Computer Graphics

John E. Howland
Department of Computer Science
Trinity University
715 Stadium Drive
San Antonio, Texas 78212-7200
Voice: (210) 999-7364
Fax: (210) 999-7477
E-mail: jhowland@Ariel.CS.Trinity.Edu
Web: http://WWW.CS.Trinity.Edu/~jhowland/

June 23, 2009

Abstract

An introduction to some of the key ideas in computer graphics is given. Modeling, 2D and 3D viewing, transformations and related ideas from linear algebra are presented.

Subject Areas: Computer Graphics.
Keywords: 2D Viewing, 3D Viewing, modeling, linear algebra.

1 Introduction

Computer graphics deals with the problem of image synthesis. Given a model (usually mathematically based) the problem of computer graphics is to produce realistic image data which may be viewed on a graphics display device. The process of producing the image data from the scene model is called *rendering*. That images are synthesized from mathematical models implies that computer graphics is a mathematcally based subject. Image synthesis involves the physics of light, properties of materials, etc. Animated imagery involves simulation theory, finite element analysis, kinematics, sampling theory and other mathmatically based fields. The study of computer graphics necessarily involves the study of many areas of mathematics. In the following sections we give an elementary view of some of these topics. Students who have an interest in computer graphics should study as much mathematics as possible.

The reverse problem of starting with image data and recovering information is called *image processing*.

2 2D Viewing

We now consider the problem of representing 2D graphics images which may be drawn as a sequence of connected line segments. Such images may be represented as a matrix of 2D points $\begin{bmatrix} x_i & y_i \end{bmatrix}$.

The J programming notation [Hui 2001] is used to describe the viewing transformations and data object representations.

For example:

```
   [ square =: 5 2 $ 0 0 10 0 10 10 0 10 0 0
 0  0
10  0
10 10
 0 10
 0  0
```

represents the square shown in Figure 1

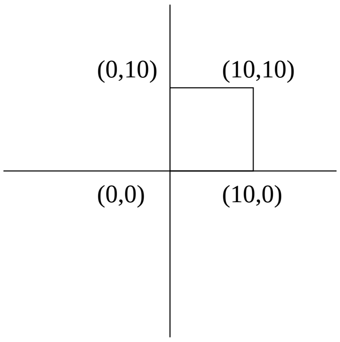

Figure 1: A Square

The idea behind this representation is that the first point represents the starting point of the first line segment drawn while the second point represents the end of the first line segment and the starting point of the second line segment. The drawing of line segments continues in similar fashion until all line segments have been drawn. A matrix having $n + 1$ points describes a figure consisting of n line segments. It is sometimes useful to think of each pair of consecutive points in this matrix representation,

$$\begin{bmatrix} x_{i-1} & y_{i-1} \\ x_i & y_i \end{bmatrix}$$

as as a vector so that the square shown in Figure 1 is the result of drawing the vectors shown in Figure 2.

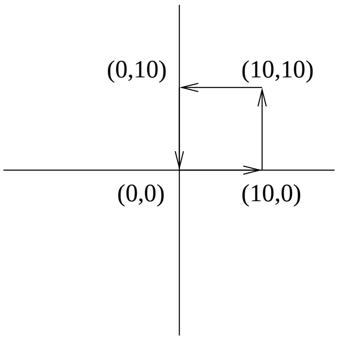

Figure 2: The Vectors in A Square

3 Rotation

Suppose we wish to rotate a figure around the origin of our 2D coordinate system. Figure 3 shows the point (x, y) being rotated θ degrees (by convention, counter clock-wise direction is positive) about the origin.

The equations for changes in the x and y coordinates are:

$$\begin{aligned} x' &= x \times cos(\theta) - y \times sin(\theta) \\ y' &= x \times sin(\theta) + y \times cos(\theta) \end{aligned} \tag{1}$$

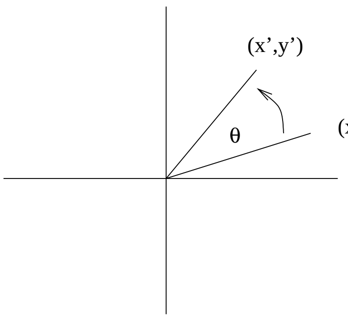

Figure 3: Rotating a Point About the Origin

If we consider the coordinates of the point (x, y) as a one row two column matrix $\begin{bmatrix} x & y \end{bmatrix}$ and the matrix

$$\begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix}$$

then, given the J definition for matrix product, `mp =: +/ . *`, we can write Equations (1) as the matrix equation

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \mathtt{mp} \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \tag{2}$$

We can define a J monad, `rotate`, which produces the rotation matrix. This monad is applied to an angle, expressed in degrees. Positive angles are measured in a counter-clockwise direction by convention.

```
   rotate =: monad def '2 2 $ 1 1 _1 1 * 2 1 1 2 o. (o. y.) % 180'
   rotate 90
 0 1
_1 0
   rotate 360
           1 _2.44921e_16
2.44921e_16           1
```

We can rotate the square of Figure 1 by:

```
   square mp rotate 90
  0  0
  0 10
_10 10
_10  0
  0  0
```

producing the square shown in Figure 4.

4 Scaling

Next we consider the problem of scaling (changing the size of) a 2D line drawing. Size changes are always made from the origin of the coordinate system. The equations for changes in the x and y coordinates are:

$$\begin{aligned} x' &= x \times S_x \\ y' &= y \times S_y \end{aligned} \tag{3}$$

As before, we consider the coordinates of the point (x, y) as a one row two column matrix $\begin{bmatrix} x & y \end{bmatrix}$ and the matrix

$$\begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

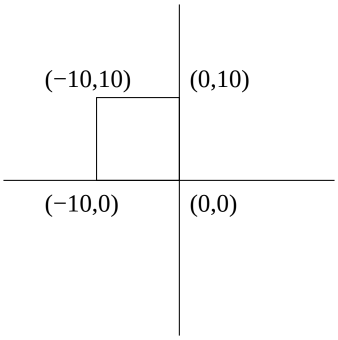

Figure 4: The Square, Rotated 90 Degrees

then, we can write Equations (3) as the matrix equation

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \texttt{mp} \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \tag{4}$$

We next define a J monad, `scale`, which produces the scale matrix. This monad is applied to a list of two scale factors for x and y respectively.

```
   scale =: monad def '2 2 $ (0 { y.),0,0,(1 { y.)'
   scale 2 3
2 0
0 3
```

We can now scale the square of Figure 1 by:

```
   square mp scale 2 3
 0  0
20  0
20 30
 0 30
 0  0
```

producing the rectangle shown in Figure 5.

5 Translation

The third 2D graphics transformation we consider is that of translating a 2D line drawing by an amount T_x along the x axis and T_y along the y axis. The translation equations may be written as:

$$\begin{aligned} x' &= x + T_x \\ y' &= y + T_y \end{aligned} \tag{5}$$

We wish to write the Equations 5 as a single matrix equation. This requires that we find a 2 by 2 matrix,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

such that $x \times a + y \times c = x + T_x$. From this it is clear that $a = 1$ and $c = 0$, but there is no way to obtain the T_x term required in the first equation of Equations 5. Similarly we must have $x \times b + y \times d = y + T_y$. Therefore, $b = 0$ and $d = 1$, and there is no way to obtain the T_y term required in the second equation of Equations 5.

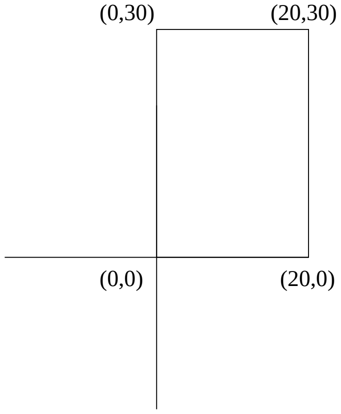

Figure 5: Scaling a Square

5.1 Homogenous Coordinates

From the above argument we now see the impossibility of representing a translation transformation as a 2 by 2 matrix. What is required at this point is to change the setting (2D coordinate space) in which we phrased our original problem. In geometry, when one encounters difficulty when trying to solve a problem in n space, it is customary to attempt to re-phrase and solve the problem in $n+1$ space. In our case this means that we should look at our 2D problem in 3 dimensional space. But how can we do this? Consider that, given a point (x, y) in 2 space, we map that point to $(x, y, 1)$. That is, we inject each point in the 2D plane into the corresponding point in 3 space in the plane $z = 1$. If we are able to solve our problem in this plane and find that the solution lies in the plane $z = 1$, then we may project this solution back to 2 space by mapping each point $(x, y, 1)$ to (x, y).

To summarize, we inject the 2D plane into 3 space by the mapping

$$(x, y) \to (x, y, 1) \tag{6}$$

Then we solve our problem, ensuring that our solution lies in the plane $z = 1$. Our final answer is obtained by the projection of the plane $z = 1$ on 2 space by the mapping

$$(x, y, 1) \to (x, y) \tag{7}$$

This process is referred to as using *homogeneous* coordinates. In the context of our problem (finding matrix representations of rotation, scaling and translation transformations) we must inject our 2D line drawings into the plane $z = 1$. In J we do this by using `stitch`, `,.`.

```
   square ,. 1
 0  0 1
10  0 1
10 10 1
 0 10 1
 0  0 1
```

We now must rewrite the Equations 5 as

$$\begin{aligned} x' &= x + T_x \\ y' &= y + T_y \\ z' &= z \end{aligned} \tag{8}$$

Consider the 3 by 3 matrix

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

We now see that the Equations 8 may be written as the matrix equation

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix} \tag{9}$$

We define the J monad `translate`, which is applied to a list of two translate values $T_x \quad T_y$.

```
   translate =: monad def '3 3 $ 1 0 0  0 1 0  , y. , 1'
   translate 10 _10
 1   0 0
 0   1 0
10 _10 1
```

We translate the square of Figure 1 by

```
   (square ,. 1) mp translate 10 _10
10 _10 1
20 _10 1
20   0 1
10   0 1
10 _10 1
```

5.2 Efficiency of Transformations

Notice that the translate matrix (having a last column 0 0 1) always produces a result which lies in the plane $z = 1$. We can perform the translation operation and project the result back on the 2D plane (saving computation time by not doing unnecessary multiplications and additions) by

```
   (square ,. 1) mp 3 2 {. translate 10 _10
10 _10
20 _10
20   0
10   0
10 _10
```

producing the translated square shown in Figure 6

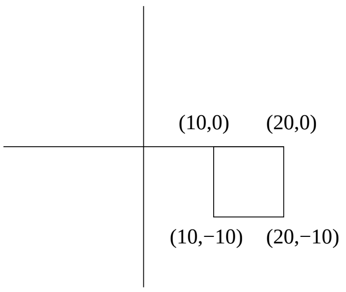

Figure 6: Translating a Square

6 Scaling and Rotation Using Homogeneous Coordinates

We want to be able to combine sequences of rotations, scaling and translations together as a single 2D graphics transformation. We accomplish this by simply multiplying the matrix representations of each transformation using matrix multiplication. However, to do this, we must go back and rewrite the Equations 1 and 3 as the following:

$$\begin{aligned} x' &= x \times cos(\theta) - y \times sin(\theta) \\ y' &= x \times sin(\theta) + y \times cos(\theta) \\ z' &= z \end{aligned} \tag{10}$$

$$\begin{aligned} x' &= x \times S_x \\ y' &= y \times S_y \\ z' &= z \end{aligned} \tag{11}$$

Similarly we rewrite the matrix Equations 2 and 4 as:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{12}$$

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{13}$$

We extend our earlier J definitions of `rotate` and `scale` to the homogenous coordinate system.

```
   rotate =: monad def '((2 2 $ 1 1 _1 1 * 2 1 1 2 o. (o. y.) % 180),.0),0 0 1'
   rotate 180
_1  0 0
 0 _1 0
 0  0 1
   (square ,. 1) mp 3 2 {. rotate 180
  0   0
_10   0
_10 _10
  0 _10
  0   0
```

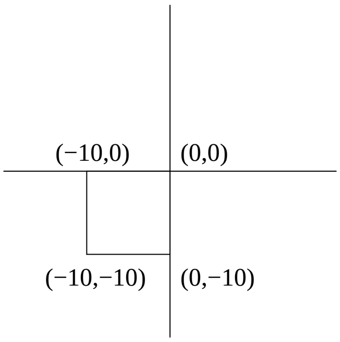

Figure 7: Rotating a Square 180 Degrees

```
   scale =: monad def '3 3 $ (0 { y.), 0 0 0 , (1 { y.), 0 0 0 1'
      scale 2 3
2 0 0
0 3 0
0 0 1
   (square ,. 1) mp 3 2 {. scale 2 3
 0  0
20  0
20 30
 0 30
 0  0
```

Figure 5 shows the resulting scaled square.

7 Combining Transformations

We can now combine together two transformations to form a single graphics operation. For example, suppose we wish to first rotate an object 90 degrees and then scale the object by 2 along the x axis.

The rotation would be expressed as:

```
    [r =: rotate 90
 0 1 0
_1 0 0
 0 0 1
```

Then the scaling operation would be expressed as:

```
   [s =: scale 2 1
2 0 0
0 1 0
0 0 1
```

Applying these operations to the square, we have:

```
   (((square ,. 1) mp 3 2 {. r) ,. 1) mp 3 2 {. s
  0  0
  0 10
_20 10
_20  0
  0  0
```

7.1 Efficiency of Operations

However, notice that

```
   (square ,. 1) mp 3 2 {. r mp s
  0  0
  0 10
_20 10
_20  0
  0  0
```

produces the same result using far fewer multiplications and additions. Figure 8 shows the rotated and scaled square.

We are allowed to perform the matrix multiplications of `r` and `s` before multiplying by `square ,. 1` because matrix multiplication is associative.

Be careful! Matrix multiplication is *not* commumative.

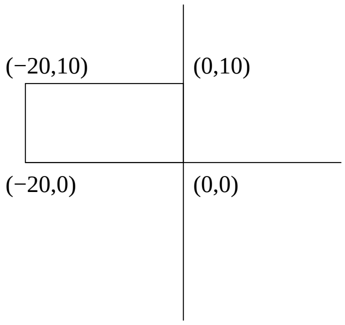

Figure 8: Rotated and Scaled Square

```
   r mp s
 0 1 0
_2 0 0
 0 0 1
   s mp r
 0 2 0
_1 0 0
 0 0 1
```

This means we must be careful about the order of application of graphics transformations.

One might be concerned about whether or not multiplying rotation, scaling and/or translation matrices produces a transformation which leaves our 2D lines in the plane $z = 1$. We can answer this question by observing that each of these matrices has a last column of $0 \quad 0 \quad 1$. Hence, when multiplying any two of these matrices, the product matrix has a last column of $0 \quad 0 \quad 1$.

8 Rotating an Object About a Point

As a final example, suppose we wish to rotate the square of Figure 1 90 degrees about its upper right corner. We must first translate the point $(10, 10)$ to the origin. This is the matrix

```
   translate _10 _10
  1   0 0
  0   1 0
_10 _10 1
```

Then we must rotate 90 degrees

```
   rotate 90
 0 1 0
_1 0 0
 0 0 1
```

Finally, we translate the square back with the matrix

```
   translate 10 10
 1  0 0
 0  1 0
10 10 1
```

Putting this all together we have:

```
   [xform =: (translate _10 _10) mp (rotate 90) mp translate 10 10
 0 1 0
_1 0 0
20 0 1
   (square ,. 1) mp 3 2 {. xform
20  0
20 10
10 10
10  0
20  0
```

which is shown in Figure 9.

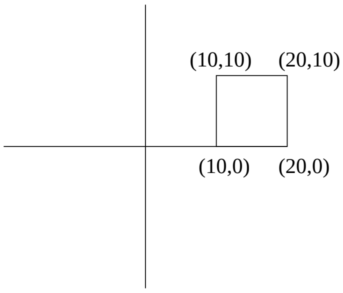

Figure 9: Rotating a Square 90 Degrees About (10,10)

9 Representing 3D Objects

Three dimensional objects may be modeled by a collection of points $[x_i, y_i, z_i]$, representing the vertices of the object, together with additional information which describes which vertices are used to form planes, surface properties such as color and texture, etc. Such an image model is often refered to as a polygonal model.

For example, the vertices of a cube of size 2, centered at the origin of three-dimensional space can be generated by:

```
   [ cube =: _1 ^ #: i. 8
 1  1  1
 1  1 _1
 1 _1  1
 1 _1 _1
_1  1  1
_1  1 _1
_1 _1  1
_1 _1 _1
```

The top plane of this cube are describe by vertices `0 1 5 4`.

```
   0 1 5 4 { cube
 1  1  1
 1  1 _1
_1  1 _1
_1  1  1
```

The five other planes in this cube are similarly described. For example, the left face is given by

```
   4 5 7 6 { cube
_1  1  1
_1  1 _1
_1 _1 _1
_1 _1  1
```

10 3D Transformations

The 2D transformations of Section 5.1 may be extended to the 3D case as follows.

10.1 Translation

Using homogeneous coordinates we extend the Equations 8 to three dimensional space:

$$\begin{aligned} x' &= x + T_x \\ y' &= y + T_y \\ z' &= z + T_z \\ w' &= w \end{aligned} \tag{14}$$

Consider the 4 by 4 matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

We see that the Equations 14 may be written as the matrix equation

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & y & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix} \tag{15}$$

We define the J monad `translate` which is applied to a list of three translate values $T_x \quad T_y \quad T_z$ to produce the translation matrix.

```
   translate =: monad def '((=/ ~ i. 3) , y. ) ,. 0 0 0 1'
   translate 1 1 1
1 0 0 0
0 1 0 0
0 0 1 0
1 1 1 1
```

Hence,

```
   (cube ,. 1) mp 4 3 {. translate 1 1 1
2 2 2
2 2 0
2 0 2
2 0 0
0 2 2
0 2 0
0 0 2
0 0 0
```

translates the cube to the positive sector of 3-space.

10.2 3D Scaling

Next we extend the Equations 3 to three dimensional space as the follows:

$$\begin{aligned} x' &= x \times S_x \\ y' &= y \times S_y \\ z' &= z \times S_z \\ w' &= w \end{aligned} \tag{16}$$

Consider the 4 by 4 matrix

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We write the Equations 16 as:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

We define the J monad `scale` which is applied to a list of three scale factors $S_x \quad S_y \quad S_z$ to produce the scaling matrix.

```
scale =: monad def '4 4 $ (0 { y.), 0 0 0 0 , (1 { y.), 0 0 0 0 , (2 { y.), 0 0 0 0 1'
```

We can scale `cube` to size 4 by:

```
   (cube ,. 1) mp 4 3 {. scale 2 2 2
 2  2  2
 2  2 _2
 2 _2  2
 2 _2 _2
_2  2  2
_2  2 _2
_2 _2  2
_2 _2 _2
```

10.3 3D Rotation

Extending the Equations 1 to three dimensional space is a bit more complex as we need to describe three rotation matrices which rotate points about the x, y, and z axes respectively.

The z axis rotation equations are:

$$\begin{aligned} x' &= x \times cos(\theta) - y \times sin(\theta) \\ y' &= x \times sin(\theta) + y \times cos(\theta) \\ z' &= z \\ w' &= w \end{aligned} \tag{18}$$

Consider the 4 by 4 matrix

$$\begin{bmatrix} cos(\theta) & sin(\theta) & 0 & 0 \\ -sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We write Equations 18 as:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} cos(\theta) & sin(\theta) & 0 & 0 \\ -sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{19}$$

We define the J monad `z_rotate` which is applied to an angle θ to produce the z axis rotation matrix.

```
  z_rotate =: monad def '(1 1 _1 1 * 2 1 1 2 o. (o. y.) % 180) (0 0;0 1;1 0;1 1) } =/ ~ i. 4'
  z_rotate 90
 0 1 0 0
_1 0 0 0
 0 0 1 0
 0 0 0 1
```

We can rotate `cube` 90 degrees about the z axis by

```
   (cube ,. 1) mp 4 3 {. z_rotate 90
_1  1  1
_1  1 _1
 1  1  1
 1  1 _1
_1 _1  1
_1 _1 _1
 1 _1  1
 1 _1 _1
```

We can also see that rotating the top face of `cube` produces the left face of `cube` described in Section 9

```
   (0 1 5 4 { cube ,. 1) mp 4 3 {. z_rotate 90
_1  1  1
_1  1 _1
_1 _1 _1
_1 _1  1
```

The y axis rotation equations are:

$$\begin{aligned} x' &= x \times cos(\theta) + z \times sin(\theta) \\ y' &= y \\ z' &= -x \times sin(\theta) + z \times cos(\theta) \\ w' &= w \end{aligned} \tag{20}$$

Consider the 4 by 4 matrix

$$\begin{bmatrix} cos(\theta) & 0 & -sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We write Equations 20 as:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{21}$$

We define the J monad `y_rotate` which is applied to an angle θ to produce the y axis rotation matrix.

```
   y_rotate =: monad def '(1 _1 1 1 * 2 1 1 2 o. (o. y.) % 180) (0 0;0 2;2 0;2 2) } =/ ~ i. 4'
   y_rotate 90
0 0 _1 0
0 1  0 0
1 0  0 0
0 0  0 1
```

We can rotate `cube` 90 degrees about the y axis by

```
   (cube ,. 1) mp 4 3 {. y_rotate 90
 1  1 _1
_1  1 _1
 1 _1 _1
_1 _1 _1
 1  1  1
_1  1  1
 1 _1  1
_1 _1  1
```

The x axis rotation equations are:

$$\begin{aligned} &x' = x \\ &y' = y \times cos(\theta) - z \times sin(\theta) \\ &z' = y \times sin(\theta) + z \times cos(\theta) \\ &w' = w \end{aligned} \tag{22}$$

Consider the 4 by 4 matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & sin(\theta) & 0 \\ 0 & -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We write Equations 22 as:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \texttt{mp} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & sin(\theta) & 0 \\ 0 & -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{23}$$

We define the J monad `x_rotate` which is applied to an angle θ to produce the x axis rotation matrix.

```
   x_rotate =: monad def '(1 1 _1 1 * 2 1 1 2 o. (o. y.) % 180) (1 1;1 2;2 1;2 2) } =/ ~ i. 4'
   x_rotate 90
1  0 0 0
0  0 1 0
0 _1 0 0
0  0 0 1
```

We can rotate `cube` 90 degrees about the x axis by

```
   (cube ,. 1) mp 4 3 {. x_rotate 90
 1 _1  1
 1  1  1
 1 _1 _1
 1  1 _1
_1 _1  1
_1  1  1
_1 _1 _1
_1  1 _1
```

11 3D Viewing

The initial viewing parameters are choosen so as to be able to give an unrestricted view of the scene. In practice, however, some simplifications are most often used as default viewing parameters.

The *projection plane*, shown in Figure 10, has the view plane defined by a point on the plane (VRP) and the view plane normal (VPN).

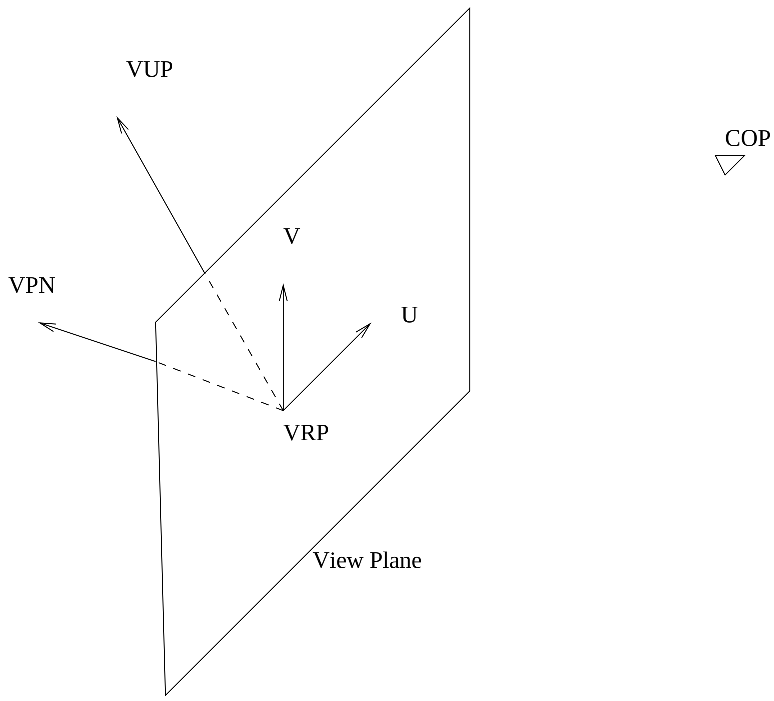

Figure 10: Initial Viewing Parameters

The VPN gives the orientation of the view plane and is often (but not required to be) parallel to the view direction. The VPN is used to define a left-handed coordinate system *screen coordinate system*. The VUP vector defines a direction which is not parallel to VPN and is taken to be the viewer's concept of *up*. VUP need not (but often is taken to) be perpendicular to VPN. The projection of VUP on the view plane defines the V axis of the screen coordinate system. The U axis of the screen coordinate system is choosen to be perpendicular to both (*orthogonal to*) V and VPN. These vectors are choosen so as to form a left-handed V, U, VPN 3D coordinate system. The VRP is the origin of the 2D screen coordinate system. However, VRP is not the origin of the left-handed 3D coordinate system we wish to define. Its origin is the location of the eye (COP). The coordinates of COP are defined relative to the VRP using world coordinates.

12 The Eye Coordinate System

We now have all of the parameters necessary to describe the 3D viewing transformation which maps the world coordinate system into the eye coordinate system. A rectangular region, Figure 11, (*viewport*) describes the clipping region of the screen coordinate system which is visible to the viewer.

This 2 dimensional viewport has sides which are parallel to the V U axis and the location and size of the viewport are given in units of the screen coordinate system using VRP as the origin. The viewport forms a *viewing pyramid* which gives the visible portion of world coordinate space from COP. All objects outside this pyramid are clipped from the scene. Actually, two additional clipping planes (Near and Far; see Figure 12) which are parallel to the view plane define portions of the seen which are either too close or too far from COP to be seen. The line from the COP through the center of the viewport defines the viewing direction and will be the positive Z axis of the eye coordinate system.

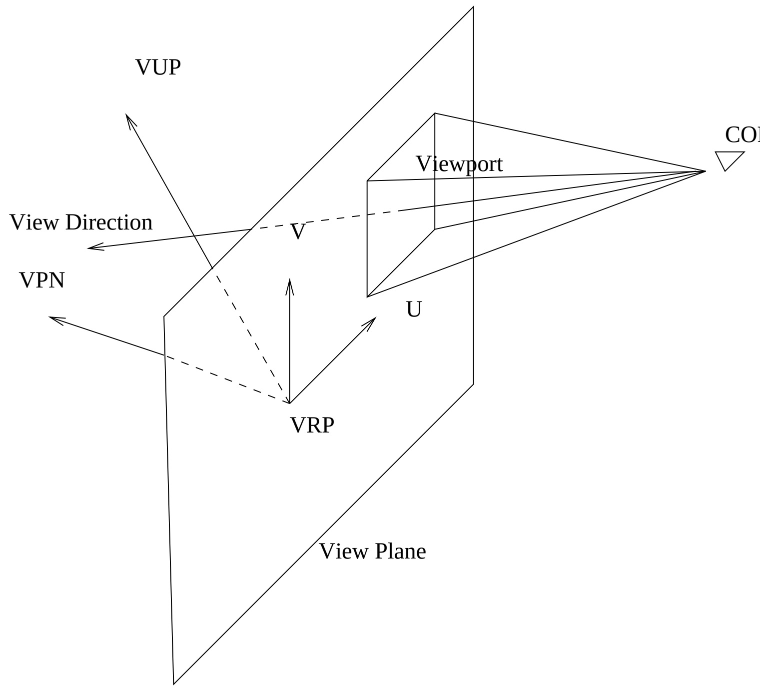

Figure 11: Viewport

13 Some Linear Algebra

A few concepts from elementary linear algebra are useful at this point. Let $\mathbf{v_1} = (x_1, y_1, z_1)$ and $\mathbf{v_2} = (x_2, y_2, z_2)$ be two 3D vectors. The *dot product* or *inner product* of $\mathbf{v_1}$ and $\mathbf{v_2}$ is defined as:

$$\mathbf{v_1} \cdot \mathbf{v_2} = x_1 \times x_2 + y_1 \times y_2 + z_1 \times z_2$$

Notice that the inner product of two vectors is a real number. The cosine of the acute angle θ between two vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ is defined as:

$$cos\theta = \frac{\mathbf{v_1} \cdot \mathbf{v_2}}{(\mid \mathbf{v_1} \mid \times \mid \mathbf{v_2} \mid)}$$

where $\mid \mathbf{v_1} \mid$ is the length of the vector $\mathbf{v_1}$ and $\mid \mathbf{v_2} \mid$ is the length of the vector $\mathbf{v_2}$. Hence we may write:

$$\mathbf{v_1} \cdot \mathbf{v_2} = \mid \mathbf{v_1} \mid \times \mid \mathbf{v_2} \mid \times cos\theta$$

Notice that if $\mathbf{v_1}$ and $\mathbf{v_2}$ are two perpendicular vectors, then the angle between each other is $\theta = 90$ and $cos\theta = 0$. Hence, the inner product of two perpendicular vectors is 0.

The *length* of a vector $\mathbf{v} = (x, y, z)$ is defined as $\sqrt{x^2 + y^2 + z^2}$. The inner product of $\mathbf{v}$ with itself yields:

$$\mathbf{v} \cdot \mathbf{v} = x^2 + y^2 + z^2$$

or

$$\mathbf{v} \cdot \mathbf{v} = \mid \mathbf{v} \mid^2$$

If $\mid \mathbf{v} \mid = 1$, then $\mathbf{v} \cdot \mathbf{v} = 1$.

Given two vectors $\mathbf{v_1} = (x_1, y_1, z_1)$ and $\mathbf{v_2} = (x_2, y_2, z_2)$, then the sum of $\mathbf{v_1}$ and $\mathbf{v_2}$ is the vector

$$\mathbf{v_1} + \mathbf{v_2} = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$$

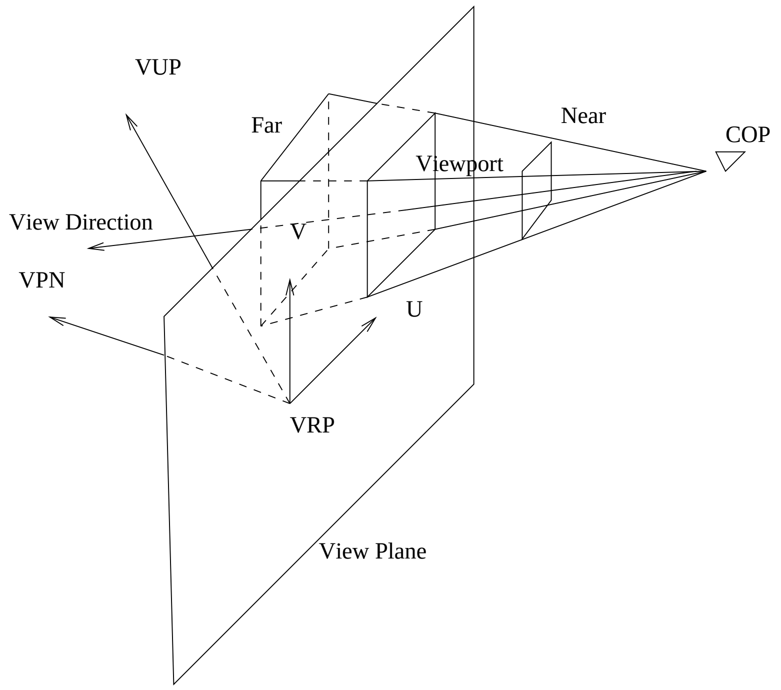

Figure 12: Viewing Pyramid

which is shown in Figure 13.

Inner product (dot product) distributes over vector addition. That is, given vectors **u**, **v**, **w**, and a real number r, then

$$\mathbf{u} \mathbin{.} (\mathbf{v} + r \times \mathbf{w}) = \mathbf{u} \mathbin{.} \mathbf{v} + r \times (\mathbf{u} \mathbin{.} \mathbf{w})$$

Suppose **n** is a vector whose length is 1 and **w** is a vector not parallel to **n**. We wish to project **w** to a vector **v** which lies on a plane perpendicular to **n** (see Figure 14).

To solve this problem define the vector **v** by the equation:

$$\mathbf{v} = \mathbf{w} - (\mathbf{n} \mathbin{.} \mathbf{w}) \times \mathbf{n}$$

Then,

$$\mathbf{n} \mathbin{.} \mathbf{v} = \mathbf{n} \mathbin{.} (\mathbf{w} - (\mathbf{n} \mathbin{.} \mathbf{w}) \times \mathbf{n}) = \mathbf{n} \mathbin{.} \mathbf{w} - (\mathbf{n} \mathbin{.} \mathbf{w}) \times (\mathbf{n} \mathbin{.} \mathbf{n})$$

But $\mathbf{n} \mathbin{.} \mathbf{n} = 1$ since **n** is of length one. Hence, $\mathbf{n} \mathbin{.} \mathbf{v} = 0$ and from this it follows that **n** and **v** are perpendicular. Therefore, **v** is a vector that lies on a plane perpendicular to **n**

The *cross product* of two vectors $\mathbf{v_1} = (x_1, y_1, z_1)$ and $\mathbf{v_2} = (x_2, y_2, z_2)$ is the vector:

$$\mathbf{v_1} \times \mathbf{v_2} = (y_1 \times z_2 - z_1 \times y_2, z_1 \times x_2 - x_1 \times z_2, x_1 \times y_2 - y_1 \times x_2)$$

The cross product of two non-parallel vectors is a vector which is perpendicular to both vectors. Hence, the inner product of either $\mathbf{v_1}$ or $\mathbf{v_2}$ with $\mathbf{v_1} \times \mathbf{v_2}$ is zero.

$$\mathbf{v_1} \mathbin{.} (\mathbf{v_1} \times \mathbf{v_2}) = \mathbf{v_2} \mathbin{.} (\mathbf{v_1} \times \mathbf{v_2}) = 0$$

The direction of the vector $\mathbf{v_1} \times \mathbf{v_2}$ is such that if the fingers of the right hand are curled around $\mathbf{v_1}$ in the direction of $\mathbf{v_2}$, then $\mathbf{v_1} \times \mathbf{v_2}$ is pointing in the direction of the thumb. Cross product is *not* commutative.

$$\mathbf{v_1} \times \mathbf{v_2} = -\mathbf{v_2} \times \mathbf{v_1}$$

which means that the direction of $\mathbf{v_2} \times \mathbf{v_1}$ is the opposite of the direction of $\mathbf{v_1} \times \mathbf{v_2}$.

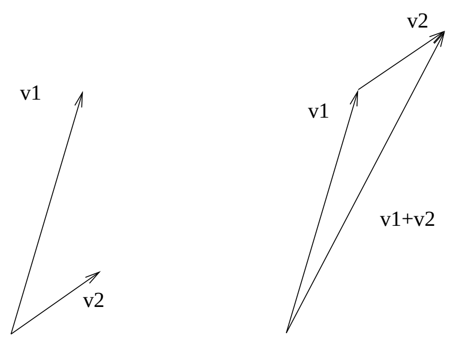

Figure 13: Sum of Vectors v1 and v2

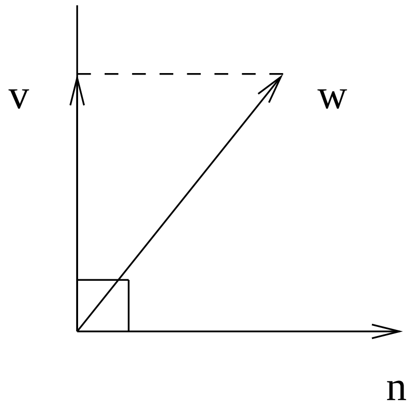

Figure 14: Projecting **w** on **v**

14 Constructing the Viewing Matrix

The 3-D viewing pipline is shown in Figure 15. The second step of the pipeline involves transforming the vertices of model objects which are given in world coordinates to the eye coordinate system. This process starts from the initial parameters of COP, VRP, VPN, and VUP. These vectors are first used to compute the **vu** coordinate system as:

$$\mathbf{v} = VUP - (VPN \cdot VUP) \times VPN$$

$$\mathbf{u} = VPN \times \mathbf{v}$$

The next step is to transform the left-handed eye coordinate system defined by **v**, **u** and VPN into the right-handed world coordinate system. This is accomplished by three steps:

1. The origin of the eye coordinate system, COP is translated to the origin $(0, 0, 0,)$ of the world coordinate system.
2. Rotate so that
   (a) The axis in the **u** direction is parallel to the world coordinate x-axis.
   (b) The axis in the **v** direction is parallel to the world coordinate y-axis.
   (c) The VPN is parallel to the negative z-axis, that is, going into the display screen.

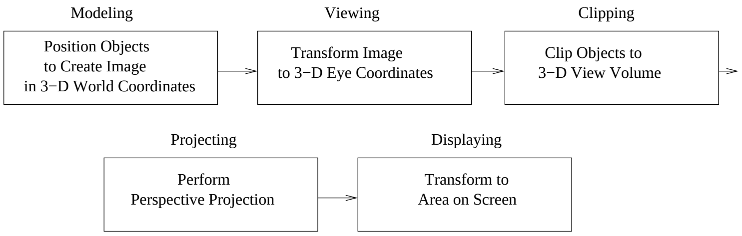

Figure 15: The Graphics Pipeline

3. Make the negative z-axis the positive direction so that the positive z direction goes into the screen, i.e., make the eye coordinate system left-handed.

The matrices required to accomplish this transformation are given next.
Since the center of projection, COP is defined relative to the VRP, the translation matrix is:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -(VRP[x]+COP[x]) & -(VRP[y]+COP[y]) & -(VRP[z]+COP[z]) & 1 \end{bmatrix}$$

The rotation matrix is

$$R = \begin{bmatrix} \mathbf{u}[x] & \mathbf{v}[x] & -VPN[x] & 0 \\ \mathbf{u}[y] & \mathbf{v}[y] & -VPN[y] & 0 \\ \mathbf{u}[z] & \mathbf{v}[z] & -VPN[z] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix to change the direction of the z-axis is

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can combine the matrices R and C by computing the matrix product $R \times C$ and rename it R producing the matrix

$$R = \begin{bmatrix} \mathbf{u}[x] & \mathbf{v}[x] & VPN[x] & 0 \\ \mathbf{u}[y] & \mathbf{v}[y] & VPN[y] & 0 \\ \mathbf{u}[z] & \mathbf{v}[z] & VPN[z] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The final matrix to produce the transformation from world coordinates to eye coordinates is the product of the two matrices $V = T \times R$.

15 Prospective Projection

After multiplying world coordinate vertices by the viewing transformation, V, and clipping to the truncated viewing pyramid, it is necessary to perform the perspective projection onto the view plane. Given a vertex in the eye coordinate system, $\mathbf{v} = (x_e, y_e, z_e)$, the projected screen coordinates, (x_s, y_s) are computed as:

$$x_s = \frac{d \times x_e}{z_e}$$
$$y_s = \frac{d \times y_e}{z_e}$$

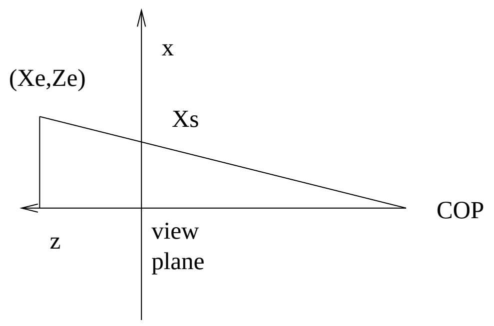

Figure 16: Perspective Projection

where d is the distance from COP to the view plane. These formulas are easily derived by considering the projection onto the $x - z$ (Figure 16) and $y - z$ planes and noting that from similar triangles

$$\frac{x_e}{z_e} = \frac{x_s}{d}$$

The equation for y_s is derived in a similar fashion.

16 C Programs to Compute the Viewing Transformation

In this section we give some C program fragments to illustrate algorithms for computation of the 3D viewing transformation which transforms world coordinates to eye coordinates.

```
typedef double Xform3d[4][4];

typedef struct Point3d                  /* the 3D homogeneous point */
{
        double x, y, z, w;

} Point3d, *Point3dPtr, **Point3dHdl;

typedef struct Graph3dView              /* the 3D graphics viewing parameters */
{
        CWindowPtr wPtr;                        /* the color graph port */
        GrafPtr oldPort;                        /* the previous graph pointer */
        Point3d vrp;                            /* the view reference point */
        Point3d vpn;                            /* the view plane normal */
        Point3d vup;                            /* the view up direction */
        Point3d cop;                            /* the center of projection (viewpoint) */
        Rect viewport;                          /* the intersection of the viewing pyramid */
        double back;                            /* the z coordinate of the back clipping plane */
        double front;                           /* the z coordinate of the front clipping plane */
        double distance;                        /* the distance of the cop from the view plane */
        Xform3d xform;                          /* the current transformation */

} Graph3dView, *Graph3dViewPtr, **Graph3dViewHdl;

/*      --------------------------------------------------------------

        scale3d

        This function returns the 3D scaling matrix given
        x, y and z scaling factors.

*/

void scale3d(double sx, double sy, double sz, Xform3d scaleMatrix)

{
        scaleMatrix[0][0] = sx;
        scaleMatrix[0][1] = 0.0;
        scaleMatrix[0][2] = 0.0;
        scaleMatrix[0][3] = 0.0;

        scaleMatrix[1][0] = 0.0;
```

```
        scaleMatrix[1][1] = sy;
        scaleMatrix[1][2] = 0.0;
        scaleMatrix[1][3] = 0.0;

        scaleMatrix[2][0] = 0.0;
        scaleMatrix[2][1] = 0.0;
        scaleMatrix[2][2] = sz;
        scaleMatrix[2][3] = 0.0;

        scaleMatrix[3][0] = 0.0;
        scaleMatrix[3][1] = 0.0;
        scaleMatrix[3][2] = 0.0;
        scaleMatrix[3][3] = 1.0;

}       /* End of scale3d */

/*      --------------------------------------------------------------

        translate3d

        This function returns the 3d translation matrix given
        x, y and z translation factors.

*/

void translate3d(double tx, double ty, double tz, Xform3d transMatrix)

{
        transMatrix[0][0] = 1.0;
        transMatrix[0][1] = 0.0;
        transMatrix[0][2] = 0.0;
        transMatrix[0][3] = 0.0;

        transMatrix[1][0] = 0.0;
        transMatrix[1][1] = 1.0;
        transMatrix[1][2] = 0.0;
        transMatrix[1][3] = 0.0;

        transMatrix[2][0] = 0.0;
        transMatrix[2][1] = 0.0;
        transMatrix[2][2] = 1.0;
        transMatrix[2][3] = 0.0;

        transMatrix[3][0] = tx;
        transMatrix[3][1] = ty;
        transMatrix[3][2] = tz;
        transMatrix[3][3] = 1.0;

}       /* End of translate3d */

/*      --------------------------------------------------------------

        identity3d

        This function returns the 4 by 4
        identity matrix.

*/

void identity3d(Xform3d identity)

{
        identity[0][0] = 1.0;
        identity[0][1] = 0.0;
        identity[0][2] = 0.0;
        identity[0][3] = 0.0;

        identity[1][0] = 0.0;
        identity[1][1] = 1.0;
        identity[1][2] = 0.0;
        identity[1][3] = 0.0;

        identity[2][0] = 0.0;
        identity[2][1] = 0.0;
        identity[2][2] = 1.0;
        identity[2][3] = 0.0;

        identity[3][0] = 0.0;
        identity[3][1] = 0.0;
        identity[3][2] = 0.0;
        identity[3][3] = 1.0;
```

```
}       /* End of identity3d */

/*      --------------------------------------------------------------

        rotateX3d

        This function returns the x axis rotation matrix given
        an angle in radians.

*/

void rotateX3d(double theta, Xform3d rotateMatrix)

{
        double  sine = sin(theta),
                cosine = cos(theta);

        rotateMatrix[0][0] = 1.0;
        rotateMatrix[0][1] = 0.0;
        rotateMatrix[0][2] = 0.0;
        rotateMatrix[0][3] = 0.0;

        rotateMatrix[1][0] = 0.0;
        rotateMatrix[1][1] = cosine;
        rotateMatrix[1][2] = sine;
        rotateMatrix[1][3] = 0.0;

        rotateMatrix[2][0] = 0.0;
        rotateMatrix[2][1] = -sine;
        rotateMatrix[2][2] = cosine;
        rotateMatrix[2][3] = 0.0;

        rotateMatrix[3][0] = 0.0;
        rotateMatrix[3][1] = 0.0;
        rotateMatrix[3][2] = 0.0;
        rotateMatrix[3][3] = 1.0;

}       /* End of rotateX3d */

/*      --------------------------------------------------------------

        rotateY3d

        This function returns the y axis rotation matrix given
        an angle in radians.

*/

void rotateY3d(double theta, Xform3d rotateMatrix)

{
        double  sine = sin(theta),
                cosine = cos(theta);

        rotateMatrix[0][0] = cosine;
        rotateMatrix[0][1] = 0.0;
        rotateMatrix[0][2] = sine;
        rotateMatrix[0][3] = 0.0;

        rotateMatrix[1][0] = 0.0;
        rotateMatrix[1][1] = 1.0;
        rotateMatrix[1][2] = 0.0;
        rotateMatrix[1][3] = 0.0;

        rotateMatrix[2][0] = -sine;
        rotateMatrix[2][1] = 0.0;
        rotateMatrix[2][2] = cosine;
        rotateMatrix[2][3] = 0.0;

        rotateMatrix[3][0] = 0.0;
        rotateMatrix[3][1] = 0.0;
        rotateMatrix[3][2] = 0.0;
        rotateMatrix[3][3] = 1.0;

}       /* End of rotateY3d */

/*      --------------------------------------------------------------

        rotateZ3d
```

```
        This function returns the z axis rotation matrix given
        an angle in radians.

*/

void rotateZ3d(double theta, Xform3d rotateMatrix)

{
        double  sine = sin(theta),
                        cosine = cos(theta);

        rotateMatrix[0][0] = cosine;
        rotateMatrix[0][1] = sine;
        rotateMatrix[0][2] = 0.0;
        rotateMatrix[0][3] = 0.0;

        rotateMatrix[1][0] = -sine;
        rotateMatrix[1][1] = cosine;
        rotateMatrix[1][2] = 0.0;
        rotateMatrix[1][3] = 0.0;

        rotateMatrix[2][0] = 0.0;
        rotateMatrix[2][1] = 0.0;
        rotateMatrix[2][2] = 1.0;
        rotateMatrix[2][3] = 0.0;

        rotateMatrix[3][0] = 0.0;
        rotateMatrix[3][1] = 0.0;
        rotateMatrix[3][2] = 0.0;
        rotateMatrix[3][3] = 1.0;

}       /* End of rotateZ3d */

/*      --------------------------------------------------------------

        shearZ3d

        This function produces the Z shearing transformation
        which maps an arbitrary line through the origin
        and passing through the non-zero point (x, y, z)
        into the Z axis without changing the z values of
        points on the line.

*/

void shearZ3d(double x, double y, double z, Xform3d zshear)

{
        zshear[0][0] = 1.0;
        zshear[0][1] = 0.0;
        zshear[0][2] = 0.0;
        zshear[0][3] = 0.0;

        zshear[1][0] = 0.0;
        zshear[1][1] = 1.0;
        zshear[1][2] = 0.0;
        zshear[1][3] = 0.0;

        zshear[2][0] = -x / z;
        zshear[2][1] = -y / z;
        zshear[2][2] = 1.0;
        zshear[2][3] = 0.0;

        zshear[3][0] = 0.0;
        zshear[3][1] = 0.0;
        zshear[3][2] = 0.0;
        zshear[3][3] = 1.0;

}       /* End of shearZ3d */

/*      --------------------------------------------------------------

        copy3dXform

        This function copies the src 4 by 4 transformation
        matrix to the dst 4 by 4 matrix.  It is assumed that
        the storage for the matrices is allocated in the
        calling routine.
*/

void copy3dXform(Xform3d dst, Xform3d src)
```

```
{
        register int i, j;
        for(i = 0; i < 4; i++)
                for(j = 0; j < 4; j++)
                        dst[i][j] = src[i][j];

}       /* End of copy3dXform */

/*      --------------------------------------------------------------

        mult3dXform

        This function multiplies two 4 by 4 transformation
        matricies producing a resulting 4 by 4
        transformation.  For efficiency, we assume that
        the last column of the Xform3d is 0 0 0 1
        (36 multiplications and 27 additions)

*/

void mult3dXform(Xform3d xform1, Xform3d xform2, Xform3d resultxform)

{
        Xform3d result;

        /* row 0 (9 * and 6 +) */
        result[0][0] = xform1[0][0] * xform2[0][0] +
                                                xform1[0][1] * xform2[1][0] +
                                                xform1[0][2] * xform2[2][0];

        result[0][1] = xform1[0][0] * xform2[0][1] +
                                                xform1[0][1] * xform2[1][1] +
                                                xform1[0][2] * xform2[2][1];

        result[0][2] = xform1[0][0] * xform2[0][2] +
                                                xform1[0][1] * xform2[1][2] +
                                                xform1[0][2] * xform2[2][2];
        result[0][3] = 0.0;

        /* row 1 (9 * and 6 +) */
        result[1][0] = xform1[1][0] * xform2[0][0] +
                                                xform1[1][1] * xform2[1][0] +
                                                xform1[1][2] * xform2[2][0];

        result[1][1] = xform1[1][0] * xform2[0][1] +
                                                xform1[1][1] * xform2[1][1] +
                                                xform1[1][2] * xform2[2][1];

        result[1][2] = xform1[1][0] * xform2[0][2] +
                                                xform1[1][1] * xform2[1][2] +
                                                xform1[1][2] * xform2[2][2];
        result[1][3] = 0.0;

        /* row 2 (9 * and 6 +) */
        result[2][0] = xform1[2][0] * xform2[0][0] +
                                                xform1[2][1] * xform2[1][0] +
                                                xform1[2][2] * xform2[2][0];

        result[2][1] = xform1[2][0] * xform2[0][1] +
                                                xform1[2][1] * xform2[1][1] +
                                                xform1[2][2] * xform2[2][1];

        result[2][2] = xform1[2][0] * xform2[0][2] +
                                                xform1[2][1] * xform2[1][2] +
                                                xform1[2][2] * xform2[2][2];
        result[2][3] = 0.0;

        /* row 3 ( 9 * and 9 +) */
        result[3][0] = xform1[3][0] * xform2[0][0] +
                                                xform1[3][1] * xform2[1][0] +
                                                xform1[3][2] * xform2[2][0] +
                                                xform2[3][0];

        result[3][1] = xform1[3][0] * xform2[0][1] +
                                                xform1[3][1] * xform2[1][1] +
                                                xform1[3][2] * xform2[2][1] +
                                                xform2[3][1];

        result[3][2] = xform1[3][0] * xform2[0][2] +
                                                xform1[3][1] * xform2[1][2] +
```

```
                                                xform1[3][2] * xform2[2][2] +
                                                xform2[3][2];
        result[3][3] = 1.0;

        /* copy the result */
        copy3dXform(resultxform, result);

}       /* End of mult3dXform */

/*      --------------------------------------------------------------

        transform3dObject

        This function multiplies an n array
        of Point3d  by a 4 by 4
        transformation matrix producing
        a resulting n array of Point3d.
        We assume that the last column of
        xform is 0 0 0 1.
        ( 9n multiplications and 9n additions)

*/

void transform3dObject(int n, Point3d object[], Xform3d xform, Point3d result[])

{
        register int i;

        for(i = 0; i < n; i++)  /* each row */
        {
                /* column 0 (3 * and 3 +) */
                result[i].x =   object[i].x * xform[0][0] +
                                                object[i].y * xform[1][0] +
                                                object[i].z * xform[2][0] + xform[3][0];

                /* column 1 (3 * and 3 +) */
                result[i].y =   object[i].x * xform[0][1] +
                                                object[i].y * xform[1][1] +
                                                object[i].z * xform[2][1] + xform[3][1];

                /* column 2 (3 * and 3 +) */
                result[i].z =   object[i].x * xform[0][2] +
                                                object[i].y * xform[1][2] +
                                                object[i].z * xform[2][2] + xform[3][2];

                /* column 3 */
                result[i].w =   object[i].w;
        }

}       /* End of transform3dObject */

/*      --------------------------------------------------------------

        copy3dObject

        This function copies a src n array
        of Point3d  to a dst n array of Point3d.
        It is assumed that storage for the src
        and dst arrays is allocated in the calling
        function.
*/

void copy3dObject(int n, Point3d dst[], Point3d src[])

{
        register int i;

        for(i = 0; i < n; i++)  /* each row */
                dst[i] = src[i];

}       /* End of copy3dObject */

/*      --------------------------------------------------------------

        pitch

        This function multiplies the transform associated with
        the given graph3dView by an X axis rotation and stores
        the resulting transformation as the new graph3dView xform.
*/
```

```
void pitch(double theta, Graph3dViewPtr viewPtr)

{
        Xform3d rotX;

        rotateX3d(theta, rotX);
        mult3dXform((*viewPtr).xform, rotX, (*viewPtr).xform);

}       /* End of pitch */

/*      --------------------------------------------------------------

        roll

        This function multiplies the transform associated with
        the given graph3dView by an Z axis rotation and stores
        the resulting transformation as the new graph3dView xform.
*/

void roll(double theta, Graph3dViewPtr viewPtr)

{
        Xform3d rotZ;

        rotateZ3d(theta, rotZ);
        mult3dXform((*viewPtr).xform, rotZ, (*viewPtr).xform);

}       /* End of roll */

/*      --------------------------------------------------------------

        yaw

        This function multiplies the transform associated with
        the given graph3dView by an Y axis rotation and stores
        the resulting transformation as the new graph3dView xform.
*/

void yaw(double theta, Graph3dViewPtr viewPtr)

{
        Xform3d rotY;

        rotateY3d(theta, rotY);
        mult3dXform((*viewPtr).xform, rotY, (*viewPtr).xform);

}       /* End of yaw */

a/*      --------------------------------------------------------------

        dotProduct

        This function computes the dot product
        (inner product) of two Point3d's.  The w
        component of the homogeneous representation
        of the 3d points is ignored in this
        calculation

*/

double dotProduct(Point3dPtr p1, Point3dPtr p2)

{
        return (*p1).x * (*p2).x + (*p1).y * (*p2).y + (*p1).z * (*p2).z;

}       /* End of dotProduct */

/*      --------------------------------------------------------------

        scalarProduct

        This function computes the scalar product
        of a double and a Point3d.  The w component of the
        homogeneous representation of the 3d points
        is ignored in this calculation.

*/

void scalarProduct(double a, Point3dPtr p, Point3dPtr result)

{
```

```
        (*result).x = a * (*p).x;
        (*result).y = a * (*p).y;
        (*result).z = a * (*p).z;

}       /* End of scalarProduct */

/*      --------------------------------------------------------------

        crossProduct

        This function computes the cross product
        of two Point3d's.  The w component of the
        homogeneous representation of the 3d points
        is ignored in this calculation.

*/

void crossProduct(Point3dPtr p1, Point3dPtr p2, Point3dPtr result)

{
        (*result).x = (*p1).y * (*p2).z - (*p1).z * (*p2).y;
        (*result).y = (*p1).z * (*p2).x - (*p1).x * (*p2).z;
        (*result).z = (*p1).x * (*p2).y - (*p1).y * (*p2).x;

}       /* End of crossProduct */

/*      --------------------------------------------------------------

        normalize

        This function normalizes the Point3d, a pointer
        to which is passed as an argument.  The w component of the
        homogeneous representation of the 3d point
        is ignored in this calculation.

*/

void normalize(Point3dPtr p)

{
        double length = sqrt(dotProduct(p, p));
        if(length != 0.0)
        {
                (*p).x /= length;
                (*p).y /= length;
                (*p).z /= length;
        }

}       /* End of normalize */

/*      --------------------------------------------------------------

        transform3dPoint

        This function applies an Xform3d to a
        Point3d, transforming that Point3d.
        A pointer to a Point3d is passed.  The w
        component of the homogeneous representation
        of the 3d point is ignored in this calculation.

*/

void transform3dPoint(Point3dPtr p, Xform3d xform)

{
        Point3d t = *p;
        (*p).x =        t.x * xform[0][0] +
                                t.y * xform[1][0] +
                                t.z * xform[2][0] + xform[3][0];

        (*p).y =        t.x * xform[0][1] +
                                t.y * xform[1][1] +
                                t.z * xform[2][1] + xform[3][1];

        (*p).z =        t.x * xform[0][2] +
                                t.y * xform[1][2] +
                                t.z * xform[2][2] + xform[3][2];


}       /* End of transform3dPoint */
```

```
/*      --------------------------------------------------------------

        subtract3dPoint

        This function subtracts Point3dPtr p2 from
        Point3dPtr p1 producing Point3dPtr result.
        The w component of the homogeneous
        representation of the 3d point
        is ignored in this calculation.

*/

void subtract3dPoint(Point3dPtr p1, Point3dPtr p2, Point3dPtr result)

{
        (*result).x = (*p1).x - (*p2).x;

        (*result).y = (*p1).y - (*p2).y;

        (*result).z = (*p1).z - (*p2).z;


}       /* End of subtract3dPoint */

/*      --------------------------------------------------------------

        add3dPoint

        This function adds Point3dPtr p2 to
        Point3dPtr p1 producing Point3dPtr result.
        The w component of the homogeneous
        representation of the 3d point
        is ignored in this calculation.

*/

void add3dPoint(Point3dPtr p1, Point3dPtr p2, Point3dPtr result)

{
        (*result).x = (*p1).x + (*p2).x;

        (*result).y = (*p1).y + (*p2).y;

        (*result).z = (*p1).z + (*p2).z;


}       /* End of add3dPoint */

/*      --------------------------------------------------------------

        initGraph3dView

        This function initializes the 3D graphics viewing
        structure and makes a full screen drawing window.
        The Graph3dView is initialized with some default
        viewing parameters.  These parameters (except for
        the wPtr) may be changed before 3D drawing occurs.

*/

void initGraph3dView(Graph3dViewPtr viewPtr)

{
        GDHandle mainDevice = GetMainDevice();
        Rect mainRect = (**mainDevice).gdRect;
        short width, height;

        /* set view reference point to origin */
        (*viewPtr).vrp.x = 0.0;
        (*viewPtr).vrp.y = 0.0;
        (*viewPtr).vrp.z = 0.0;
        (*viewPtr).vrp.w = 1.0;

        /* set view plane normal to z axis */
        (*viewPtr).vpn.x = 0.0;
        (*viewPtr).vpn.y = 0.0;
        (*viewPtr).vpn.z = 1.0;
        (*viewPtr).vpn.w = 1.0;

        /* set view up direction to y axis */
        (*viewPtr).vup.x = 0.0;
```

```
        (*viewPtr).vup.y = 1.0;
        (*viewPtr).vup.z = 0.0;
        (*viewPtr).vup.w = 1.0;

        /* set view center of projection (viewpoint) */
        (*viewPtr).cop.x = 0.0;
        (*viewPtr).cop.y = 0.0;
        (*viewPtr).cop.z = 720.0;
        (*viewPtr).cop.w = 1.0;

        /* set view Rect */
        width = (mainRect.right - mainRect.left - 5) / 2;
        height = (mainRect.bottom - mainRect.top - 43) / 2;
        /* order lower left to upper right */
        SetRect(&(*viewPtr).viewport, - width, - height, width, height);

        /* set back and front Z clipping plane values */
        (*viewPtr).back = 1000.0;
        (*viewPtr).front = 0.0;
        (*viewPtr).distance = (*viewPtr).cop.z - (*viewPtr).vrp.z;

        GetPort(&(*viewPtr).oldPort);
        (*viewPtr).wPtr = makeDrawingWindow((char *)"\p3D Graphics View",
                                                        mainRect.left + 2,
                                                        mainRect.top + 40,
                                                        mainRect.right - 3,
                                                        mainRect.bottom -3);

        /* set the transformation to be identity */
        identity3d((*viewPtr).xform);

}       /* End of initGraph3dView */

/*      --------------------------------------------------------------

        viewing

        This function produces the viewing transformation matrix
        which transforms an abject from world coordinates to
        eye coordinates..
*/

void viewing(Graph3dViewPtr viewPtr, Xform3d *view)

{
        Point3d t, v, u;
        Xform3d tr, rot;

        /* project vup on the viewing plane getting the v axis for viewing plane*/
        scalarProduct(dotProduct(&((*viewPtr).vpn), &((*viewPtr).vup)), &((*viewPtr).vpn), &t);
        subtract3dPoint(&((*viewPtr).vup), &t, &v);

        /* compute the u axis of the viewing plane */
        crossProduct(&((*viewPtr).vpn), &v, &u);

        /* translate the cop + vrp to the origin */
        add3dPoint(&((*viewPtr).cop), &((*viewPtr).vrp), &t);
        translate3d(-t.x, -t.y, -t.z, tr);

        /* compute the rotation matrix */
        identity3d(rot);
        rot[0][0] = u.x; rot[1][0] = u.y; rot[2][0] = u.z;
        rot[0][1] = v.x; rot[1][1] = v.y; rot[2][1] = v.z;
        rot[0][2] = (*viewPtr).vpn.x; rot[1][2] = (*viewPtr).vpn.y; rot[2][2] = (*viewPtr).vpn.z;

        /* multiply the translation and rotation producing the view transformation */
        mult3dXform(tr, rot, *view);

}       /* End of viewing */
```

References

[Ber 1986] Berger, Marc, *Computer Graphics with PASCAL*, The Benjamin/Cummings Publishing Company, Inc., 1986.

[Hui 2001] Hui, Roger K. W., Iverson, Kenneth E., *J Dictionary*, J Software, Toronto, Canada, May 2001.