Bryan Bridge 3/12/15

Perspective Projection of 3D Objects

Problem and Motivation

The first goal of our project was to understand and demonstrate perspective projection as well as other linear transformations as they are applied to shapes in 3D. The secondary goal was to attempt some form of shading for the surface of these shapes. Our inspiration came from a lecture last week, when it was explained how linear algebra is related to computer graphics. We were all fascinated by this topic as it's something we interact with every day. CGI reaches impressive new levels of detail every year and it's exciting knowing that at its core it's just math.

Creation of Objects in 3D Space

We chose three objects for our demonstration: a cube, a dodecahedron, and a sphere. Each object was first created at the origin. In order to set up the cube and dodecahedron, an array of standard vertices was stored: $[(\pm1, \pm1, \pm1)]$ and $[(\pm1, \pm1, \pm1), (0, \pm1/\varphi, \pm\varphi), (\pm1/\varphi, \pm\varphi, 0), (\pm\varphi, 0, \pm1/\varphi)]$, respectively. For both of these shapes, by checking the distance between each vertex, a new array was created listing the edges that should be drawn. The shapes' size could be changed using scalar multiplication.

For the sphere the points of a wireframe were generated using two simple loops that created Cartesian coordinates from spherical ones. The loops were written to allow us to vary the step size for θ and ϕ, and the size of the radius. In this case the lines to be drawn were from one generated point to the next, skipping a line at regular intervals to avoid connecting unnecessary points in the wireframe.

Translating and rotating objects in our scene proved to be straightforward, although due to being short on time we only implemented rotation for the cube. Rotation needed to be implemented at the origin, as conceptually it is done by rotating an object around an axis. Each point in an object could be rotated about an axis using these rotation matrixes:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In order to translate the objects, or move them in our scene, we added the desired position to each point that made up the object. Starting each object centered at (0, 0, 0) made this simple. Figures 1 and 2 show a couple of 3D renderings with varied translations and rotations.

Figure 1

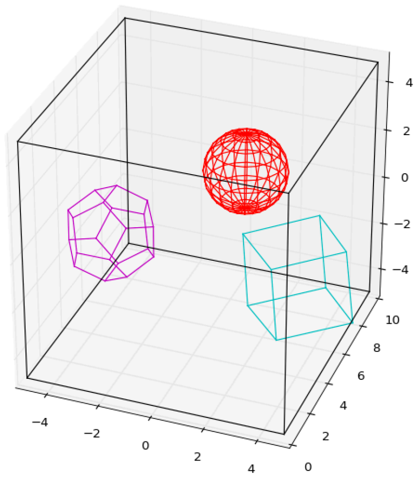
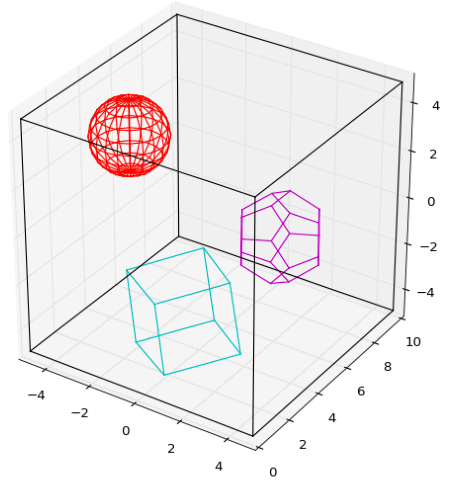


Figure 2



Projection to 2D

Given coordinates and line/edge drawing pairs for the objects in 3D, we were able to draw a projected image in 2D as seen from a locked position on the z-axis, known as the image plane. The camera sits at the origin looking down the z-axis (all values on the z-axis positive)- the distance of the image plane from the origin we label "d." When each stored point for the objects is mapped to 2D, its x and y-values are scaled by the ratio d/z. Of course the z-value is lost and now depth and size information have been combined and are indistinguishable. By changing d we can change the field of view. Figure 3 gives a visual explanation of the relationship between coordinates and the d value, while Figures 4 and 5 show our output for a scene with two different values for d.

Figure 3

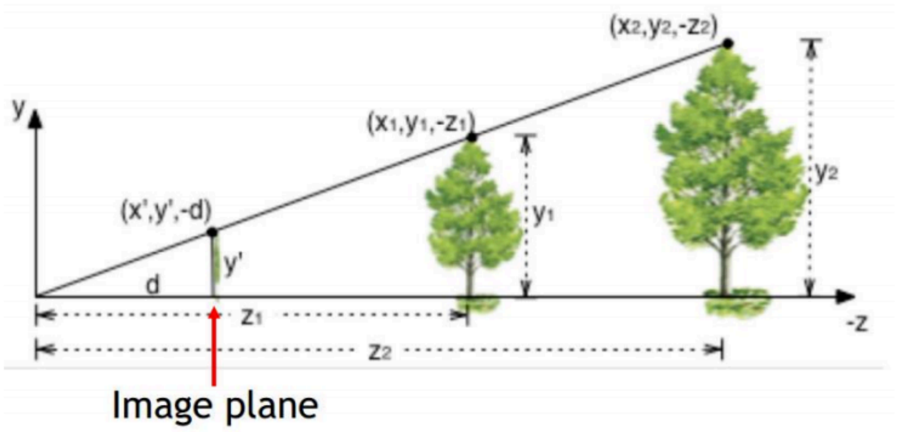
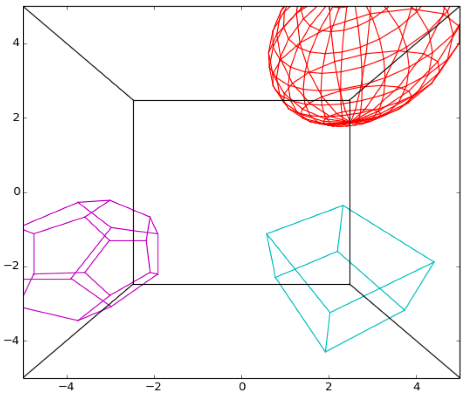
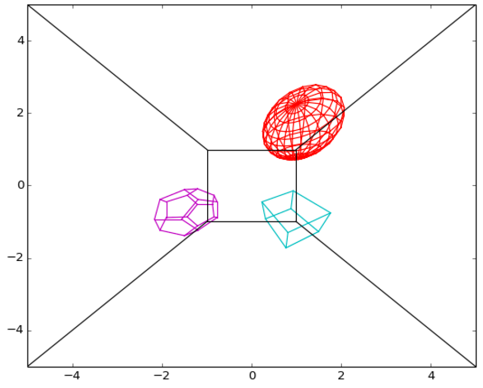


Figure 4



d = 5.0

Figure 5



d = 2.0

Culling and Shading

Unfortunately we only had time to implement culling and shading for the cube. Back-face culling and shading with a provided vector for light direction were written together as one method. Each face of the cube was stored as its vertices in CCW order. With that information we generated two vectors from edges of a face, then took their cross product to find the vector normal to the face. If the vector from the origin (camera) to any point on the face is at an angle of 90° or greater to the surface normal, then the face can't be seen. We used one of the vertices of the face for the vector from the camera, and then found this angle using Equation 1. If the face could be

Equation 1

$$\cos^{-1}\left(\frac{\vec{\mathbf{v}} \cdot \vec{\mathbf{u}}}{\|\vec{\mathbf{v}}\|\|\vec{\mathbf{u}}\|}\right)$$

seen, then it was shaded. Using the vector for the direction of light, the method again uses Equation 1 and compares that vector to the surface normal. The angle returned fit into one of several preset intervals between 0° and 180°, and the face was shaded according to the interval. Figure 6 shows the cube shaded with light shining from top right, at the back of the scene- and figure 7 shows it shaded with light shining from bottom left, below the camera.

Figure 6

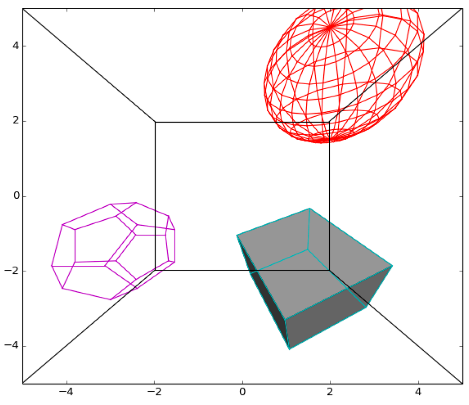
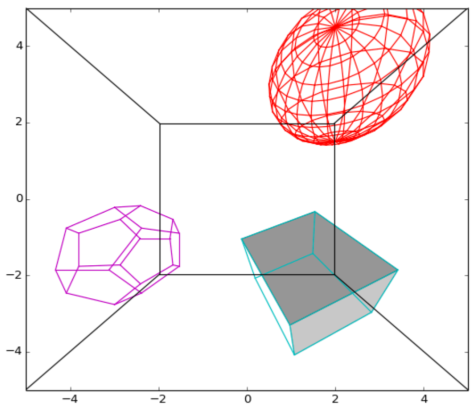


Figure 7



Results and Conclusion

We found all visual effects behaved as expected for the code we had working. Given more time, we would have liked to implement shading for the other objects, a shading algorithm with a gradient function so that faces are not all one color, and the ability to move the camera around the scene. While this work was not heavy in numerical methods, we were proud to have chosen a project that we were passionate about and had exciting results we could demonstrate.