CMPE 360

Fall 2023

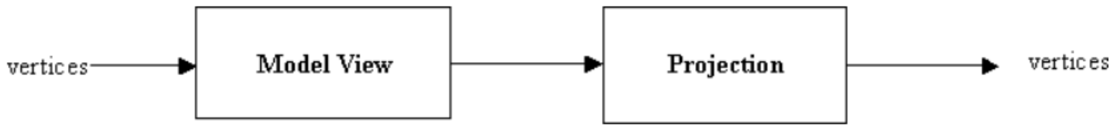
Project7

WebGL_2

This project is due by 23:59 on Sunday, 10 December 2023

HELPING NOTES FOR PROJECT

The classic OpenGL pipeline had two main stages of vertex transformation, each with its own transformation matrix. These were built into the graphics hardware. These days, other transformation pipelines have become possible since transformations are done in the vertex shader. However, in this assignment, we will try to implement the classic pipeline.



Each vertex in the scene passes through two main stages of transformations:

- *Model view transformation* (translation, rotation, and scaling of objects, 3D viewing transformation)
- *Projection* (perspective or orthographic)

There is one global matrix internally for each of the two stage above:

- $M_{modelview}$
- $M_{projection}$

Given a 3D vertex of a polygon, $P = [x, y, z, 1]^T$, in homogeneous coordinates, applying the model view transformation matrix to it will yield a vertex in eye relative coordinates:

$$P' = [x', y', z', 1]^T = M_{\text{modelview}} * P.$$

By applying projection to P’, a 2D coordinate in homogeneous form is produced:

$$P'' = [x'', y'', 1]^T = M_{\text{projection}} * P'.$$

The final coordinate [x”, y”] is in a normalized coordinate form and can be easily mapped to a location on the screen to be drawn.

Setting Up The Modelview and Projection Matrices in your shader

Since OpenGL Core Profile and **WebGL** always **use** shaders, neither the modelview nor the projection matrix is available. You have to set them up yourself. The matrices will be allocated and given their values in the main program, and they will be applied to vertices in the shader program.

To help us create and manipulate matrices in our main program we will use the matrix classes and helper functions in mat.h . Each matrix will be initialized to identity if you use the default constructor. So to create our initial modelview and projection matrices we would declare two mat4 objects like so:

```
var mv = new mat4();   // create a modelview matrix and set it to the identity matrix.
var p = new mat4();    // create a projection matrix and set it to the identity matrix.
```

These two matrices can be modified either by assigning or post-multiplying transformation matrices on to them like this:

```
p  = perspective(45.0f, aspect, 0.1f, 10.0f); // Set the projection matrix to
                                              // a perspective transformation

mv = mult( mv, rotateY(45) ); // Rotate the modelview matrix by 45 degrees
                              // around the Y axis.
```

As in this example, we will usually set the projection matrix p by assignment, and accumulate transformations in the modelview matrix mv by post multiplying.

You will use uniforms to send your transformations to the vertex shader and apply them to incoming vertices. Last assignment you did this for colours by making vector type uniforms and for point sizes by making a float uniform. Uniforms can also be matrices.

```
//other declarations
//...

//Uniform declarations
uniform mat4 mv; //declare modelview matrix in shader
uniform mat4 p;  //declare projection matrix in shader

void main()
{
 //other shader code
 //...

 //apply transformations to incoming points (vPosition)
 gl_Position = p * mv * vPosition;

 //other shader code
 //...
}
```

To set the value of uniform shader variables you must first request their location like this:

```
//Global matrix variables
var projLoc;
var mvLoc;


//In your init code
// Get location of projection matrix in shader
projLoc = gl.getUniformLocation(program, "p");

// Get location of modelview matrix in shader
mvLoc = gl.getUniformLocation(program, "mv");
```

Then, you use a uniformMatrix* function with the uniform location and a local variable to set their value. Do this whenever you need to update a matrix - usually when the window is resized or right before you draw something. To set the value of our 4x4 float type matrices we will use the form uniformMatrix4fv:

```
//in display routine, after applying transformations to mv
//and before drawing a new object:
gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv))); // copy mv to
uniform value in shader

//after calculating a new projection matrix
//or as needed to achieve special effects
gl.uniformMatrix4fv(projLoc, gl.FALSE, flatten(transpose(p))); // copy p to
uniform value in shader
```

Important: *Notice that we use the flatten() and transpose() functions from MVnew.js. You are probably used to flatten() by now. The transpose() function is necessary because most CPU oriented languages expect matrices to be in* ***Row Major*** *order, but your GPU and GLSL expect matrices to be in* ***Column Major****. Some Javascript math matrix libraries are written to match GLSL, but MVnew.js is not. In a full featured OpenGL, the second argument can be set to true to ask for a transpose, but the OpenGL ES and WebGL standards require that argument to be false.*

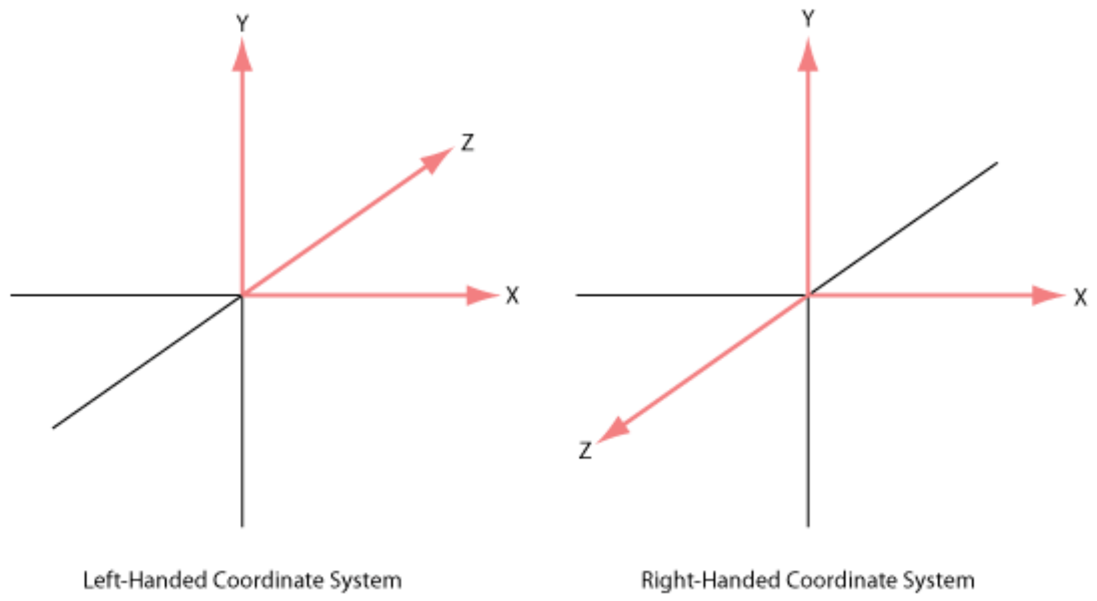
B. Elementary Transformations

This section covers the 3D transformation matrix functions provided with your textbook. There are alternative libraries that come with their own in-depth discussion of similar transformations, such as Mozilla's Matrix math for the web article.

Defining a Coordinate System – Which Way Does Z-point?

- ***Right-handed*** and ***left-handed coordinate system***: With your right hand line your first two fingers up with the positive y axis and line your thumb up with the positive x axis.

When you bend your remaining two fingers, the direction they point is the positive z axis in a right handed coordinate system. Compare to the figure below. The other system shown is a left-handed coordinate system. It is sometimes used in graphics texts. A consequence of using the right-handed system is that the negative z-axis goes into the screen instead of the positive as you might expect.



- Right-handed coordinate system is used most often. In OpenGL, both the local coordinate system for object models (such as cube, sphere), and the camera coordinate system use a right-handed system.
- In the following discussion, we assume that all transformation function calls return a matrix that you will post-multiply onto $M_{modelview}$, unless the other is specifically mentioned.
- All transformation functions in this discussion that do not begin with gl. are equivalent or similar to a classic OpenGL transformation function and are defined in MVnew.js. They all use the float data type for simple values.

Translation:

translate(dx, dy, dz);

Where [**dx, dy, dz**] is the translation vector.

The effect of calling this function is to create the translation matrix defined by the parameters [dx, dy, dz] which you should concatenate to the global model view matrix:

$M_{modelview} = M_{modelview} * T(dx, dy, dz);$

Where *T(dx, dy, dz)* = $\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$

In general, a new transformation matrix is always concatenated to the global matrix from the right. This is often called **post-multiplication**.

```
mv = mult( mv, translate(0,0,-6) ); //Translate by -6 units on z-axis
```

Rotation:

There are two forms of rotation in MVnew.js.

rotate(angle, vec3(x, y, z));

The first is similar to the only one available in Classic OpenGL. It is capable of rotating by angle degrees about an arbitrary vector. However, it is often easier to rotate about only one of the major axes:

- the x-axis: vec3(1,0,0)
- the y-axis: vec3(0,1,0)
- the z-axis: vec3(0,0,1)

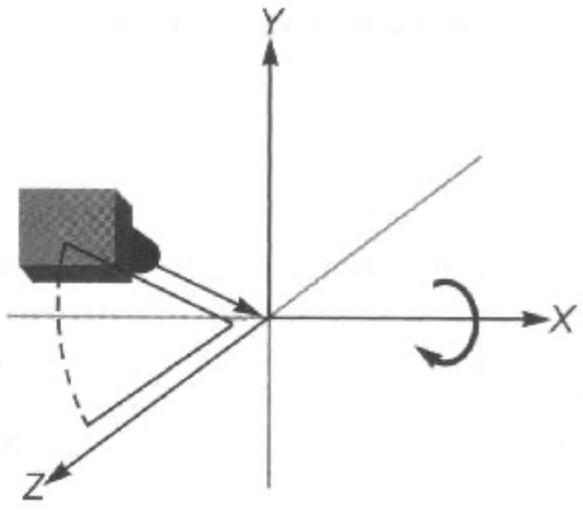
These simple rotations are then concatenated to produce the arbitrary rotation desired. For example:

```
mv = mult( mv, rotate(20,vec3(0,1,0)) ); // Rotate 20 degrees CCW around Y axis
```

Rotating around only one axis at a time is so common that many matrix libraries provide special functions dedicated to each axis.

rotate*(angle)

In the second form, **angle** is the angle of counterclockwise rotation in degrees, and * is one of **X, Y** or **Z**.



Positive Rotation about the *X*-Axis

The method for calling a rotation matrix is similar to translation. For example, this:

```
mv = mult( mv, rotateX(a) );
```

will have the following effect:

$M_{modelview} = M_{modelview} * R_x(a);$

Where $R_x(a)$ denotes the rotation matrix about the x-axis for degree *a:*

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

Applying rotation around the y-axis or z-axis can be achieved respectively by these function calls:

```
mv = mult( mv, rotateY(a) ); // rotation about the y-axis
mv = mult( mv, rotateZ(a) ); // rotation about the z-axis
```

Scaling

```
scale(sx, sy, sz);
```

where **sx, sy** and **sz** are the scaling factors along each axis with respect to the local coordinate system of the model. The scaling transformation allows a transformation matrix to change the dimensions of an object by shrinking or stretching along the major axes centered on the origin.

Example: to make the wire cube in this week's sample code three times as high, we can stretch it along the y-axis by a factor of 3 by using the following commands.

```
// make the y dimension 3 times larger
mv = mult( mv, scale(1, 3, 1));

//Send mv to the shader
gl.uniformMatrix4fv(mvLoc, gl.FALSE, mv);

// draw the cube
gl.drawArrays(gl.LINE_STRIP, wireCubeStart, wireCubeVertices);
```

- It should be noted that the scaling is always about the origin along each dimension with the respective scaling factors. This means that if the object being scaled does not overlap the origin, it will move farther away if it is scaled up, and closer if it is scaled down.
- The effect of concatenating the resulting matrix to the global model view matrix is similar to translation and rotation.

C. The Order of Transformations

- When you post-multiply transformations as we are doing and as is done in classic OpenGL, the order in which the transformations are applied is the opposite of the order in which they appear in the program. In other words, the last transformation specified is the first one applied. This property is illustrated by the following examples.
- The initial default position for the camera is at the origin, and the lens is looking into the negative z direction.
- Most object models, such as cubes or spheres, are also defined at the origin with a unit size by default.
- The purpose of model view transformation is to allow a user to re-orient and re-size these objects and place them at any desired location, and to simplify positioning them relative to one another.

  Example: Suppose we want to rotate a cube 30 degrees and place it 5 units away from the camera for drawing. You might write the program intuitively as below:

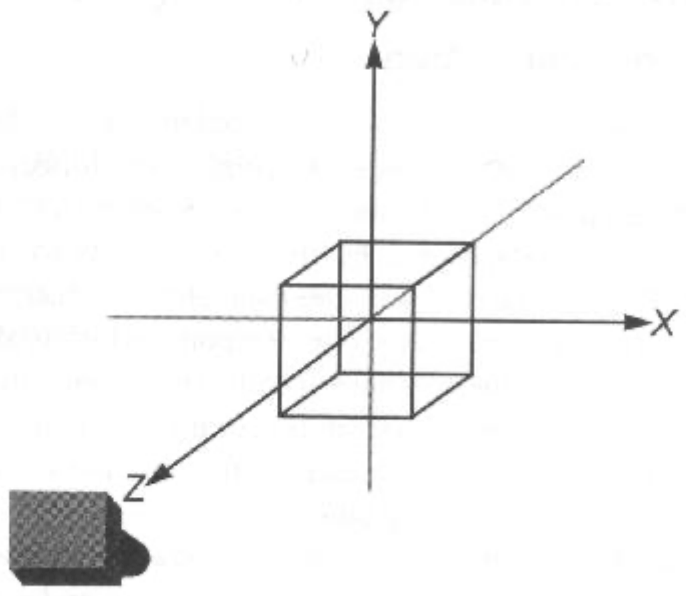
```
// first rotate about the x axis by 30 degrees
mv = mult( mv, rotateX(30));

// then translate back 5
mv = mult( mv, translate(0, 0, -5));

// Copy mv to the shader
gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));

// Draw a cube model centered at the origin
gl.drawArrays(gl.LINE_STRIP, wireCubeStart, wireCubeVertices);
```

The following figure shows the effect of these transforms:



Results of Rotation Followed by Translation

If you run this program, you might be surprised to find that nothing appears in the picture! Think about **WHY**.

If we modify the program slightly as below:

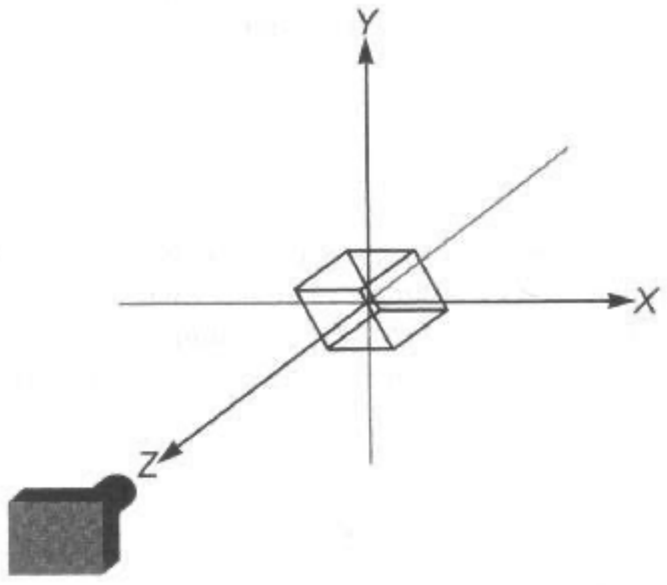
```
// first translate back 5
mv = mult( mv, translate(0, 0, -5) );

// then rotate about the x axis by 30 degrees
mv = mult( mv, rotateX(30) );

// Copy mv to the shader
gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));

// Draw a cube modelcentered at the origin
gl.drawArrays(gl.LINE_STRIP, wireCubeStart, wireCubeVertices);
```

The following figure shows the new result:



Results of Translation Followed by Rotation

D. Modeling Transformation vs. Viewing Transformation

- OpenGL uses concepts of a modelling transformation and a viewing transformation.
- The ***modelling transformation*** is the product of the calculations for creating and laying out your model (making sure everything is correctly positioned and oriented relative to everything else in the model). The transformation functions scale(), rotate*() and translate() can be used to alter the modeling matrix.
- The ***viewing transformation*** is the sequence of calculations for viewing the model (positioning the viewpoint so that you view the model from the orientation and position you desire). You could also use the combination of scale(), rotate*() and translate() for viewing transformations. The following discussion explains how this approach works. However, it involves the concepts of local and global coordinates and could be very confusing to some students. I would like to suggest students to skip this part first (notice I labeled it **OPTIONAL**), and proceed with the easy approach, lookAt(), discussed next.

OPTIONAL

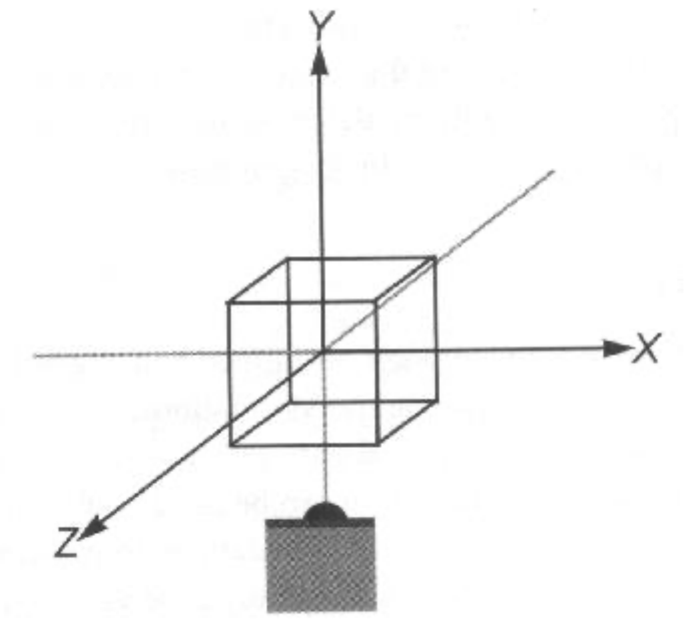
- First let's look at the following code:

```
• mv = mult( mv, translate(0, 0, -5) );
• mv = mult( mv, rotateY(30) );
• gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));
  draw...(gl...., ..., ...); //Draw the model
```

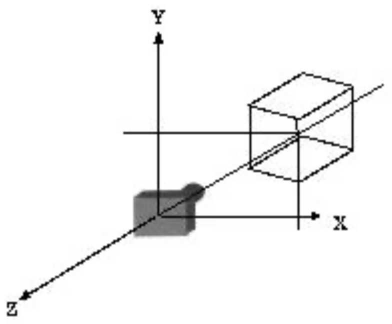
Working down to the model's **local coordinate** system, we first move the local origin down the negative z-axis by 5 units and then rotate that coordinate system about the y-axis by 30 degrees.

Working up to the **global coordinate** system from the model, we first rotate the coordinate system about its origin by -30 degrees, then move it's origin down the positive z-axis by 5 units. The model is fixed, but the global coordinate system is rotated and translated. The viewpoint locates at the origin of the global coordinate system. Remember that in the global coordinate approach, the order is reversed, and the orientation order is also reversed.

The following picture illustrates the local approach to a rotation:



The following picture illustrates the global approach to a rotation:



The Gloabal Coordinate Approach to a Rotation

viewing with lookAt

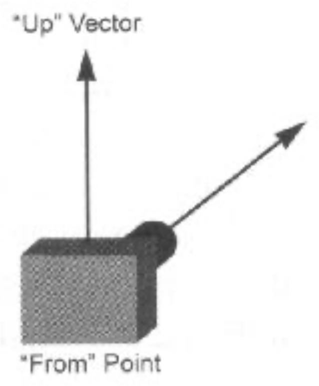
mat4 lookAt (vec3 eye, vec3 at, vec3 up)

The lookAt() function defines a viewing transformation

Parameters

- **eye:** specifies the position of the eye point
- **at:** specifies the position of the reference point
- **up:** specifies the direction of the up vector

The lookAt() function makes it easy to move both the "from" and the "to" points in a linear manner. For example, if you need to pan along the wall of a building located away from the origin and aligned along no axes in particular, you could simply take the "to" point to be one corner of the building and calculate the "from" as a constant distance from the "to" point. To pan along the building, just vary the "to" point.



Physical Relationship of Parameters for `gluLookAt()`

The ModelView Matrix

You may be confused about the exact distinction between modelling and viewing. This confusion stems from the fact that we are imitating classic OpenGL which uses **one** matrix to represent **both** the modeling and viewing steps for everything that is drawn - the **Modelview** matrix. The transformation used to describe the model and the transformation used to describe the viewpoint's location and orientation coexist in that one matrix. This approach results in a simpler set of calculations in the graphics pipeline - it is much faster to calculate the modelview matrix once on the CPU than to potentially do it thousands of times in the shader program - once for every single vertex to be drawn. Matrix multiplication is not commutative but rather **associative**, which means that the product of ((AB)C) is the same as (A(BC)). Thus OpenGL's Modelview matrix is **logically** the product of a viewing matrix and a modeling matrix.

$M_{modelview} = M_{viewing} * M_{modeling}$

What this means is that your viewing transformations must be entered into the Modelview matrix before modeling transformations.

E. Saving and Restoring the Matrix

Whichever method you use, you will almost always need to either reset the matrix to the identity matrix, or save and restore a previous matrix state. To reset to the identity matrix use code like this:

```
mv = mat4(); //restore mv to the identity matrix
```

To save and restore a matrix you can use a matrix stack. Classic OpenGL had one built in, but, like the rest of the matrix functions, it is missing in modern OpenGL varieties and must be provided by an external library. You can use any stack-like data structure that can handle your math library's matrix class. Javascript's arrays, which provide .push() and .pop() functions, are perfect.

To make a matrix stack in Javascript, write code like this:

- Add to your global variables

```
//global modelview matrix stack
var matStack = [];
```

- In display, use .push() and .pop() around transforms that should only affect one or a limited set of objects.

```
matStack.push(mv);

//Apply transforms to modelview matrix
//Draw objects
//... etc ...

//restore old modelview matrix
mv = matStack.pop();
```

You can store any mat4 matrix on the matrix stack so long as you remember to pop back to the correct matrices in the correct sequence.

F. Viewport and Projection Transformations

Once you have learned Modelview transformations, the next step is to understand projection modes and viewport mapping.

Viewport Transformation

The gl.viewport() function is used to specify a **viewport**, or the drawable area in your WebGL canvas. It can be cause your draws to use all or only a portion of the canvas. It is best to call it at least as often as the canvas changes size and only after you know the size of the canvas. In WebGL, that means it could be in your init() function, but if you are using a variable sized canvas it would be best to call gl.viewport() as part of rendering.

```
gl.viewport(x, y, width, height)
```

- **x, y**: specify the lower left corner of the viewport in **canvas** coordinates.
- **width, height**: specify the width and height of the viewport in **canvas** coordinates.

Example:

```
// Correctly uses the size of the canvas as stored in the gl context
// to set a full size viewport
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
```

Projection Transformations

There are two basic methods of converting 3D images into 2D ones.

- The first is **orthographic**, or **parallel** projection. You use this style of projection to maintain the scale of objects and their angles without regard to their apparent distance. MVnew.js provides ortho() to do this type of projection.
- The second is **Perspective projection**. This is the most popular choice in 3D graphics. A perspective projection matrix can be created with the perspective() function.

Projection is handled by the $\mathbf{M}_{\mathbf{Projection}}$ matrix. You do not usually concatenate to the projection matrix as you do with the modelview matrix.

Orthographic Projection

ortho()

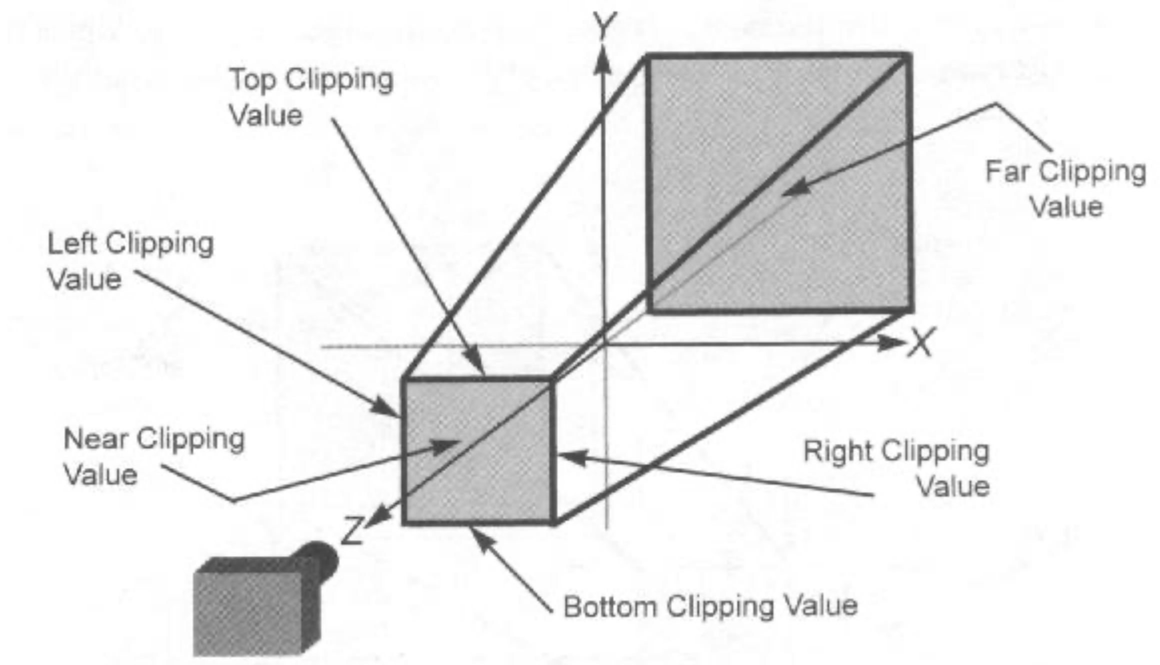
```
void ortho( left,   right,
         bottom, top,
         near,   far )
```

Parameters:

- **left, right**: Specify the coordinates for the left and right vertical clipping planes;
- **bottom, top**: Specify the coordinates for the bottom and top horizontal clipping planes;
- **near, far**: Specify the distances to the near and far depth clipping planes. Both distances must be positive.

ortho() describes an orthographic projection matrix. (left, bottom, -near) and (right, top, -near) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively, assuming that the eye is located at (0, 0, 0). -far specifies the location of the far clipping plane. Both near and far must be positive.

The following figure approximates an orthographic (actually it is for frustum() - see below) volume and the ortho() parameters



perspective()

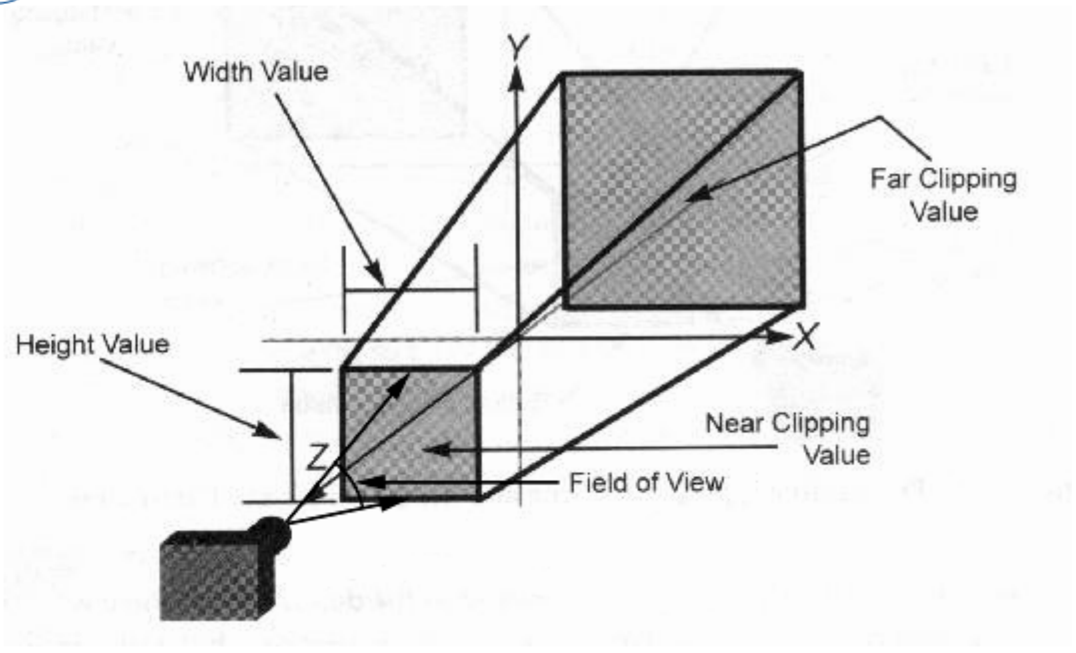
In old OpenGL systems, a function with the same parameters as ortho() could create perspective transformations. It was called frustum() and though it was powerful, it was not very intuitive. There is a much simpler perspective command, called perspective().
Like frustum() it generates a perspective viewing volume but only a simple one. It lacks the flexibility of frustum which can be manipulated to achieve special effects.

```
    void perspective( fovy,  aspect,
                zNear, zFar )
```

Parameters: fovy: Specifies the field of view angle, in degrees, in the y direction; aspect: Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height); zNear: Specifies the distance from the viewer to the near clipping plane (always positive); zFar: Specifies the distance from the viewer to the far clipping plane (always positive).

perspective() specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in perspective should match the aspect ratio of the associated viewport. For example, aspect=2.0 means the viewer's angle of view is twice as wide in x as it is in y. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The following shows perspective viewing volume and the perspective() parameters



PROJECT7

Firstly open LMS and download **Project7.zip** file.

- You will practice modelling and viewing transformations with boxes.html and boxes.js then answer some questions.

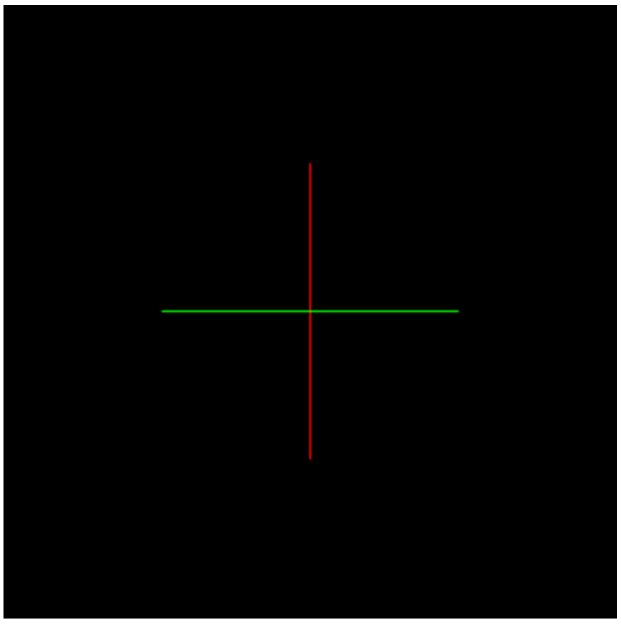
Goals of this Project:

- Viewing Transformations: through lookAt or equivalent modelling transformations
- Projection Transformations: through perspective, ortho
- Modelling Transformations: rotate*, translate, scale and matrix stack.

PART 1

Start with boxes.html and boxes.js form **Project7.zip** file.

When you run boxes.html, you should see as below firstly.



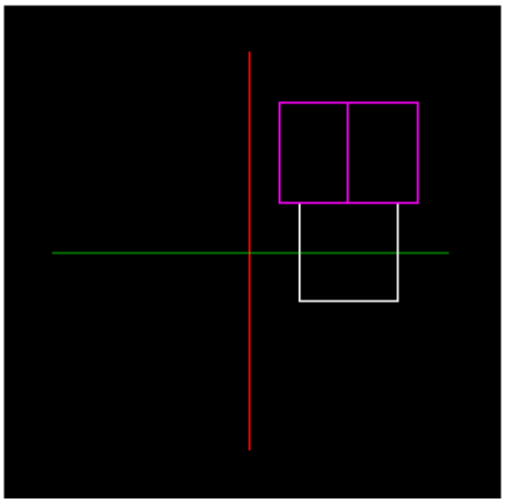
- As written, this program draws a basic coordinate system with a green x-axis, a red y-axis, and a blue z-axis. These will be referred to in the instructions as the **axes.**
- With the initial camera settings, you are looking directly down the z-axis so you will not see it.

Make the following changes. **Follow the instruction and prepare a project report pdf file for uploading to the LMS, your pdf file should include all parts. Please make sure your answers are numbered as below:**

1. Comment out the lookAt() call and replace it with translate() with parameters (0,0,-15). Is there any change in the display? Why ? Why not? Please explain in detailed and add image. **Explain your process.**
2. Comment out both the lookAt() and translate() lines. What happens? Why? **Explain your process.**
3. Restore the lookAt() call.
4. Take a look at the perspective() call. The aspect ratio you were originally given was 1.0. **Explain your process.**
   a. What happens when the aspect ratio is 1.0 and you change the canvas dimensions in boxes.html to width=”400”, height=”400? Add the result image.
   b. How about width=”256” and height=”256”. Add the result image.

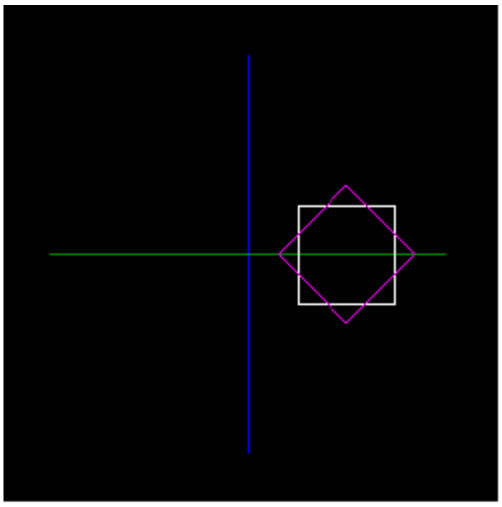
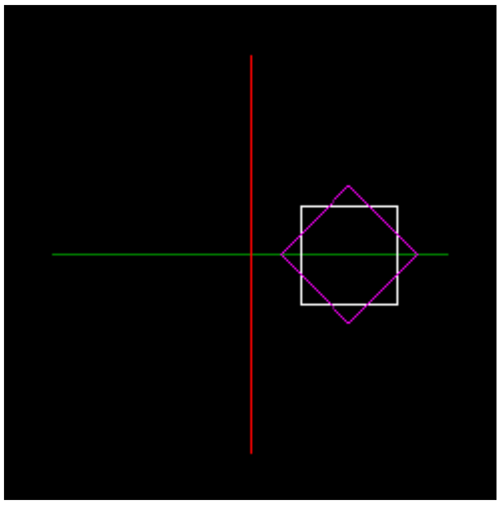
c. The aspect ratio allows us to compensate for different shapes of canvas. Study there commendations in point **2.** of **WebGLAnti-Patterns**, then modify the aspect ratio in your perspective call so that it is an appropriate ratio of width to height based on the actual dimensions of the viewing area. Test the result with the two suggested canvas shapes to be sure you got it right.

5. Draw a wireCube centered (0,0,0) relative to the axes. You can use the provided buffers and related shape data. Do this in the render function. **Explain your process.**
6. Move this cube so that it is centered as (1,0,0) relative to the axes. **Explain your process.**
7. Draw a second cube after the first –in a new colour(**blue**) if you can- and rotate it 45 degrees around the y-axis. **Explain your process.**
8. Place this rotated cube directly above the first cube. It will be centered at (1,0,0) relative to the axes. Be careful of the order of transformation. **Explain your process.**
9. The perspective view makes the two cubes look a little awkward. Try using orthographic projection instead of the perspective call. The function for that is: ortho. Use left, right, bottom, top, near and far values that include the whole scene and not much more. See the picture for expected results. **Explain your process.**

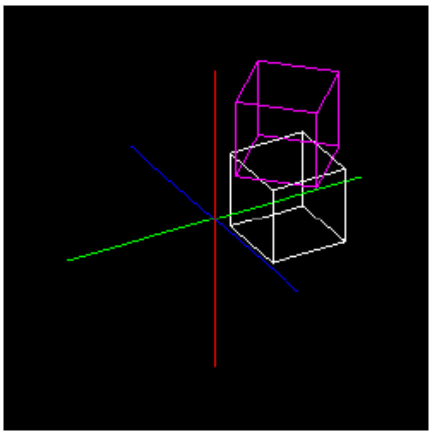


10. Rotate everything (using modelling transformations NOT lookAt) so that looking down at the top of the boxes and seeing the blue z-axis (and no red y-

axis). See the picture for expected results. **Explain your process.**

11. Rotate everything so that you can see all three axes along with the cubes. See the picture for expected results. **Explain your process.**



Submission

IMPORTANT: You can do this project in groups of 2 people. Project8 will be the continuation of Project7, so you should work with the same group of friends in Project7 and Project8.

PROJECT 7

A working version of the program showing a result similar to that shown in step 11. Be sure to leave commented code where requested. Please upload your files as a zip file to “Project7 Files Submission” part.

- You will create project report and upload your project report to “Project7 Report Submission” part on LMS.
- Your report file(pdf) with written answers for the questions with the results images in Steps 1, 2, 4, 5, 6, 7, 8, 9,10 and 11 and explain your process for Part1.

Grading Rubric

PROJECT 7	**Points**
A working version of the program for Part1 and source files	15
Answers of questions and explain your process	50
Quiz	35