# 3D Viewing and Rotation Using Orthonormal Bases

Steve Cunningham
Computer Science Department
California State University, Stanislaus
Turlock, Ca 95380
rsc@altair.csustan.edu

This note discusses the general viewing transformation and rotations about a general line. It contrasts the common textbook approach to this problem with an approach based on orthonormal bases from linear algebra and shows that a straightforward piece of mathematics can improve the implementations of viewing and rotation transformations in actual work.

The common approach to viewing and general rotation operations treats these as a translation followed by a sequence of rotations about the coordinate axes in model space, and seems to go back to Newman and Sproull [4]. The viewing transformation requires three of these rotations, while the general rotation requires four. The entries in these rotation matrices are trigonometric functions of angles which are not readily seen, though the actual computations rarely use the trig functions; the entries are computed from the components of vectors derived from the translated eye vector as the rotations proceed. See Newman and Sproull (pps. 348 - 351) for more details; this same approach has been used in Hearn and Baker [3] and other books. Some books such as Foley and van Dam 2] discuss a (U,V,N) viewplane coordinate system such as we build below, but still use rotations to build the actual viewing transformation.

The approach we suggest was developed independently but is not original to this note. It appears in Berger [1] (pp. 274 - 282, where it seems a bit obscure), as well as in Salmon and Slater [5] (pp 406-408, where it is developed more formally), but is not widely known. No graphics texts seem to use this approach to general rotations in 3-space. The approach is straightforward: after the usual translation of the view reference point to the origin, use the eye point and up point in the viewing information to compute an orthonormal basis (U,V,N) for 3-space for which the eye vector is one component (the Z-axis analogue) and the up vector is projected onto another (the Y-axis analogue). Then the viewing transformation is simply a change

of basis and its matrix is directly written from the orthonormal (U,V,N) basis. The general rotation is much the same, with the up vector taken randomly, the desired rotation applied after the initial viewing transformation, and then the inverse of the viewing transformation is applied.

The advantages of the orthonormal basis approach are twofold: there is a logical basis for the approach which comes naturally from linear algebra, and the computation involves fewer steps. I have found that students can apply these ideas in their own code more quickly and accurately than they can the traditional approach.

## The New Approach

Consider a standard setup for 3D viewing (there are variations, but this uses standard information common to them all):

- A view reference point VRP
- An eye point EP
- An "up point" UP

From these we compute two vectors:

- An eye vector EV as the vector from VRP to EP
- An up vector UV as the vector from VRP to UP

and we have the situation shown in Figure 1:

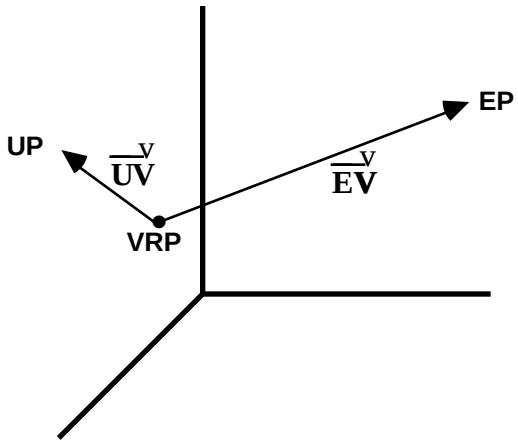


Figure 1: A standard viewing environment

The new process starts in the same way as the standard process: by defining a translation to move the view reference point to the origin. This has the standard matrix form

$$T_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -xv & -yv & -zv & 0 \end{bmatrix}$$

Next compute the orthonormal basis for the space as seen by the viewer, as follows:

1. normalize EV and call the result N,
2. normalize UV and continue to call the result UV,
3. Compute $V_1$ orthogonal to N by setting $V_1 = UV - (N \cdot UV)N$, as shown in Fig. 2,

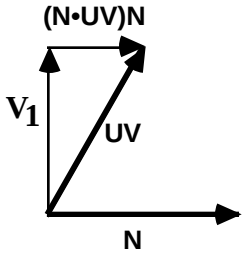


Figure 2: Orthogonalizing UV and N

4. normalize $V_1$ and call the result V,
5. compute the cross product $U = V \times N$ .

This creates a new coordinate system within the original model space which represents the coordinated of the desired viewing space. This coordinate system is shown in Figure 3.

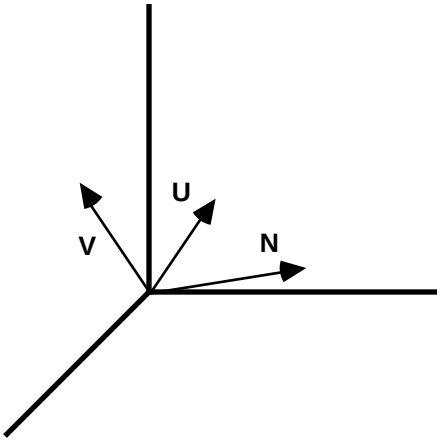


Figure 3: The UVN Coordinate System

Then the change of basis from (X,Y,Z)-space to (U,V,N)-space (assuming that matrices multiply on the right of their operands) has U as its first row, V as its second row, and N as its third row. The inverse of this matrix is the transformation from (U,V,N)-space to (X,Y,Z)-space and provides the construction of the rotation part of the matrix M of the viewing transformation. Since U, V, and N are orthonormal, this inverse is the transpose of the matrix above; the first three rows of M have U as the first column, V as the second column, and N as the third column, and the rest are zero. Finally, the viewing transformation is $T_0M$. It is easier to compute than the standard transformation and is as efficient to apply (if all three rotations are computed and multiplied together) or more efficient (if the rotations are applied separately). The actual matrix of this transformation is the output of the `BuildViewTransform` function below.

**Rotations About a General Line**

The usual approach to a rotation by an angle $\alpha$ about a general line is to move the line so it goes through the origin (a translation), rotate the line into the XZ-plane, rotate the resulting line to align with the Z-axis, perform the desired rotation by $\alpha$ now, reverse the two earlier rotations, and then reverse the translation. This requires five rotations and two translations, and suffers from the same

difficulties as the viewing transformation: the angles are not easy to see and the computations are obscure.

The approach above to the viewing transformation extends easily to these rotations. The line is assumed to be given by a point P = (xp,yp,zp) and a direction vector D = <A,B,C>. Then the plane perpendicular to the line at the given point has equation

$$A(x-xp)+B(y-yp)+C(z-zp) = 0.$$

Let T be the translation that moves P to the origin. Pick any point Q = (x,y,z) in the plane and let UP = (x-xp,y-yp,z-zp), as shown in Figure 4. Let N be the result when D is normalized, and let V be the result when UP is normalized. Then compute U as in the viewing transformation to complete the (U,V,N) triple, and build the change-of-basis matrix M with U, V, and N as the first, second, and third columns, respectively.

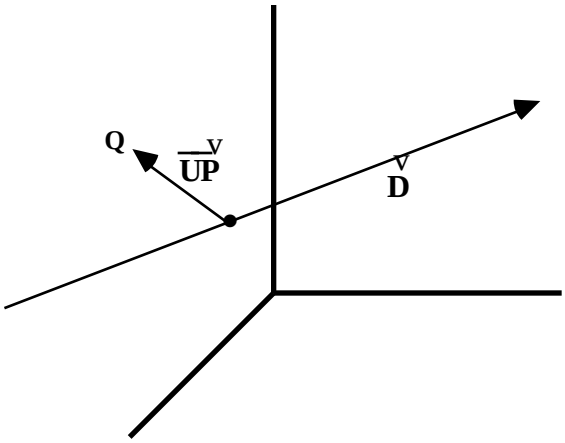


Figure 4: The Setup for General Rotations

Now let R be the matrix of the rotation by $\alpha$ about the Z-axis, and let N and S be the (trivially computed) inverses of M and T respectively. Then the matrix of the rotation by Q about the line is T M R N S, which requires fewer matrices and less computation than the traditional method of rotation about the coordinate axes. There is another benefit for students and for floating-point speed: it uses no trigonometric functions except those in the matrix R.

## Pseudocode for Constructing the Viewing Transformation

The following pseudocode produces the matrix for the viewing transformation with the three standard view specification points as input. It is expanded into actual C code in an appendix. No pseudocode is given for the general rotation, since the matrices M and N of the previous section are easily computed by the pseucode above (and the code in the appendix) if the last translation step is omitted.

```
BuildViewTransform( VRP, EP, UP, T )

      Input:      points  VRP, EP, UP  as in the text
      Output:     transformation matrix  T

      Compute vector  N ← EP – VRP  and normalize  N
      Compute vector  V ← VP – VRP
      Make vector  V  orthogonal to  N  and normalize  V
      Compute vector  U ← V x N  (cross product)
      Write the vectors U, V, and N as the first three rows of the
            first, second, and third columns of  T, respectively,
      Compute the fourth row of  T  to include the translation of
            VRP  to the origin
```

## References

[1] Berger, Mark. *Computer Graphics with Pascal*. Benjamin/Cummings, Menlo Park, CA, 1986.

[2] Foley, James D. and Andries Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, MA, 1982.

[3] Hearn, Donald and M. Pauline Baker. *Computer Graphics*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[4] Newman, William M. and Robert F. Sproull, *Principles of Interactive Computer Graphics*, Second Edition. McGraw-Hill, New York, 1979.

[5] Salmon, Rod and Mel Slater, *Computer Graphics: Systems and Concepts*. Addison-Wesley, Reading, MA, 1987.

## Appendix: C Code for Constructing the Viewing Transformation

This code contains the pseudocode from the article above as comments, showing just how the pseudocode was expanded into the actual code.

```
#include "GraphicsGems.h"      /* Header file for Graphics Gems */

/* Transformations are presented as 4 by 3 matrices, omitting the
 * fourth column to save memory.
 *
 * Functions are used from the Graphics Gems vector C library
 */

typedef float Transform[4][3];

void BuildViewTransform( VRP, EP, UP, T )
    Point3 VRP, EP, UP;
    Transform T;
{
    Vector3  U, V, N;
    float    dot;

    // Compute vector  N ← EP – VRP  and normalize  N
    N.x = EP.x - VRP.x; N.y = EP.y - VRP.y; N.z = EP.z - VRP.z;
    V3Normalize(N);

    /* Compute vector  V ← VP – VRP
     * Make vector  V  orthogonal to  N  and normalize  V
     */
    V.x = UP.x - VRP.x; V.y = UP.y - VRP.y; V.z = UP.z - VRP.z;
    dot = V3Dot(V,N);
    V.x -= dot * N.x; V.y -= dot * N.y; V.z -= dot * N.z;
    V3Normalize(V);

    // Compute vector  U ← V x N  (cross product)
    V3Cross(V,N,U);

    /* Write the vectors U, V, and N as the first three rows of the
     *       first, second, and third columns of  T, respectively
     */
    T[0][0] = U.x;        /* column 1 ← vector U */
    T[1][0] = U.y;
    T[2][0] = U.z;
    T[0][1] = V.x;        /* column 2 ← vector V */
    T[1][1] = V.y;
    T[2][1] = V.z;
    T[0][2] = N.x;        /* column 3 ← vector N */
    T[1][2] = N.y;
    T[2][2] = N.z;

    /* Compute the fourth row of  T  to include the translation of
     *       VRP  to the origin
     */
    T[3][0] = - U.x * VRP.x - U.y * VRP.y - U.z * VRP.z;
    T[3][1] = - V.x * VRP.x - V.y * VRP.y - V.z * VRP.z;
    T[3][2] = - N.x * VRP.x - N.y * VRP.y - N.z * VRP.z;
    return;
}
```