The MVP Matrix

We have mentioned in our discussion of the graphics pipeline, that the vertex shader is in charge of converting our vertices from *model space* into *clipping space* and we mentioned that this process is achieved through the concatenation of the following matrices:

- The *model matrix* (M) which takes us from *model* to *world* space,
- The *view matrix* (V) which takes from *world* into *view* space, and
- The *projection matrix* (P) which takes us from *view* into *clipping* space.

Given that we know have a better grasp of the pipeline, we are now ready to begin discussing how these matrices are derived and what they look like.

The Model Matrix M

The model matrix is the same as was described for ray tracing. It is constructed by concatenating the different transforms that we want our object to have. Recall that order of operations is important, and so care must be taken to ensure that the multiplication of the matrices happens in the correct sequence.

The View Matrix V

Recall that for a simple pinhole camera, we can define it with the following vectors:

- $\mathbf{w} = (e - l)/|e - l|$
- $u = \mathbf{up} \times \mathbf{w}/|\mathbf{up} \times \mathbf{w}$
- $v = \mathbf{w} \times \mathbf{u}$

These vectors define an *orthonormal basis* which conform camera space, and the origin is then the position of the camera e. So, what we want is a matrix that can transform world space into camera space.

To start, note that the first (and easiest) thing we can do is to shift the origin of the camera to align itself with the origin of world space Therefore, what we are interested in is a translation matrix that will take us to e. We can construct this very simply as follows:

$$\begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The next part is to map the axes of camera space into world space. Ultimately this is going to be accomplished by rotating the axes so they line up with each other. We therefore can construct this matrix as:

$$\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We therefore can construct the final view matrix as follows:

$$V = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & \mathbf{u} \cdot -e \\ v_x & v_y & v_z & \mathbf{v} \cdot -e \\ w_x & w_y & w_z & \mathbf{w} \cdot -e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice the order of operations here. In order to ensure that we are rotating around the right axes, we first need to translate the camera into the origin and then rotate the axes so they line up.

The Projection Matrix P

When we discussed ray tracing, we defined two types of projections: the orthographic and perspective projection. It makes sense then to construct equivalent matrices for OpenGL.

Up until this point, all transforms have left the w component of the vectors unaffected and the bottom row of all transform matrices has always been (0 0 0 1). The exception to this rule are the *perspective projection* matrices. The bottom row contains vector and point manipulating numbers, and we often need to homogenize the points. That is to say, since the final value of w is not 1, we need to divide by w to obtain the homogeneous point. These points exist in a space called *normalize device coordinates* and are used in order to perform clipping computations. In fact, the way clipping is performed is by looking at the value of x, y, z and comparing them against $-w, w$. If they are not within this range, the vertices are discarded.

Orthographic projection on the other hand does not have this problem and will result in the vertices being left in homogeneous coordinates. We begin first with orthographic projections.

Orthographic Projection

Recall that a characteristic of the orthographic projection is that parallel lines remain parallel after projection. A simple projection matrix is then:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix will leave the x, y coordinates and sets the z coordinate to 0, effectively projecting the points onto the plane $z = 0$. This transform has several issues: first it is non-invertible, due to the fact that it drops one dimension (z) and there is no way of retrieving it. The other problem is that the matrix will map *all* values of z onto the plane. It is often convenient (and in fact used) to restrict the values of z to a certain interval within n (near plane) and f (far plane).

A more common way of defining this matrix is by the usage of a six-tuple (l, r, b, t, n, f) denoting the left, right, bottom, top, near, and far planes. This matrix scales and rotates the *axis-aligned bounding box* defined by these planes into an axis-aligned unit cube centred at the origin. Therefore, the viewing volume of the orthographic projection is a parallelogram as we have seen before. It is important to realize that $n > f$ since we are looking down the negative z axis. Since it usually makes more sense to have $n < f$, then it is is simple task of letting the user specify them in this way and negating them afterwards.

In OpenGL, the axis-aligned cube has a minimum corner of $(-1, -1, -1)$ and a maximum corner of $(1, 1, 1)$, while in DirectX the bounds are from $(-1, -1, 0)$ to $(1, 1, 1)$. This cube is called the *canonical view volume* and the coordinates that are defined within it are called *normalized device coordinates*. The matrix P_o is then

$$P_o = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that we can split this into a translation matrix T and a scaling matrix S. In computer graphics, we often use a left-hand coordinate system after projection. This means that for the viewport, the x axis goes to the right, the y axis goes up, and the z axis goes into the view port. Because the far value is less than the near value for the way we defined our box, the orthographic transformation must always include a mirroring transform that mirrors the z coordinate.

Perspective Projection

The perspective transform is more complex, though it is more common in computer graphics. Recall that here, parallel lines are not parallel after projection, and objects that are further away are smaller.

Assume that the camera is located at the origin, and that we want to project a point p onto the plane $z = -d, d > 0$, yielding a new point $q = (q_x, q_y, -d)$. We can use similar triangles to derive the x component of q as

$$q_x = -d\frac{p_x}{p_z}$$

The expression for q_y is $q_y = -dp_y/p_z$ and finally $q_z = -d$. We can therefore construct a perspective projection matrix P_p as follows:

$$P_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix}$$

We can confirm that we obtain the required point by multiplying by p and divide the result by w.

As with the orthographic projection, there is a perspective transform that rather than actually projecting onto a plane, transforms the viewing volume into the canonical viewing volume. In this case, the viewing volume is assumed to start at $z = n$ and end at $z = f$ with $0 > n > f$. Recall that for a perspective projection, the viewing volume is in fact a *frustum* (a pyramid with without a top). The frustum is composed of a smaller rectangle at the top and a larger one at the bottom. Specifically, the smaller rectangle is located at $z = n$ and has corners located at (l, b, n) and (r, t, n).

As with the orthographic projection, the values (l, r, b, t, n, f) determine the frustum. The vertical field of view is determined by the angle between the top and bottom planes (determined by t and b). Intuitively, the larger this angle becomes, the more the camera will be able to "see". We can create asymmetric frusta by $r \neq -l$ or $t \neq -b$. These type of frusta are used for stereo viewing and for virtual reality.

The perspective transform matrix that maps the frustum into a unit cube is:

$$P_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

After applying this transform to a given point, we will end up with a new point $q = (q_x, q_y, q_z, q_w)^T$ where q_w is will be (generally) nonzero and not equal to 1. In order to obtain the final projected point, we will divide q_x, q_y, q_z, q_w by q_w. It is also important to note that this matrix will map $z = f$ to $+1$ and $z = n$ to -1. The matrix can be modified to allow for a far plane at infinity by replacing $f + n/f - n$ with 1 and $-2fn/f - n$ with $-2n$.

Regardless of which variant of the matrix is applied, the process is always the same. After the point is multiplied, clipping is performed by noticing that any points with coordinates greater than 1 or smaller than -1 are beyond the viewing frustum. Finally, the coordinates are normalized by dividing by w resulting in normalized device coordinates.

Similar to how we had to flip the z axis for the orthographic projection for OpenGL, we must do the same here. After this transformation, the near and far values can be entered as positive values where $0 < n' < f'$ as they would be traditionally presented to the user. The OpenGL version of the projection matrix is then:

$$P_p = \begin{bmatrix} \frac{2n'}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{1}$$

A simpler setup involves providing just the vertical field of view ϕ expressed as the ratio $a = w/h$ where $w \times h$ is the screen resolution. We would still have to provide n', f', and so we obtain

$$P_p = \begin{bmatrix} \frac{c}{a} & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Where

$$c = \frac{1}{\tan(\phi/2)}$$

Z-Fighting

Let us examine the results of the matrices that we just presented for perspective projection. Specifically, we note the following relationship between the z-coordinate of a point in view space p_e and the point in clipping space p_c after being multiplied by the projection matrix and divided by w. From eq. 1, we can obtain the following value for z_c:

$$z_c = \frac{\frac{(-f+n)z_e - 2fn)}{f-n}}{-z_e}$$

Note that the division by $-z_e$ comes as a result of the value for $w = -z_e$. If we examine this function, we will determine that it is rational (fractional) and establishes a non-linear relationship between z_c and z_e. This implies that there is a very high precision at the *near* plane, but very little precision at the *far* plane. If the range $[-n, -f]$ gets larger, this will cause depth problems known as *z-fighting*. What this means is that a small change of z_e near the far-plane will not affect the value of z_n. This issue arises due to the limited precision of the floating-point calculations that occur in the z-buffer, and unfortunately there is

no way around this problem. That being said, there are ways in which we can minimize this:

- The first option is to ensure that we never place objects too close to each other such that their geometries closely overlap. This option is possible if we have prior knowledge of the scene, but becomes very difficult to manage when we don't know the scene before hand, or if the scene is dynamic (there is motion that could change the positions of the objects between each frame).
- The second option is to move the near plane as far back as possible, thereby ensuring that small variations in z_e close to the far plane result in changes for z_c. This may be possible depending on the scene, but it could also introduce clipping artifacts.
- The third option is to increase the precision of the z-buffer from the normal 24-bits to 32. This does come at the cost of space and performance, though it is important to note that z-fighting *will* still occur.

Ultimately the best solution is entirely dependent on the application and what our requirements are.