Lab 7: Transformations with Matrices

In this lab you will learn how to use matrices to perform model, view orientation, and projection transformations. To begin, download the Lab7.zip file from the Canvas site, unzip it, and then open the Lab7.html file in your browser. You should see parts of three triangles (yellow, cyan, and red) with a white specular highlight partially visible. You can use the left and right arrow keys to move the cyan triangle to the left and right on the screen. If you press and hold the up-arrow key, you will notice that the triangles disappear one at a time (first the red, then the cyan, then the yellow triangle). The reason the triangles disappear is that the eye point is a being moved behind them. I have the code written so that the up and down arrow keys can be used to modify the z position of the eye. Once the viewer "passes" a triangle, it gets "culled" away since it is no longer visible from the view position and direction of view.

To implement the eye effect, I am using the glMatrix library discussed in the textbook. I have included a copy of gl-matrix.js with the Lab 7 download, but you can also obtain a copy from glmatrix.net. The Lab7.html file loads gl-matrix.js so that we can access its functions in our JavaScript code. The eye effect has been created using the `mat4.lookAt` function from the glMatrix library to create a view orientation transformation matrix (see `drawModel` in model.js). This matrix is sent to the vertex shader using a uniform variable to be applied to each vertex. Also, in `drawModel`, you can see how the three triangles are being translated, rotated, and scaled to position them on the display. Take a few moments to look at the code in the `drawModel` function to observe how appropriate model transformation matrices are being constructed using the glMatrix functions (translate, rotate, and scale), concatenated when necessary, and then passed to the vertex shader using uniform variables. This makes it possible to define a single triangle object and then render it many times in different orientations using the matrix transformations. Vertices are defined for only one triangle in the code, but I am drawing it three times with different colors and modeling transformations applied so that it appears as three different triangles.

The glMatrix library contains an `mat4.ortho` function that I am using to create an orthographic projection transformation matrix (see the code right after the comment `//Set up the projection transformation matrix` in `drawModel`). It is important to understand that the orthographic projection now makes it possible for us to have visible coordinates outside of the range of [-1,1] for each axis. As was the case with the model matrix and view orientation matrix, the projection matrix is being sent to the vertex shader and multiplied by each vertex (after the model and view orientation matrices have been applied). Try changing parameters two, three, four, and five (the left, right, bottom, and top parameters) of the `mat4.ortho` function. What effect do your changes have?

If you look carefully at the `mat4.translate` calls in `drawModel` you will notice that each triangle is actually translated a different amount in the z direction (which is why they disappear at different times when the eye point is moved towards them). The eye position is actually closest to the red triangle, but this triangle appears behind the other two triangles. This is because WebGL renders objects in the order they are drawn unless you tell WebGL to do a **depth test** (meaning draw the closest object to the eye rather than the last one drawn). Look in `initModel` and find the comment `//TODO 1`. Follow the instructions there to enable depth testing. If you save the change and reload the page in your browser the red triangle should now appear in front of the other two triangles.

You have probably noticed that all three triangles appear to be the same size. However, in the real world if two objects are the same size but one is further away from the eye than the other, then the further object will appear smaller. We can model this effect (called **perspective foreshortening**) using a perspective projection (instead of an orthographic projection that preserves the distances between vertices). To use the perspective projection, you will need to add the appropriate code (and remove the orthographic projection) – see the comment that starts `//TODO 2:` in `drawModel`. When you make the change and run the program again all you should see is the red triangle (and it now appears a whole lot larger)! This is because the viewer is right in front of the red triangle and it obstructs the view of the other two triangles. If you move through the red triangle (using the up arrow key) you should be able to see the other two triangles, and as you move towards them they should get larger. Perspective projections are critical to any type of first-person perspective graphics applications (like first person shooters). Try changing the positions and orientations of the triangles using the modeling transformation functions `mat4.translate`, `mat4.rotate`, and `mat4.scale` to build an appropriate `model_matrix` for each triangle (see `//TODO 3:`). Are you able to move the objects around in the scene in the way you hoped?

Now, look at the vertex shader code in Lab7.html. Notice how the vertex position sent into the shader is multiplied by the modelMatrix, viewMatrix, and projectionMatrix to determine the final vertex position assigned to `gl_Position`. The other big change in this shader code is that the `out` variables `normal` and `position` are assigned values in "eye coordinates". Lighting calculations in the fragment shader are done from the vantage point of the eye, so the values of these variables need to be passed to the shader in that coordinate system.

Note that to the extent possible, to get a performance benefit you really want to multiply matrices in your WebGL application before sending them to your shaders. It may have been better to multiply all of the matrices together in the application and then send a single *transformation matrix* to the vertex shader as a uniform variable, rather than have three separate uniform variables for the transformation matrices as I did in this lab. I wrote the lab in the way I did to help emphasize the point of the three different types of transformations (modeling, view orientation, and projection). Similarly, I have calculated the normal matrix in the vertex shader when it would be calculated fewer times if done in the JavaScript application code. Another inefficiency in the program is the recalculation of the projection matrix in each call to `drawModel`. The projection rarely changes during the execution of a program and can often just be calculated once in `initModel`.

There is one more example in this lab you can try out as time allows. You will need to modify the code by commenting out the first example (everything between the comments //THE FIRST EXAMPLE STARTS HERE and //THE SECOND EXAMPLE STARTS HERE), and then uncommenting the code for the second example. The second example shows the transformation of the point that we worked out together during class time (see slide 17 of the PowerPoint slides COMP153_matrices_solutions). The green dot is the initial position of the point (5, 0, 0) and the red dot is the final position of the point after the transformations have been applied (10, 2.5, 0).

Once you are finished with this lab you can get started on Assignment 6.