UNIT II

A. The Classic OpenGL Transformation Pipeline

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 module, as in the textbook, 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:

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 = Mmodelview*P.

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

P" = [x", y", 1]T = Mprojection*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 always uses 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 module 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
GLint projLoc;
GLint 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 uniform* (GLES 2.0 man page) (WebGL Spec) 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, 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, p); // copy p to uniform value in shader
```

B. Elementary Transformations

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 are 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 Mmodelview, 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 MV.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:

Mmodelview = Mmodelview * T(dx, dy, dz);

Where T(dx, dy, dz) =

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.

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

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

will have the following effect:

```
Mmodelview = Mmodelview * Rx(a);
```

Where Rx(a) denotes the rotation matrix about the x-axis for degree a: Rx(a) =

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

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(mv));

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

The following figure shows the new result:

D. Modeling Transformation vs. Viewing Transformation

OpenGL uses concepts of a modeling transformation and a viewing transformation.
The modeling 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()