Computing the 3D Viewing Transformation

John E. Howland
Department of Computer Science
Trinity University
715 Stadium Drive
San Antonio, Texas 78212-7200
Voice: (210) 999-7380
Fax: (210) 999-7477
E-mail: jhowland@Ariel.CS.Trinity.Edu
Web: http://WWW.CS.Trinity.Edu/~jhowland/

October 24, 2004

Abstract

A method of computing the 3D viewing transformation which transforms the right-hand world coordinate system to left-hand eye coordinate system is presented.

Subject Areas: Computer Graphics.
Keywords: 3D Viewing Transformation.

1 Introduction

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 1, has the view plane defined by a point on the plane (VRP) and the view plane normal (VPN).

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.

2 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 2, (*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 3) which are parallel to the view plane define portions of the seen which are either too close or too far from

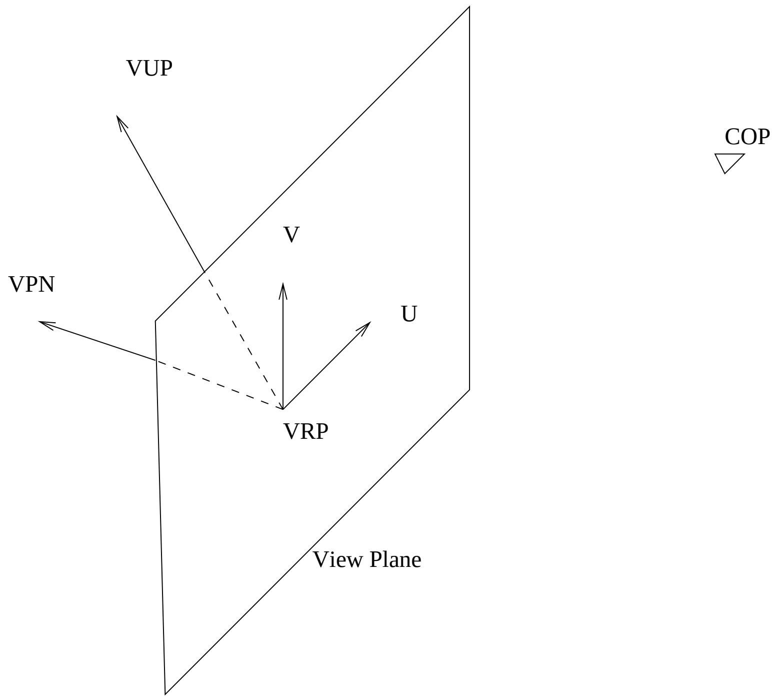

Figure 1: Initial Viewing Parameters

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.

3 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 $\mathbf{v} \cdot \mathbf{v} = 1$.

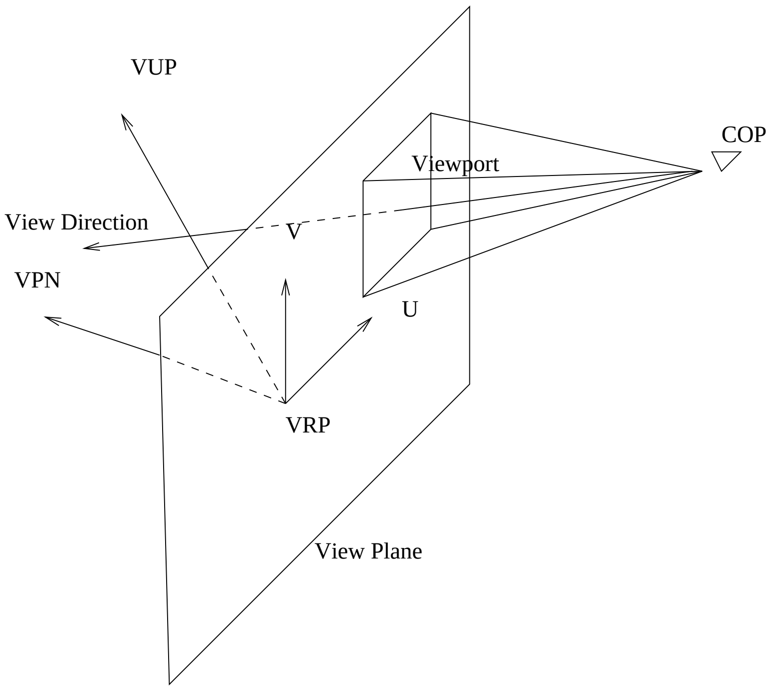

Figure 2: Viewport

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

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

which is shown in Figure 4.

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

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

Suppose $\mathbf{n}$ is a vector whose length is 1 and $\mathbf{w}$ is a vector not parallel to $\mathbf{n}$. We wish to project $\mathbf{w}$ to a vector $\mathbf{v}$ which lies on a plane perpendicular to $\mathbf{n}$ (see Figure 5).

To solve this problem define the vector $\mathbf{v}$ by the equation:

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

Then,

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

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

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.

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

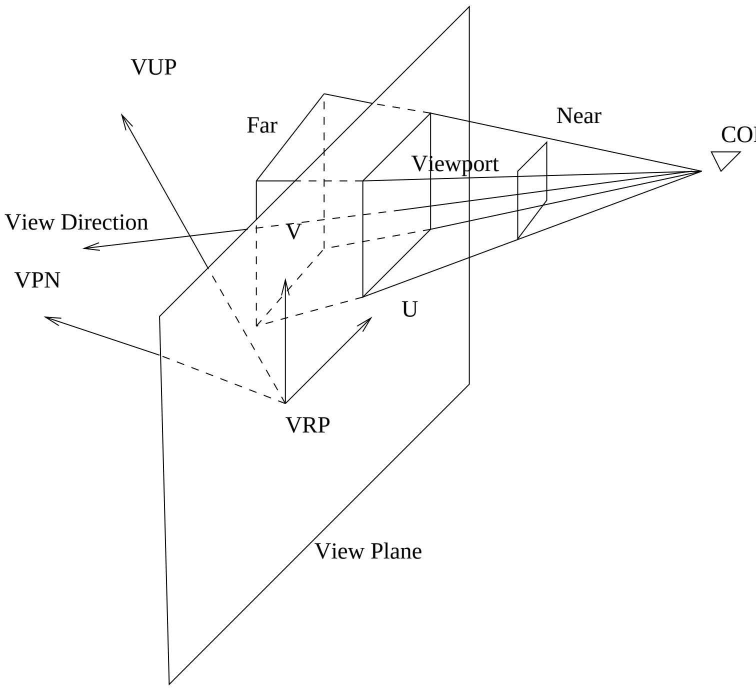

Figure 3: Viewing Pyramid

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 $\mathbf{v_1} \times \mathbf{v_2}$.

4 Constructing the Viewing Matrix

The 3-D viewing pipline is shown in Figure 6. 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.

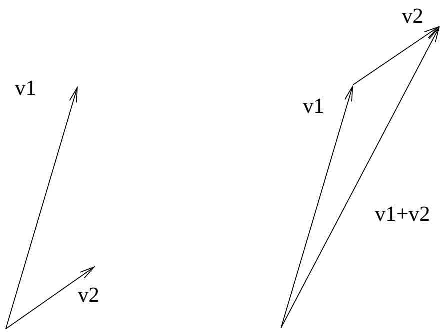

Figure 4: Sum of Vectors v1 and v2

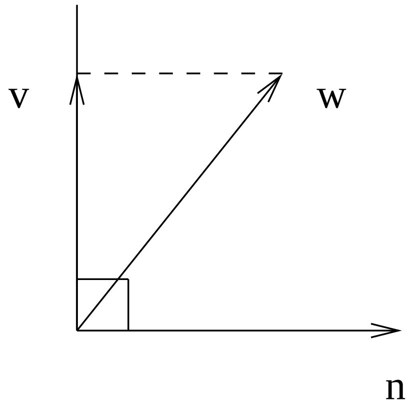

Figure 5: Projecting **w** on **v**

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

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

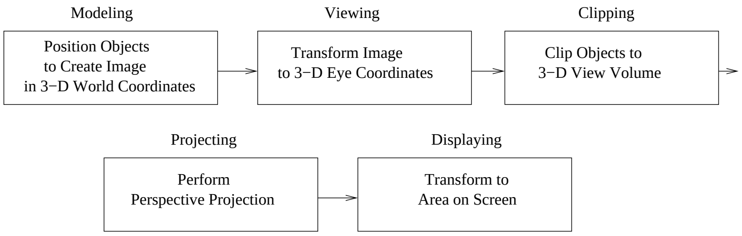

Figure 6: The Graphics Pipeline

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 $V = T \times R$.

5 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:

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

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 7) and $y - z$ planes and noting that from similar triangles

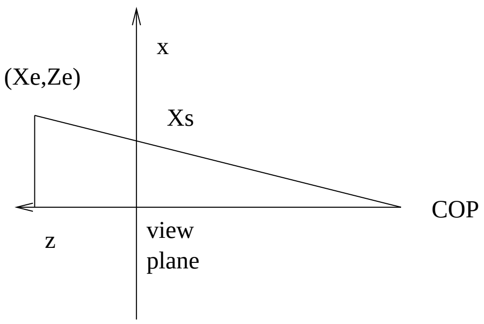

Figure 7: Perspective Projection

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

The equation for y_s is derived in a similar fashion.

6 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.