CS370 - Assignment 2

1. Typically when we draw 3D objects on paper (or the board) we draw the x and y axes at 90 degrees (x axis pointing right and y axis pointing up). We then represent the z axis (which would be coming out of the board) by a line at -135 degrees from the x axis (diagonally down towards the left). This type of projection is known as an *oblique* projection. Determine the projection matrix that would render the objects in this fashion.

To achieve this projection we need to convert the axes as follows

$$x = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \Rightarrow \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \qquad y = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \Rightarrow \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \qquad z = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \Rightarrow \begin{bmatrix}-\frac{1}{\sqrt{2}}\\-\frac{1}{\sqrt{2}}\\0\\0\end{bmatrix}$$

using the generic projection matrix

$$P = \begin{bmatrix}p_{11} & p_{12} & p_{13} & 0\\ p_{21} & p_{22} & p_{23} & 0\\ p_{31} & p_{32} & p_{33} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Thus taking the original x-axis and multiplying it by P and equating it to the projected x-axis gives

$$Px = \begin{bmatrix}p_{11} & p_{12} & p_{13} & 0\\ p_{21} & p_{22} & p_{23} & 0\\ p_{31} & p_{32} & p_{33} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}p_{11}\\p_{21}\\p_{31}\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$$

Thus we see that $p_{11} = 1, p_{21} = 0, p_{31} = 0$

Similarly, taking the original y-axis and multiplying it by P and equating it to the projected y-axis gives

$$Py = \begin{bmatrix}1 & p_{12} & p_{13} & 0\\ 0 & p_{22} & p_{23} & 0\\ 0 & p_{32} & p_{33} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}p_{12}\\p_{22}\\p_{32}\\0\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

Thus we see that $p_{12} = 0, p_{22} = 1, p_{32} = 0$

Lastly, taking the original z-axis and multiplying it by P and equating it to the projected z-axis gives

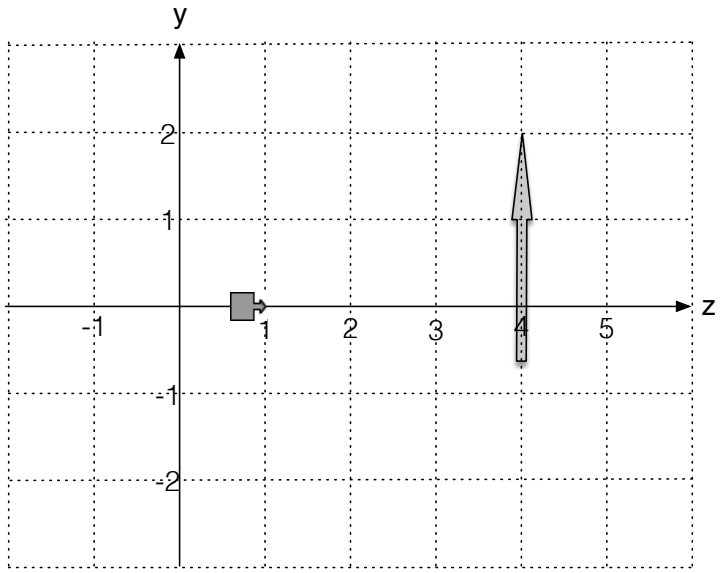
$$Pz = \begin{bmatrix}1 & 0 & p_{13} & 0\\ 0 & 1 & p_{23} & 0\\ 0 & 0 & p_{33} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}p_{13}\\p_{23}\\p_{33}\\0\end{bmatrix} = \begin{bmatrix}-\frac{1}{\sqrt{2}}\\-\frac{1}{\sqrt{2}}\\0\\0\end{bmatrix}$$

Thus we see that $p_{13} = -\frac{1}{\sqrt{2}}, p_{23} = -\frac{1}{\sqrt{2}}, p_{33} = 0$

Thus the final projection matrix for an *oblique* projection is given by

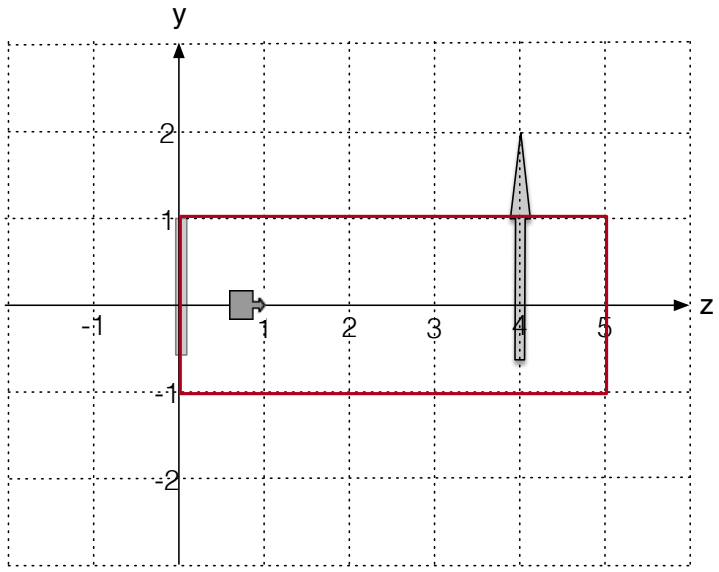
$$P = \begin{bmatrix} 1 & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & 1 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Given the following scene with the camera located at (x, 0, 1) looking at (x, 0, 2), sketch the viewing volume and determine the final size of the object in the rendered scene for the following projections (**Note:** the x-axis is ignored.) **Hint:** Use similar triangles to relate the relative sizes of the objects to the relative distances from the camera.



- **ortho(x, x, -1, 1, -1, 4)**

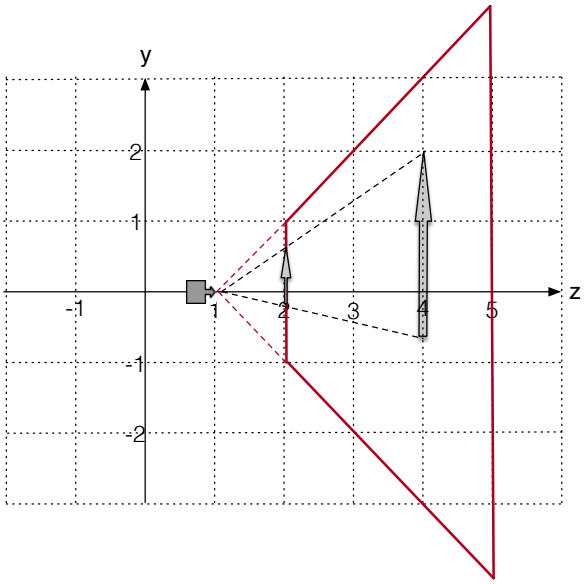
Note: The extents of the viewing volume are *relative* to the position of the camera, thus the viewing volume will extend one unit below, one unit above, one unit behind, and four units in front of the camera location. Furthermore, since the projection is *orthographic*, the viewing volume will be **rectangular** as shown below



For *orthographic* projections, the height of the object in the rendered scene is **the same** as the original height of the object. However, we see that the top part of the object is *clipped* by the viewing volume, thus the portion of the object remaining in the viewing volume is roughly 1.5 units high.

- **frustum(x, x, -1, 1, 1, 4)**

Note: The extents of the viewing volume are for the *near clipping plane, relative* to the position of the camera, thus the *near clipping plane* will extend one unit below, and one unit above the camera, and be one unit in front. The *far clipping plane* will then be four units in front of the camera with the height determined by the projectors from the camera through the extents of the near clipping plane. Furthermore, since the projection is *perspective*, the viewing volume will be a **frustum** as shown below

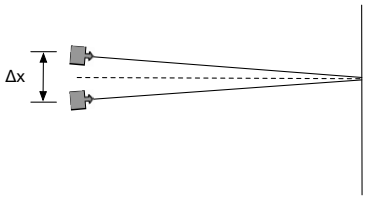


For *perspective* projections, the height of the object in the rendered scene is **proportional** to the ratio of the distance to the camera as shown by the projectors above. Thus we can approximate the height of the image, where h_i is the height of the image, h_o is the height of the object, d_n is the distance of the near clipping plane *from the camera*, and d_o is the distance of the object *from the camera* as

$$h_i = h_o \frac{d_n}{d_o} = 2.5\frac{1}{3} = \frac{5}{6} \approx .83$$

3. Some of my research has been in the area of stereoscopic 3D images, which is now used extensively for VR. To create a stereoscopic 3D image, we simply render the scene from two different viewpoints (one to represent what the left eye would see and one to represent what the right eye would see) and then display the corresponding image to each eye using the VR headset. If the viewer is considered to be at the origin with an ocular spacing of Δx, what are the appropriate **lookat()** functions to produce a stereo image pair? **Hint:** We need to render the scene from two different camera locations (separated by Δx). Consider two possible locations where these cameras can be pointed, i.e. the *center* location, to produce a stereoscopic image.

There are two schools of thought on creating stereo views, but both involve generating two images by offsetting the camera by $\pm\frac{\Delta x}{2}$ from a nominal center point (i.e. the standard camera rendering position). The first is known as the *toe-in* method where the cameras are pointed at a common point (e.g. the origin). This would be done using

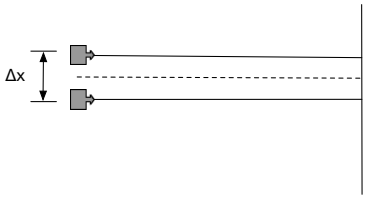


with corresponding

```
lookat(vec3(x - dx/2, y, z), vec3(0, y, 0), vec3(0, 1, 0));
lookat(vec3(x + dx/2, y, z), vec3(0, y, 0), vec3(0, 1, 0));
```

While this method is simple, it can produce vertical parallax issues.

Thus a better method is known as *parallel parallax* which has the two cameras pointing in the same parallel direction. This would be done using



with corresponding

```
lookat(vec3(x - dx/2, y, z), vec3(x - dx/2, y, 0), vec3(0, 1, 0));
lookat(vec3(x + dx/2, y, z), vec3(x + dx/2, y, 0), vec3(0, 1, 0));
```