Projection and View Frustums

David E. Johnson

I. Introduction

In mathematics, a projection reduces an N-dimensional vector space $\mathbb{R}^N$ to a subspace $\mathbb{W}$. This subspace is likely lower-dimensional than the original space. For example, a shadow is a projection of 3-space onto a 2D manifold. A projection matrix is an $N \times N$ square matrix that defines the projection, although other projection operators are valid. An example is the dot product of a vector with a unit vector u

$$proj_u l = (l \cdot u)u$$

which returns a vector on u with a length of l in the u direction. All these projections are linear transformations. Thinking of the shadow example, straight lines connect all points on the 3D object to the projected shadow. Since it is a linear transformation, the dot product can be encoded as a matrix as well.

$$\begin{bmatrix} u_x u_x & u_y u_x \\ u_x u_y & u_y u_y \end{bmatrix} \begin{bmatrix} l_x \\ l_y \end{bmatrix} = \begin{bmatrix} (u_x l_x + u_y l_y) u_x \\ (u_x l_x + u_y l_y) u_y \end{bmatrix} = (l \cdot u)u.$$

A projection matrix P has the property that

$$P^2 = P.$$

This makes intuitive sense since once the dimensionality of the original object has been reduced, projecting it again leaves it the same. Take this example, where P is

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

This projection takes an object from a 3D (x, y, z) space to a 2D (x, y) space. Performing matrix multiplication

$$P^2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = P,$$

so its projection property is verified.

Makers of maps, illustrators, architects, and engineers have developed conventions for a number of projections. Computer graphics regularly uses just a few of them. The proir example projection from 3D to a 2D is a basic operation in computer graphics, where a 3D virtual world must be mapped to the 2D screen. In virtual reality, an application developer must sometimes use projections that are non-standard for typical graphics programs.

A. Orthogonal Projections

An orthogonal projection takes points in space onto a viewing plane where all the motions of the points are orthgonal, or normal, to the viewing plane. The previous example transformation is an example of an orthogonal projection. Figure 1 shows a orthogonal projection of a virtual object onto the viewing plane. The box surrounding the viewing plane and virtual space represents the *viewing frustum* or *view frustum*. The view frustum represents the region of space that is projected onto the viewing plane. It defines the field of view of the virtual camera defining the projection.

In OpenGL, the view frustum shape is set on the `GL_PROJECTION` stack and the *glOrtho* command creates orthogonal projections.

```
glMatrixMode(GL_PROJECTION);
glOrtho( left, right, bottom, top, near, far);
```

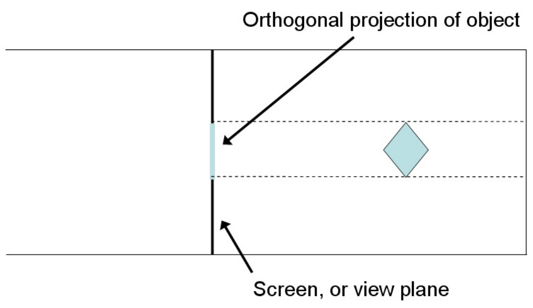

Fig. 1. The orthogonal projection of a virtual object on the screen moves each point of the object in the screen normal direction.

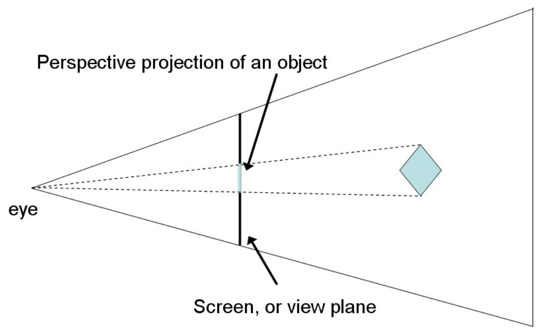

Fig. 2. The perspective projection of a virtual object on the screen scales the other coordinates by their distance from the eye.

The `near` and `far` arguments may seem unnecessary, but they are used to define the possible range of depths for the Z-buffer. Therefore, too large a range will result in Z-buffer precision problems.

OpenGL always has the camera pointing the same way (eye at the origin looking down negative z, with y up), so to look at something else, you move the world to get in front of the camera. This is done on the `GL_MODELVIEW` stack. The easiest way to do this is to use gluLookAt, which takes an eye point, a point the eye is looking at, and an up vector.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x, eye_y, eye_z, center_x, center_y, center_z, up_x, up_y, up_z);
```

I would like to emphasize that this occurs in the `GL_MODELVIEW` stack, rather than the `GL_PROJECTION` stack. Because it is helping to define the camera, the temptation is to put it on the wrong stack, since that is where camera fov and other properties are defined. But it really is just moving the world to the correct place. The consequences can be difficult to discern, but shows up when the distance from vertices to the camera must be found (fog, lighting, some texturing, Z-buffer precision).

B. Perspective Projections

Orthogonal projections are not realistic looking projections since the way we perceive the world is through perspective projection. In perspective projections, far away objects look smaller (see Figure 2). The basic matrix setup for perspective transformations is

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 1/d & 0\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} = \begin{bmatrix}x\\ y\\ z\\ z/d\end{bmatrix} = \begin{bmatrix}xd/z\\ yd/z\\ d\\ 1\end{bmatrix}$$

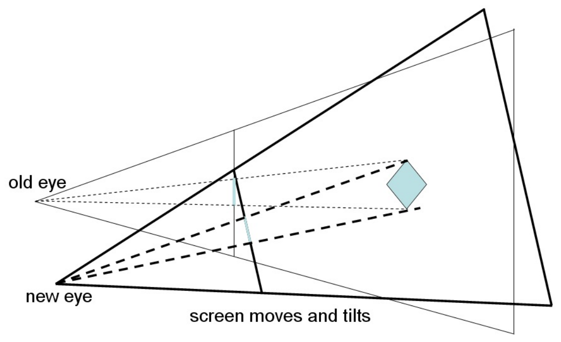

Fig. 3. Moving the eye point relative to the screen with gluPerspective also tilts the presumed position of the physical screen.

Notice how the x and y values are scaled by the ratio of the projection plane distance, d, to the object distance. It is easy to see by similar triangles that x and y should shrink by that ratio.

The easiest way to specify a perspective transform in OpenGL is to use gluPerspective.

```
glMatrixMode(GL_PROJECTION);
gluPerspective( fieldOfView, aspectRatio, near, far );
```

The fieldOfView value is the view angle in degrees for the y direction, the aspectRatio determines the relative x direction field of view, and near and far are the clip planes. Note that the aspectRatio should match the actual aspect ratio of your view window.

However, gluPerspective is not adequate to describe viewing for head-tracked VR. Imagine a large screen centered in a viewer's gaze and 1 meter away. The screen takes up about 50 x 40 degrees of the viewer's field of view. Then the perspective matrix could be specified by

```
glMatrixMode(GL_PROJECTION);
gluPerspective( 40, 50.0/40.0, 0.5, 2.0 );
```

Now, if the viewer were head-tracked and looking at a computer monitor, then gluLookAt modifies what part of the virtual world gets moved inside this frustum. But it also creates a tilt to the projection plane (Figure 3), since the volume defined by gluPerspective doesn't change. Because the physical screen cannot move along with this changing view, another approach is needed.

A more general command to specify a perspective transform looks more like the glOrtho command. It is glFrustum, which is called like

```
glMatrixMode(GL_PROJECTION);
glFrustum( left, right, bottom, top, near, far );
```

glFrustum is more general because the left/right and top/bottom need not be symmetric around the z-axis. This is needed for a head-tracked view frustum. It sets up a view that maps (left, bottom, -near) to the lower left of the screen and (right, top, -near) to the upper right of the screen when the eye is at (0,0,0) looking down the -z axis.

When the head position is tracked, the desired behavior is shown in Figure 4. In this image, the eye is free to move around space and change its orientation. The first thing to notice is that the desired projection on the screen does not change with view direction. That is, whether the screen is being viewed full on or in the periphery, the center of projection is the same. The view frustum defines how much of the physical field of view the display window is filling, it has nothing to do with where the eye is looking. The second thing to notice is that the view frustum from the eye to the screen is asymmetric relative to the viewing plane normal. Each time the tracker updates the head position, the view frustum must be adjusted. How can this be specified?

Imagine the standard OpenGL camera sitting at the origin and looking down the negative z-axis. Since glFrustum only specifies scalar left-right-bottom-top values, the viewing plane *must* remain in an xy-plane and is offset in z by the given near value. Again, think of the camera looking along the z-axis and a viewing plane orthogonal to the z-axis, translated around as specified by the left-right-bottom-top-near values.

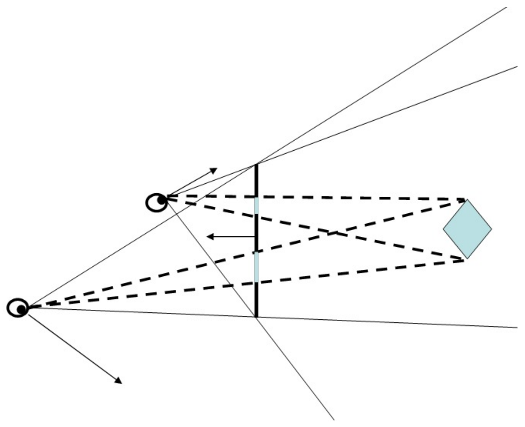

Fig. 4. Moving the eye point relative to the screen and the desired projections of the object.

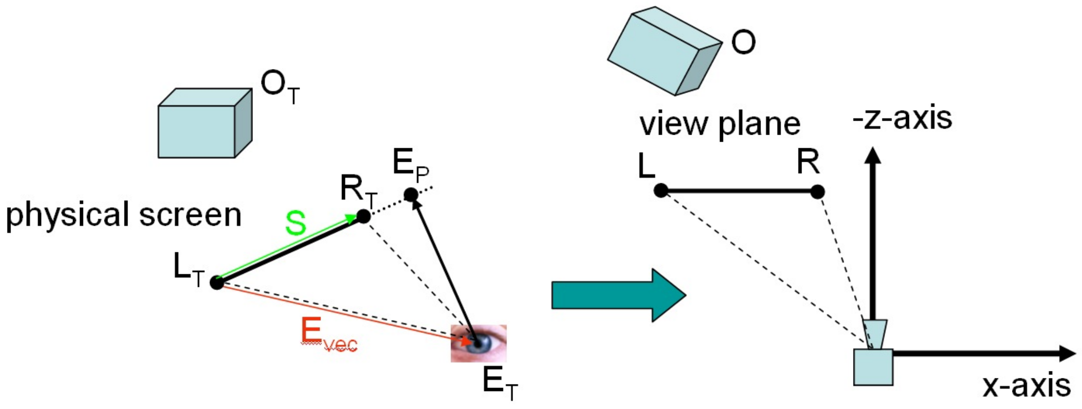

Fig. 5. The view frustum is setup by projecting the eye onto the screen, then finding the coordinates of the corners of the window in that coordinate frame.

Thus, the goal of our glFrustum call is to recreate a virtual setup where the viewing plane is positioned relative to the OpenGL camera in the same way that the physically located OpenGL window is located relative to our tracked eyeball. Clearly, in order to do this, the physical window's location must be known in the tracker coordinate system, so at some point, the tracker must be used to measure points on the computer monitor. And given that the OpenGL camera is fixed at the origin, the coordinates of the viewing plane are just the changes in coordinates between the eye in tracker coordinates and the display window in tracker coordinates.

Let us work through an example demonstrating this approach. To leave a little for you to think about, this example will be done in a two-dimensional world.

In this example, (see Figure 5), the tracked eye is at E_T in the tracker coordinate system. The physical screen has been measured with the tracker and found to have corners L_T and R_T. Furthermore, given the screen coordinates, an object has been modeled to lie behind this screen in the tracker coordinate system at O_T.

We know that eventually the tracked eye will become the OpenGL camera. Around the eye, we want to build a local coordinate system that will become the -z-axis of the camera. Looking at the figure, it should be clear that the vector from E_T to its projection on the plane of the physical screen is that vector. To find E_P, we define a vector along the screen, S, and find the projection of E_T on it.

$$\begin{aligned} S &= R_T - L_T \\ E_{vec} &= E_T - L_T \\ E_P &= L_T + (E_{vec} \cdot \frac{S}{||S||})\frac{S}{||S||} \end{aligned}$$

Now the view frustum offsets L and R are just the distances from L_T and R_T to E_P. (We can avoid projecting the vectors between these points onto the new coordinate system since we know that these points lie in the plane orthogonal to the new local coordinate system z-axis.)

$$\begin{aligned} L &= ||L_T - E_P|| \\ R &= ||R_T - E_P|| \end{aligned}$$

The z coordinate of the view plane is just the distance from the eye to its projection, or

$$near = ||E_T - E_P||$$

giving us all we need to make our 2D glFrustum call.

```
glMatrixMode(GL_PROJECTION);
gl2DFrustum( L, R, near, 20.0 );
```

However, the modeled object O_T must also be positioned. Recall that only the tracked position of the eye is important, not the direction. So what "at" point should be used? Well, the gluLookAt function moves the world so that the "at" point lies along the z-axis. From the figure, E_P clearly fulfills this functions, so it should be used as the "at" point in the call.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glu2DLookAt( E_T, E_P, up);
```

Together, this process moves the eye to the default OpenGL position and moves the world to locate it correctly for viewing. Given perfect tracking, the virtual object should appear to float in space as you move your head, even without stereo.

Here are some things to think about:

- In perspective projection, what happens to the projected size of an object when the field of view grows/shrinks?
- Using a head-tracked viewing frustum, what happens to the field of view when you move further from the display?
- Can you reconcile your answers above?