Michael McQuade

Zhigang Deng

Computer Graphics

28 March 2021

Transformation and Projection

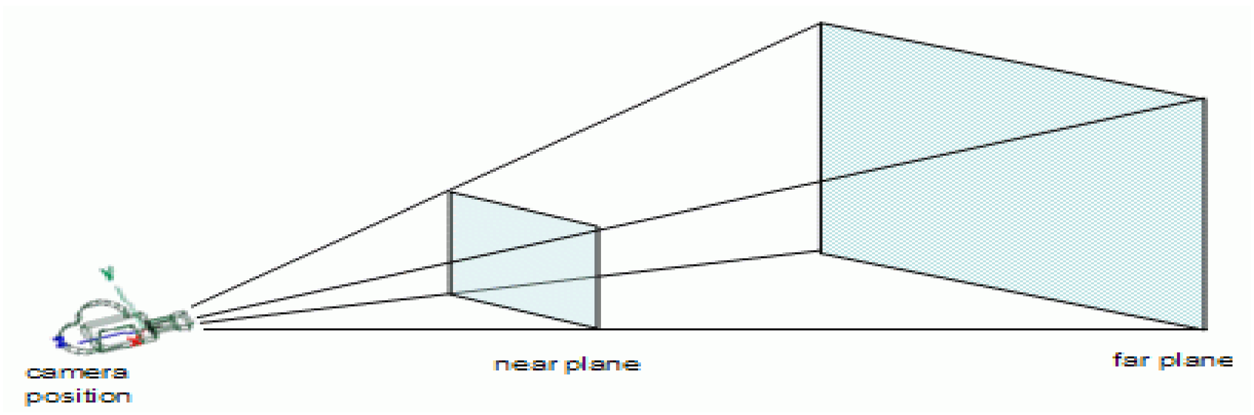
This report will cover the methods involved in applying various transformations to a 3D object. In the previous assignment, the goal was to render a 3D object, namely a teapot. This teapot was rendered given an input file of triangles using the scan line method. In this assignment, that method is still used, but now the focus of the assignment is implementing various transformations.

In order to do this, representing the various vertices that make up a triangle needs to be done with matrices now instead of vectors. In our case, we were representing the vectors in a data structure called GzVertex, and now we will need to introduce a new data structure, GzMatrix. This GzMatrix data structure will be our representation of a matrix in our C++ implementation. When the vertices of the triangle are represented as a matrix it makes it much easier to apply any kind of transformation to it using linear algebra and matrix multiplication.

One important thing is that matrix multiplication has a few caveats such as not all matrices can be multiplied against each other, there are undefined multiplications. In order to avoid this, additional error handling and assertions are made inside of the code to make sure that this situation is not processed. Additionally, multiplication is not commutative. That is to say, you cannot multiply matrices in any order and expect the same result. In fact, depending on the shape of the input matrices, reordering the multiplication may make the multiplication undefined.

There are two matrices that need to be kept track of in our implementation. The transformation matrix and the projection matrix. Both of these matrices start out as the identity matrix. The identity matrix has a special property in that any matrix multiplied by an identity matrix will return or output the same matrix. This way, we can build up any transformations, and if there are none, we end up multiplying it by the identity matrix. The transformation matrix will hold any types of transformations that will be applied to our vertices. That is, any rotation of the object, scaling, or even the viewing transformation, will be stored in this transformation matrix. Since these transformations are all matrices, we don't need to keep separate matrices for each transformation, instead, they can be multiplied with each other and the final transformation matrix will apply all of the transformations as if we had multiplied all of those matrices independently.

The other matrix, the projection matrix, is generated by the functions orthographic or perspective. Only one of these should be called for each call of draw. The perspective function sets up a viewing frustum so that the display of the 3D teapot is projected as if seen through a pane of glass. In this case, that would be the users monitor. To calculate this, near and far depth values are required as well as the angle of the field of view. Due to the nature of this, depending on the position of the camera, objects may look larger or smaller. See below image for reference.



The other function that utilizes the projection matrix is the orthographic function. This function considers all light travels in parallel, so it produces a flatter 2D representation of the 3D object. A midpoint between these two views would be the isometric projection, which is not covered in this assignment or report.

One hurdle to getting started with this assignment was that the sizes of the vertices are so small. That is because once projection is done, the sizes will match up with the screen space. However, before getting the projection methods to work, another idea that I pondered was applying the transformations and then applying a viewport transformation in order to render the teapot. Using the viewport as scale, this actually allowed initial rendering for the teapot. While this code is not in use, the code looked like such, and was multiplied with the final matrix to get the teapot to render.

```
GzMatrix Gz::CreateViewportMatrix(int x, int y, int w, int h)
{
   GzMatrix m = Identity(4);
   m[X][W] = x + w / 2.f;
   m[Y][W] = y + h / 2.f;
   m[Z][W] = 255 / 2.f;

   m[X][X] = w / 2.f;
   m[Y][Y] = h / 2.f;
   m[Z][Z] = 255 / 2.f;
   return m;
}
```

Code for creating a viewport matrix, which is an additional transformation

This was not the correct choice in getting the object to render, but it did provide a great starting point. Eventually this was removed from the code after projection was fully implemented because the projection solved the issues that this viewport matrix was trying to solve.

Works Cited

“GluLookat.” *Khronos*, 2006,

www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml.

“GluMultMatrix.” *Khronos, 2006,*

https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glMultMatrix.xml

“Rotation (Mathematics).” *Wikipedia*, Wikimedia Foundation, 2 Mar. 2021,

https://en.wikipedia.org/wiki/Rotation_(mathematics).

“Translation (Geometry).” *Wikipedia*, Wikimedia Foundation, 25 Jan. 2021,

https://en.wikipedia.org/wiki/Translation_(geometry).

gluPerspective.” *Khronos, 2006,*

https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluPerspective.xml

“glOrtho.” *Khronos, 2006,*

https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glOrtho.xml