In addition, studying the documentation available at the Three.js website allow for the use of additional objects, textures, colors and shapes.