Perspective Projections

George Francis

February 3, 2001

1 Introduction

What follows may also be considered a summary of Chapter 7 in the Version 1.0 of the Guide on coordinate transformations. Naturally, we take a somewhat different view of the underlying 3-dimensional geometry than the authors of the Guide. Since it is, however, advisable to use the Iris graphics library whenever possible, we shall concentrate on explaining their primitives as well.

2 Homogeneous Coordinates

Here is what affine geometry looks like in homogeneous coordinates. Points in 3-space are represented by four single precision (32 bit) floating point numbers, \((x, y, z, w)\). The last transformation done by the Iris Geometry Pipeline is to truncate the ratios \((x/w, y/w)\) to lie between \(-1.0\) and \(+1.0\), so that the point fits into whatever window\(^1\) it is to be displayed in. Think to these values as fractions of the window width and height.

The ratio \(z/w\) is not discarded. It too is truncated and then used for the \(z\)-buffer. The \(z\)-buffer holds the depth information for each pixel, so that a further away pixel is obscured\(^2\) by a nearer one. We shall ignore the clipping to the interval \([-1.0, +1.0]\) until later, and adjoin the ratio \(z/w\), so that we may speak of the point \((x/w, y/w, z/w)\) in Cartesian 3-space. The same point, \((X, Y, Z)\) in 3-space has many quadruples, \((x, y, z, w)\), representing

\(^1\)Unless specifically altered, the viewport is coextensive with the window. We generally mean the former when saying the latter out of habit.

\(^2\)Or otherwise blended on Iris’s with alpha-blending buffers.
it in homogeneous coordinates, but only one with \( w = 1 \). Thus, at the
beginning of the pipeline there is a stream of points \((x, y, z, 1)\), which are
subsequently transformed.

One important reason for using homogeneous coordinates is that an affine
transformation \([Q, a]\), acts on a point as a 4x4 matrix

\[
[x, y, z, w] \begin{bmatrix}
q_{00} & q_{01} & q_{02} & 0 \\
q_{10} & q_{11} & q_{12} & 0 \\
q_{20} & q_{21} & q_{22} & 0 \\
a_0 & a_1 & a_2 & 1
\end{bmatrix} = [x' + a_0 w, y' + a_1 w, z' + a_2 w, w]
\]

where

\[
[x, y, z] \begin{bmatrix}
q_{00} & q_{01} & q_{02} \\
q_{10} & q_{11} & q_{12} \\
q_{20} & q_{21} & q_{22}
\end{bmatrix} = [x', y', z']
\]

is the effect of applying the linear transformation \( Q \) to \([x, y, z]\).

Now divide by \( w \), and note that in Cartesian coordinates we have mapped

\[
\left[\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right] \rightarrow \left[\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right]Q + [a_0, a_1, a_2],
\]

which is exactly what the affine transformation \([Q, a]\) is supposed to do.

### 3 Simple Affine Transformations

The primitives for the affine group are translation, rotation about each of the
principal axes, and scaling the three axes. The corresponding homogeneous
matrices, and the calls to the graphics library generating the matrix, are as
follows:

**Translation:**

\texttt{translate}(a_0, a_1, a_2) ;

\[
[I, a] = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
a_0 & a_1 & a_2 & 1
\end{bmatrix}
\]
Principal rotation

\[ \text{rotate}( \text{th} , 'z' ); \text{ in deci-degrees}^{4} \text{ about the z-axis.} \]

\[ [R(\theta, \hat{z}), 0] = \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ \text{rotate}( \text{th} , 'x' ); \text{ in deci-degrees about the x-axis} \]

\[ [R(\theta, \hat{x}), 0] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ \text{rotate}( \text{th} , 'y' ); \text{ in deci-degrees about the y-axis} \]

\[ [R(\theta, \hat{y}), 0] = \begin{bmatrix} -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 1 & 0 & 0 \\ \cos \theta & 0 & \sin \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Scaling and Mirroring

\[ \text{scale}(k_0, k_1, k_2); \]

\[ [D(k_0, k_1, k_2), 0] = \begin{bmatrix} k_0 & 0 & 0 & 0 \\ 0 & k_1 & 0 & 0 \\ 0 & 0 & k_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Of course, none of the scale factors may be zero. Negative scale factors correspond to reflections in the principal coordinate planes.

4 Compound Affine Transformations

There are two important compound affine transformations

---

3A positive angle of rotation effects a cyclic twisting of space about the axis. Point your right thumb along the axis, your fingers wrap about the axis in the direction of the rotation. A right-handed screw advances in this manner.

4One of the more absurdly useful units, a tenth of a degree equals \(\pi/1800\), and thus \(\text{th} = \theta \times 1800/\pi\).
polarview($\rho, \alpha, \beta, \tau$); and lookat($e_0, e_1, e_2, o_0, o_1, o_2, \tau$);

where angles $\alpha, \beta$, and $\tau$ are calibrated in tenth of a degree, and the rest are floating point variables. The affine transformation associated\(^\text{5}\) to polarview() is the one required to effect the movement of the camera away from the origin and to a point on a sphere of radius $\rho$, but still pointing toward the origin\(^\text{6}\). The camera is first moved along the x-axis to the equator. Then along the equator an angle of $\alpha/10$ degrees. This angle is called the azimuth. Then it moves along the meridian towards the northpole a (signed) angle of inclination $\beta$. Finally, the camera is rotated about the line of sight by the twist angle $\tau$.

In the case of lookat(), the camera is moved from the origin to the eye point $e$ and pointed towards the object point $o$, leaving the remaining degree of freedom to a twist angle.

Exercise 1. Demonstrate conclusively, or disprove by a counterexample, that the affine transformations generated by these two compounds may be factored as claimed in the Guide thus:

\[
\text{polarview}(\rho, \alpha, \beta, \tau) : [\begin{vmatrix} I & e \\ R(-\alpha, \hat{z}) & R(-\beta, \hat{x})R(-\tau, \hat{z}), -\rho \hat{z} \end{vmatrix} = [P, r],
\]

\[
\text{lookat}(e_0, e_1, e_2, o_0, o_1, o_2, \tau) : [I, e][R(\theta_1, \hat{y})R(\theta_0, \hat{x})R(\theta_2, \hat{z}), 0] = [Q, eQ]
\] .

Exercise 2 The three angles, $\theta_i$, in the factorization of the rotational part $Q$ of lookat() depend on the two points $e$ and $o$. Find this dependence.

Experimentation reveals which features of polarview() and lookat() are useful to the geometer and which can be more efficiently written out explicitly. Notable is the fact that in the first the object is rotated and translated from the viewer in one operation. The three angles are not independent. Since the mouse has only two degrees of freedom, it is useful to couple it with two of the three angles, and leave the third one fixed at zero.

In the second case, the rotation is performed after the translation. So it is useful when this combination is desired. The fact that distance from the object observed is not constant makes lookat() useful mostly for flight simulators and other “fly-by” animations. There seems to be little reason to use these functions for mathematical simulations other than compactness of the program code.

\(^{\text{5}}\) As we shall see later, the function call is dynamic in the sense that the current projection matrix is multiplied, on the left, by this affine matrix.

\(^{\text{6}}\) The radius $\rho$ is also truncated to $+1$ and so cannot be used for “zooming” through the origin.
5 Linear Perspective

A great deal of mystery and confusion may be avoided if we accept Albrecht Dürer’s pictorial paradigm of how to draw in perspective. The painter looks through an eyepiece which remains in one place. As he looks at a point on the object, a horizontal wire and a vertical wire across the picture frame are moved so as to cross on the line of sight. Attached to the frame on hinges is the picture screen. The window is closed, a point is marked on the screen where the wires cross. Once sufficiently many reference points are marked, the painter completes the picture in the usual manner.

In terms of Cartesian coordinates, let \((x, y)\) measure the horizontal and vertical displacement of a point from the central line of sight which emanates from the eye and is perpendicular to the picture plane. Two distances affect the proportional change in size of these coordinates on the picture plane, namely the *focal* distance \(f\) from the eye to the picture, and the distance \(r\) from the picture to the object. A simple proportion argument on right triangles reveals that the *foreshortening factor* is \((1 + \frac{r}{f})^{-1}\). Thus, a natural choice for the third coordinate is to let \(z = r\), which places the eye at \((0, 0, -f)\), and the objects \((x, y, z)\) behind the picture plane have positive \(z\)-coordinates. This yields a left-handed Cartesian coordinate system. To keep a right-handed coordinate system some people prefer a down-pointing \(y\)-axis rather than having to place visible objects at negative \(z\)-coordinates.

It is also convenient to use the reciprocal, \(\rho = 1/f\), of the focal distance, so that orthographic projection (no perspective) is obtained by setting \(\rho = 0\). Iris geometers prefer to look into the negative \(z\)-direction. Thus \(r = -z\) and the matrix representation in homogeneous coordinates becomes

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & -\rho \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
= 
\begin{bmatrix}
x \\
y \\
z - \rho z \\
1
\end{bmatrix}
\]

6 Projection Transformations

There are a variety of reasons, some good, some bad, why the Silicon Graphics geometers did not choose to use this model for perspective. The eye of their observer, the camera, *always* remains at the origin. Objects built near the origin must first be translated to some location with negative \(z\)-coordinates.
In fact, the picture plane is set up one focal distance, called \textit{near}, away into the negative \( z \)-direction. The picture plane, \( z + \text{near} = 0 \), is also the \textit{near clipping plane}. The other clipping plane, a distance \textit{far} away from the origin, is thus located at \( z = -\text{far} \). Conceptually simple, this paradigm makes for horrible algebra when it comes to designing the projection matrix. There are, in fact, four choices.

\textbf{Orthographic}

\begin{equation*}
\text{ortho2}(x_0, x_1, y_0, y_1);
\begin{bmatrix}
\frac{1}{\sigma_x} & 0 & 0 & 0 \\
0 & \frac{1}{\sigma_y} & 0 & 0 \\
0 & 0 & -1 & 0 \\
-\bar{x} & -\bar{y} & \bar{z} & 1
\end{bmatrix}
\end{equation*}

where we borrow from statistical notation for mnemonic purposes:

\( \bar{x} = \frac{x_1 + x_0}{2} \quad \sigma_x = \frac{x_1 - x_0}{2} \)
\( \bar{y} = \frac{y_1 + y_0}{2} \quad \sigma_y = \frac{y_1 - y_0}{2} \)

Recall that the fractions \( x/w, y/w \) are also clipped to lie between \( \pm 1 \). Thus the first two columns of this matrix have the effect of first moving the rectangle \([x_0, x_1] \times [y_0, y_1]\) into the standard 2 square. In other words, this is the location and shape of Dürer’s picture frame. The \(-1\) on the diagonal has the effect of reversing the orientation in the \( z \)-direction. Since this is used only for the \( z \)-buffer, and the \( z \)-buffer has this direction by default, \texttt{ortho2()} can also be used as a quick and dirty orthographic projection from 3-space to a 2-plane.

\textbf{3-D Orthographics}

The general, 3-dimensional orthographic projection call is

\begin{equation*}
\text{ortho}(x_0, x_1, y_0, y_1, \text{near, far});
\begin{bmatrix}
\frac{1}{\sigma_x} & 0 & 0 & 0 \\
0 & \frac{1}{\sigma_y} & 0 & 0 \\
0 & 0 & -\frac{1}{\sigma_z} & 0 \\
-\bar{z} & -\bar{y} & \bar{z} & 1
\end{bmatrix}
\end{equation*}

Here, writing \( z_0 = -\text{far}, z_1 = -\text{near} \), we have

\( \bar{z} = \frac{z_1 + z_0}{2} = -\frac{\text{far} + \text{near}}{2} \)
\[ \sigma_z = \frac{z_1 - z_0}{2} = \frac{\text{far} - \text{near}}{2} \]

**Exercise 3.** Show experimentally\(^7\) that \(\text{ortho}(-1,1,-1,1,1,-1)\) is associated with the \(4 \times 4\) identity matrix and that

\[ \text{ortho}(-1,1,-1,1,1,-1) = \text{ortho2}(-1,1,-1,1) . \]

**Exercise 4** Prove that the Iris Geometry Pipeline really does what is claimed above regarding the near and far clipping plane. You should check that the effect of the matrix on \((x, y, z, 1)\) maps the cube

\[ [x_0, x_1] \times [y_0, y_1] \times [-\text{far}, -\text{near}] \]

into

\[ [-1, +1] \times [-1, +1] \times [+1, -1] \]

Note the reversal of orientation in the \(z\)-direction.

There are two perspective primitives, one of which uses the same parameters as \(\text{ortho}\). Its effect is to project to the *front face* the clipping box at \(z = -\text{near}\) what can be seen from the origin. In effect, Dürer’s picture frame becomes a window on the world, but only up to the rear clipping plane at \(z = -\text{far}\).

**Perspective**

\[
\text{window}(x_0, x_1, y_0, y_1, \text{near}, \text{far});
\]

\[
\begin{bmatrix}
\text{near} \\ \sigma_x \\
0 \\ \text{near} \\
\bar{x} \\ \sigma_x \\
0 \\ 0 \\
0 \\ \text{near} \times \text{far} \\
\sigma_z \\
\end{bmatrix}
\]

The other uses a *viewing angle* of \(2\phi\) modified by an *aspect ratio* \(\alpha\). This cannot be used for oblique perspective.

---

\(^7\)Experiments like this are essential for maintaining temporal integrity of software on the Iris because mystifying changes occur in successive editions of the graphics library.
perspective\((2\phi, \alpha, \text{near}, \text{far})\);

\[
\begin{bmatrix}
(\alpha \tan \phi)^{-1} & 0 & 0 & 0 \\
0 & (\tan \phi)^{-1} & 0 & 0 \\
0 & 0 & \frac{1}{\sigma_z} & -1 \\
0 & 0 & \frac{\text{near} \times \text{far}}{\sigma_z} & 0
\end{bmatrix}
\]

Exercise 5 Justify the algebra for the above matrices.

Note that there is no easy way to alter the focal distance interactively. Both object and clipping box must be moved back and forth in the visible halfspace. The size of the image must be adjusted dynamically as well to keep the object roughly the same size. The near and far clipping conditions require an oddly non-intuitive approach that leads to considerable experimentation before your program works the way you want it to.

7 The Iris Affine Projector Pipeline

When working with affine transformations it is good to keep one coordinate system fixed. A natural candidate for this is related to the observer’s own position, with the origin at his head, and principal axes horizontal, vertical and straight ahead. The Iris coordinates come close to this. The origin is the screen center. The first two axes are to the right and up, that is, East and North, as you look at the screen. These are the x-axis and the y-axis, respectively. It will be convenient to also refer to them as the 0-axis and the 1-axis.

To maintain a right handed coordinate system, the third axis or z-axis must point at the observer. It comes out to the computer. In particular, the z-coordinates of visible points are all negative, and this is the source of some confusion until one gets used to it.

The other source of considerable confusion is the reverse order in which affine transformation must be called in a program. The Iris maintains one projection matrix on the top of the projection matrix stack\(^8\) Every point plotted is multiplied into this matrix on the left, and the result is then sent to the system to be displayed in a window on the screen. The only control over what happens on the other side of the projector is via the

\[
\text{viewport(left, right, bottom, top)};
\]

---

\(^8\)This applies to the single-matrix mode. Multimatrix modes are discussed elsewhere.

Renaissance Experimental Laboratory and UIMATH.graßXlab.
in pixel coordinates, which is discussed in greater detail later.

The first graphics call should therefore place some initial matrix on top of the stack. Hand built matrices should be defined like this, for example.

```c
float mater[4][4], pater[4][4];
```

The entries may be assigned in the standard C-fashion.

```c
loadmatrix(mater);  makes a copy of the named matrix on top of the projector.
getmatrix(pater);    makes a copy of the projector in a named matrix.
multmatrix(mater);   multiplies the projector on the left.
```

The above sequence of three commands would leave the square of the original matrix `mater` on the projector, and another copy of it in `pater`.

To summarize, a new matrix, Π, is installed in the projector by any one of these calls:

```c
ortho2(); ortho(); window(); perspective(); loadmatrix();
```

Hence it is never correct to use more than one of these in building a projector. Other functions generating matrices multiply the current projector by their associated affine matrix, A, on the left, that is between the points to be plotted and Π. In particular, a sequence of affine transformations generated by $A_i$ from among (in this order $A_0, A_1, A_2, ... , A_n$) will have the effect of modifying Π thus

$$Π ← A_nA_{n-1}...A_0Π.$$ 

There are also three stack operations which are used for building affine transformations. To produce a product of affine transformations in this way, you should first load the identity matrix, easily defined thus

```c
for(ii=0;ii<4;ii++)for(jj=0;jj<4;jj++)id[ii][jj]=(ii==jj)?1.:0.;
loadmatrix(id); 
```

Of course, the command `ortho2(-1.,1.,-1.,1.,1.,-1.)` may be quicker but more confusing. After the appropriate left-multiplications, a final `getmatrix()` command is used to store the compound for later use.
There is a shortcut for this procedure, which is useful if an animation loop needs to place objects in various positions in the scene. The

\begin{verbatim}
pushmatrix(); popmatrix();
\end{verbatim}

pair duplicates the current projector on top, and drops it again respectively. In stack language, these are \textit{dup} and \textit{drop} operations.