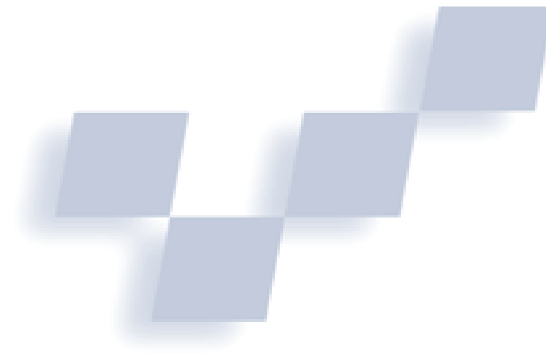
Applications of Vector Geometry for Robustness and Speed



James R. Miller
The University of Kansas

Implementing and using computer graphics and modeling systems rely on mathematical operations on points and vectors. Part 1 of this tutorial in the previous issue (May/June 1999) described the basic language and tools of vector geometric analysis. Here, Part 2 shows how to use those concepts to derive and implement common graphics and modeling operations that maximize speed and numerical reliability.

In Part 1 of this tutorial I described the basic language of vector geometry and made certain claims about its effectiveness when deriving geometric expressions for important algorithms in computer graphics and geometric modeling.[1] Here I provide specific examples of using these techniques, showing representative derivations and resulting computer code. The C++ code illustrates the use of the primitive point, vector, and matrix operations given in Part 1. Taken together, this article and its companion represent a condensed version of a technical report with expanded applications.[2]

Vector geometric techniques represent and manipulate intrinsic relationships between objects independent of coordinate systems. Their applications and resulting computer code demonstrate their usefulness.

Recall we describe as *vector geometric* that class of techniques based on representing and manipulating intrinsic relationships between objects independent of any coordinate system, such as the centroid of a group of points or the vector normal to two others. By contrast, coordinate-based approaches generally operate by comparing and manipulating individual x, y, and z coordinates of points. For example, a particular algorithm may select one of two points based on whose z coordinate is larger.

It turns out that vector geometric techniques work best when operating on objects whose position and orientation with respect to the current coordinate system remain completely general, and when no axis of the coordinate system has any special relationship to the current problem of interest. Coordinate-based methods work better when we either know a priori that the geometry occupies a known, simple position in the coordinate system, or we preprocess it so that it does.

First I'll review required mathematical concepts and tools, then look at using these tools to generate the affine transformations that implement certain common modeling transformations. I'll quantify my claims that you can easily develop computer implementations of these methods and that they maximize numerical reliability and speed. Finally, I'll illustrate a few sample applications of vector geometry outside the context of modeling and viewing transformations.

A quick review and more

The first half of this tutorial[1] developed a characterization of points and vectors as well as the set of operations well defined on them. We saw that we apply a general affine transformation X to affine points as

$$X(Q) = \mathbf{M}Q + \mathbf{t}$$

where $\mathbf{M}$ represents a 3×3 matrix capturing the transformation's rotation, scale, and shear aspects, and $\mathbf{t}$, a vector in the associated vector space, describes the translation component. If we embed Q in projective space, we can express this same transformation as

$$X^P(Q^P) = \begin{pmatrix} Q'_x \\ Q'_y \\ Q'_z \\ 1 \end{pmatrix} = \mathbf{M}_{4\times4}Q^P = \begin{pmatrix} & \mathbf{M} & & t_x \\ & & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix}$$

Graphics systems such as Programmer's Hierarchical Interactive Graphics System (PHIGS)[3] and OpenGL[4] use this convention.

We also saw that we can apply this same affine transformation to vectors as

$$X(\mathbf{v}) = \mathbf{M}\mathbf{v} \tag{1}$$

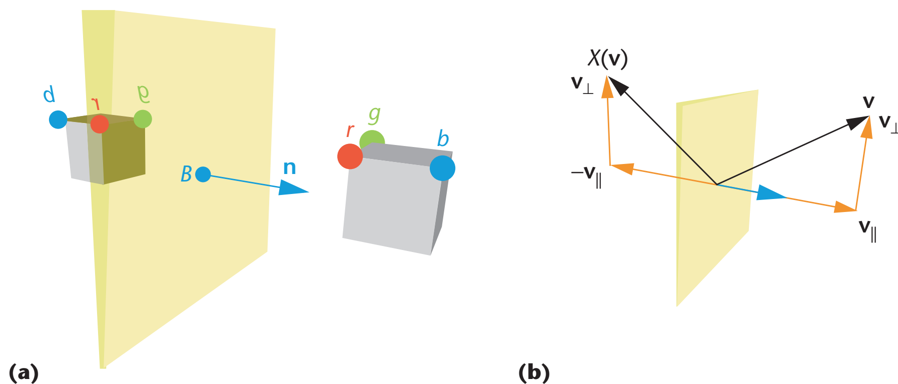
Note that we don't use the translation vector $\mathbf{t}$ when applying the transformation to vectors.

0272-1716/99/$10.00 © 1999 IEEE

My main focus in this article is to show how we can derive **M** and **t** from various types of transformation specifications. The fact that Equation 1 doesn't depend on **t** enables a generic method of computing **t** for a given transformation. For example, suppose we have determined the 3×3 matrix **M** that represents the effect of some affine transformation on vectors. Suppose further that we know a fixed point F of the transformation. Without knowing anything about what the affine transformation does, we can mechanically compute the translation component **t** in terms of the fixed point F:

$$X(F) = F = \mathbf{M}F + \mathbf{t} \Rightarrow \mathbf{t} = F - \mathbf{M}F \tag{2}$$

Determining fixed points for modeling transformations proves trivial. Any point on a rotation axis, for example, is a fixed point of the corresponding rotation affine transformation. Thus we focus on characterizing how a transformation affects vectors. We can then mechanically complete the specification of the corresponding affine transformation for points by using a fixed point as indicated above.



1 (a) A mirror transformation and its effect on points *r*, *g*, and *b*. (b) Visualizing the effect of the mirror transformation on vectors.

Example 1: A mirror affine transformation

Mirror transformations—common modeling operations—flip the "handedness" of geometry. For example, if I have a model of the left rear door of a car, I can apply a mirror transformation to obtain the geometry of the right rear door. We can construct a general mirror transformation from a combination of rotation, translation, and scale transformations. However, since it's such a common operation and since specifying the mirror plane directly proves so much simpler than determining and assembling the required component rotation, translation, and scaling transformations, we treat it directly here.

We define the mirror transformation by specifying a plane in space to use as the mirror. Defining this mirror plane typically involves specifying a point B on the plane and a vector **n** perpendicular to the plane, as illustrated in Figure 1a. Recall that we can't assume the plane has any particular relationship with respect to the coordinate axes. In our car example, the designer would give us the plane that cuts the car in half, splitting the driver's and passenger's sides. That is, B would represent some point in the middle of the car, perhaps where the hood ornament attaches to the hood. The vector **n** would represent a vector pointing from the left side of the car to the right side. (As required, the transformation we derive stays the same if the vector points instead from the right side to the left side.) The mirror transformation will then map a point on one side of the plane to its mirror image on the other side, as shown in Figure 1a.

Goldman stated without proof a 4×4 matrix defined only in terms of B and **n** that represents this transformation.[5] The remainder of this example illustrates how to use the vector geometric tools we have studied to derive this matrix. The example here concludes with sample C++ code implementing the formula.

As explained in the previous section, we focus first on how to mirror a vector. The expressions we derive and the subsequent computer implementation become somewhat easier if we use a unit vector for the plane normal. We therefore begin by computing $\hat{\mathbf{n}}$, the unit vector in the direction of **n**. (In the computer implementation, we check at this point to make sure that **n** is not a zero vector—the only error condition or special case that can arise.)

We can write an arbitrary vector **v** as the sum of two vectors, one parallel to and the other perpendicular to the unit plane normal $\hat{\mathbf{n}}$. As indicated in Figure 1b, the mirror transformation has the effect of negating the component parallel to $\hat{\mathbf{n}}$; it leaves the component perpendicular to $\hat{\mathbf{n}}$ unchanged. Therefore,

$$\begin{aligned} X(\mathbf{v}) &= X(\mathbf{v}_{\parallel} + \mathbf{v}_{\perp}) \\ &= X(\mathbf{v}_{\parallel}) + X(\mathbf{v}_{\perp}) \\ &= -\mathbf{v}_{\parallel} + \mathbf{v}_{\perp} \end{aligned}$$

We compute the components of **v** parallel and perpendicular to $\hat{\mathbf{n}}$ as $\mathbf{v}_{\parallel} = (\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$ and $\mathbf{v}_{\perp} = (\mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}})$. Substituting these expressions into the equation above, we find

$$X(\mathbf{v}) = \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

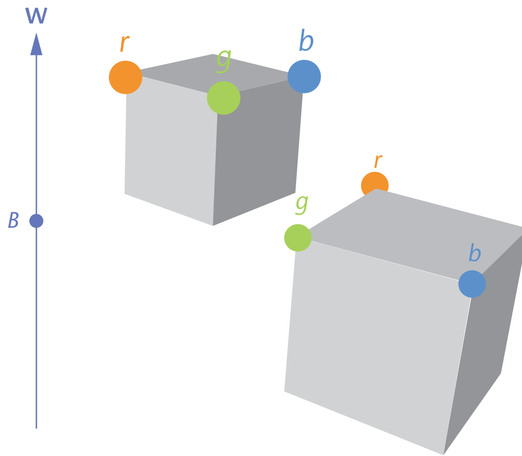
A direct implementation of this equation most efficiently mirrors an individual vector **v**. If instead we need a matrix representation (for example, for use as a modeling transformation matrix in a graphics pipeline), then we need to write the dot product operations in matrix form. It's straightforward to demonstrate that we can write this as

$$X(\mathbf{v}) = \mathbf{M}\mathbf{v}$$

where

$$\mathbf{M} = \mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}})$$

"**I**" represents the 3×3 identity matrix, and "$\otimes$" is the



2 A rotation transformation in 3D affine space and its effects on points *r*, *g*, and *b*.

tensor product operator. The tensor product ($\hat{\mathbf{n}} \otimes \hat{\mathbf{n}}$) is a 3×3 matrix in which the (i, j)th element is $n_i n_j$. That is, if we consider $\hat{\mathbf{n}}$ as a column vector, then $\hat{\mathbf{n}} \otimes \hat{\mathbf{n}} = \hat{\mathbf{n}} * \hat{\mathbf{n}}^T$. Finally, we compute $\mathbf{t}$ using Equation 2 with F set to the mirror plane point, B.

By now it should be clear that the claim made earlier holds. That is, $\mathbf{M}$ and $\mathbf{t}$ remain unaffected if $-\hat{\mathbf{n}}$ replaces $\hat{\mathbf{n}}$.

In the earlier technical report[2] I showed C++ class definitions for points, vectors, and matrices with methods and overloaded operators implementing the basic operations in Part 1 of this tutorial.[1] Using these definitions, we can write a C++ function that creates the 3×3 matrix $\mathbf{M}$ and the translation vector $\mathbf{t}$ for this affine transformation:

```
bool BuildMirrorTransformation(
          aPoint B, aVector n,
          Matrix3x3& M, aVector& t)
{   aVector nHat;
    double length =
              n.normalizeToCopy(nHat);
    if (length < tolerance)
        // a zero vector was
        // provided for n. We
        // cannot proceed.
        return false;
    Matrix3x3 T =
       Matrix3x3::tensorProductMatrix
                          (nHat,nHat);
    M = Matrix3x3::IdentityMatrix -
        2.0*T;
    // The point B is a fixed point
    // of the transformation, hence:
    t = B - M*B;
    return true;
}
```

Example 2: A general rotation affine transformation

We define a rotation transformation in 3D affine space as a rotation by an angle θ about an axis passing through a point B with direction given by a unit vector $\hat{\mathbf{w}}$. (See Figure 2.) Goldman[5] stated the following formula without proof:

$$\mathbf{M} = \cos\theta \mathbf{I} + (1-\cos\theta)\hat{\mathbf{w}} \otimes \hat{\mathbf{w}} + \sin\theta \mathbf{W}$$
$$\mathbf{t} = B - \mathbf{M}B$$

where

$$\mathbf{W} = \begin{pmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{pmatrix}$$

As with mirror transformations, a derivation of this formula begins by considering an arbitrary vector $\mathbf{v}$ and characterizing the rotation of its components parallel and perpendicular to $\hat{\mathbf{w}}$.[2] (See Figure 3.) In this case, the parallel component remains unchanged, while the effect on the perpendicular component can be characterized with a 2D geometric analysis as indicated in the middle of Figure 3.

I'll refer to the computer-based implementation of these expressions in the next section when discussing ease of implementation, robustness, and computational efficiency.

On efficiency, simplicity of implementation, and robustness

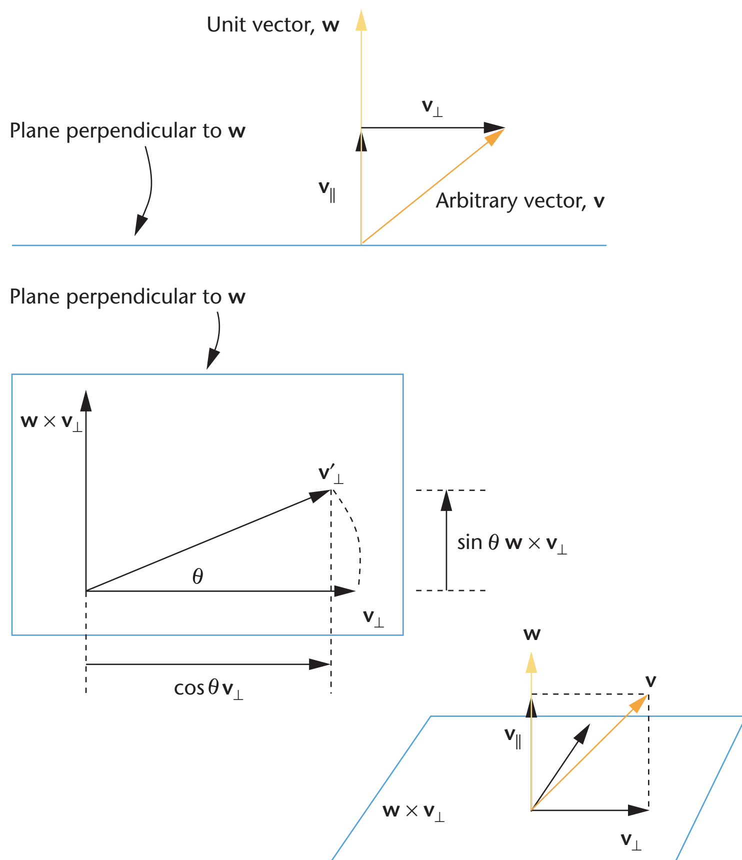
This approach to developing modeling transformations relies solely on vector geometric analysis. As a major benefit, the final computer code you write ends up much simpler, more compact, and free of nasty special-case handling. It typically involves fewer arithmetic operations as well.

I won't obsess here with exhaustive operation counts, but I will make some general observations to justify the claim that this method usually requires fewer arithmetic operations. Consider the general rotation matrix $\mathbf{M}$ presented in Example 2. Given the sine and cosine of the rotation angle (required by any approach), it takes 36 multiplies and 19 adds for a completely dumb implementation of this equation. By "dumb" I mean that we don't try to exploit any special properties of the terms such as the symmetry of $\hat{\mathbf{w}} \otimes \hat{\mathbf{w}}$, the zeros in the $\mathbf{W}$ matrix, or the fact that we can actually generate $\cos\theta\mathbf{I}$ with no multiplications. If instead we symbolically expand this equation and implement the result, we can compute $\mathbf{M}$ with just 15 multiplies and 10 adds. Computing the translation vector $\mathbf{t}$ then requires an additional 9 multiplies and 9 adds. Therefore the total required to compute $(\mathbf{M}, \mathbf{t})$ equals 45 multiplies and 28 adds for the dumb approach, or 24 multiplies and 19 adds for the optimized approach.

The traditional coordinate-based algorithm described in the standard texts embodies a "reduce to the previously solved problem" approach. It works by combining primitive affine transformations to achieve more general ones. The cost for concatenating two affine transformations is 36 multiplies and 27 adds, approximately 90 percent the cost of our "completely dumb" implementation and 50 percent more expensive than our optimized general implementation. Remember, however, this compares between a complete computation of the rotation matrix using the results of the vector geometric analysis as shown in Example 2 versus a single concatenation of

a pair of affine transformations. The traditional "reduce to the previously solved problem" approach requires several concatenations of affine transformations. To do a rotation about a general axis in space, for example, you need to combine seven affine transformations (that is, perform six concatenations of affine transformations). Therefore, creating a general rotation matrix would require 216 multiplies and 162 adds. Clearly, this proves far more expensive than the vector geometric approach.

As we saw in Part 1 of this tutorial,[1] the traditional approach is even more complex in that you must "guard" the construction of the various matrices by checking for certain special cases like distances computed for denominators being zero or nearly zero. By contrast, once we determine that we haven't gotten the zero vector for a rotation axis (analogous to the initial test in the `BuildMirrorTransformation` function of Example 1), we don't need any special-case detection to implement our vector geometric equation for **M**. Therefore, while it's true that we can best implement generating a "primitive" rotation about the *x*-, *y*-, or *z*-axis by directly writing code to construct the appropriate matrix, anything more general is best done with a direct implementation of the equation given in Example 2.



3 Rotating a vector **v** about a unit axis vector **w** by θ.

Viewing transformations

The examples above illustrate applying vector geometric analysis to generating modeling transformations. We can apply these same techniques to generate so-called "viewing transformations," the affine transformations that map world coordinates to the "eye coordinate system" used in computer graphics systems. No mathematical difference exists between modeling and viewing transformations—both simply transform geometry from one coordinate system into another. The difference is purely conceptual and relates to the interface presented, either to a programmer or to an end user. That is, modeling and viewing transformations typically differ in terms of the most natural way to specify the desired transformation. For example, rather than instructing the graphics system to rotate and then translate geometry, the graphics programmer (or end user) wants to say "I am standing here and looking in that direction. What do I see?"

Instead of an eye coordinate system, PHIGS employs a slightly more general coordinate system called the view reference coordinate system (VRC).[3] The primary difference is that the eye's position need not lie at the origin of VRC. In PHIGS, the so-called view orientation transformation maps world coordinates to VRC. You would define it by specifying a view reference point *V*, a view plane normal vector **n**, and an up vector **v**.[3] From **n** and **v**, we compute the three mutually perpendicular unit vectors defining the VRC axes as measured in world coordinate space:

$$\hat{\mathbf{n}} = \frac{\mathbf{n}}{|\mathbf{n}|}$$

$$\hat{\mathbf{v}} = \text{normalize}(\mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}})$$

Notice that $\hat{\mathbf{v}}$ will be a unit vector in the direction of the component of **v** perpendicular to $\hat{\mathbf{n}}$. We thus compute the final axis direction as

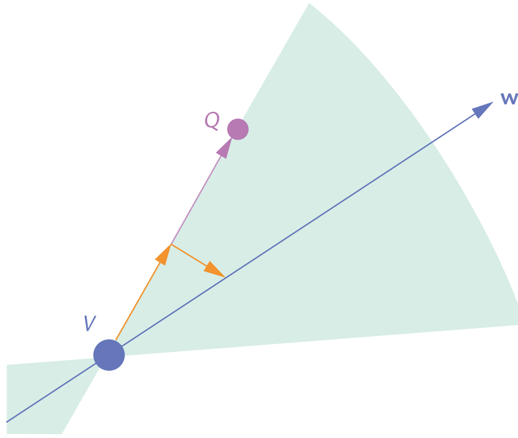
$$\hat{\mathbf{u}} = \hat{\mathbf{v}} \times \hat{\mathbf{n}}$$

Given these mutually perpendicular unit vectors, we write the 3×3 matrix **M** of the affine transformation describing the view orientation operation as

$$\mathbf{M} = \begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{pmatrix}$$

Now, what about **t**? In general, there exists no fixed point of a view orientation transformation. However, the view reference point *V* is defined as the origin of VRC, hence it must get mapped to (0, 0, 0). That is,

$$X(V) = (0, 0, 0) = \mathbf{M}V + \mathbf{t} \Rightarrow \mathbf{t} = -\mathbf{M}V$$



4 Finding the outward pointing normal to a right circular cone at a point *Q*.

We transform geometry into the viewing coordinate system because this coordinate system has a special relationship to the display device.[6] Exploiting these special properties can make rendering operations considerably more efficient and simpler to write. Subsequent analysis is therefore best accomplished using coordinate-based schemes that exploit the special properties built into the viewing coordinate system. See elsewhere for further details and examples.[2,6]

Other common operations

We can easily express many other queries and operations common in graphics and geometric modeling using similar applications of vector geometric techniques. We frequently need computations involving distances and signed distances between various combinations of points, lines, planes, and other basic curves and surfaces. The first two examples here show C++ code using vector geometric operations for two common distance queries. Derivations for these and similar queries appear in the literature.[2,7,8] I leave others as an exercise.

We can characterize a plane in space by a point *B* on the plane and a vector **n** perpendicular to the plane (for example, the mirror plane of Figure 1a). Given such a plane, we define the signed distance of an arbitrary point *Q* from the plane as the actual distance if *Q* lies on the side of the plane to which **n** points and the negative of this distance if it lies on the other side. We can write a C++ routine using the point and vector utilities described in the earlier technical report[2] and implementing this definition as

```
bool signedDistancePntPlane(
   // the point:
   aPoint Q,
   // the plane:
   aPoint B, aVector n,
   // the computed signed distance:
   double& signedDist)
{  aVector nHat;
   double length =
            n.normalizeToCopy(nHat);
   if (length < tolerance)
      // a zero vector was provided
      // for n. We cannot proceed.
      return false;
   signedDist =
      aVector::dot( (Q-B) , nHat );
   return true;
}
```

We can define a line in space by a point *B* on the line and a vector **w** specifying the direction of the line (for example, the rotation axis of Figure 2). We frequently need to compute the distance between an arbitrary point *Q* and the line. We could adopt an approach based on the Pythagorean Theorem by considering the right triangle formed by *B*, *Q*, and the perpendicular projection of *Q* onto the line. In the following code, however, we compute the component of the vector (*Q*–*B*) perpendicular to **w**, then return the length of this vector:

```
bool distancePntLine(
   // the point:
   aPoint Q,
   // the line:
   aPoint B, aVector w,
   // the computed distance:
   double& distance)
{  aVector wHat;
   double length =
            w.normalizeToCopy(wHat);
   if (length < tolerance)
      // a zero vector was provided
      // for w. We cannot proceed.
      return false;
   aVector vParallel, vPerpendicular;
   wHat.decompose( (Q-B),
         vParallel, vPerpendicular );
   distance =
            vPerpendicular.length();
   return true;
}
```

It's frequently necessary to compute a unit vector perpendicular to a surface at a given point on the surface. Intersection computations and rendering algorithms offer two examples of the use of this utility. Suppose we wish to compute the outward pointing normal to a cylinder—defined by its axis ($B,\hat{\mathbf{w}}$) and radius *r*—at a point *Q* on the cylinder's surface. The implementation would be identical to what we just saw in `distancePntLine`, except that rather than computing the length of `vPerpendicular` at the end, we would normalize it and return it as the unit outward pointing normal. Furthermore, if *Q* didn't precisely lie on the cylinder, the normal so computed would be the normal for the point on the cylinder closest to *Q*. This extended notion of "normal at *Q*" only fails if *Q* lies on the cylinder's axis, that is, if `vPerpendicular` is the zero vector.

A slightly more complicated but still relatively straightforward example finds the outward pointing normal to a right circular cone at a point *Q* on the cone different from the vertex. Up to a sign, the desired normal is the component of the cone axis vector perpen-

dicular to the ruling containing Q. In Figure 4, the red vector along the cone ruling represents **`wParallel`**; the red vector pointing into the cone represents **`wPerpendicular`**.

```
bool normalToCone(
   // the point:
   aPoint Q,
   // the cone axis (vertex=V):
   aPoint V, aVector w,
   // the computed outward
   // pointing normal vector:
   aVector& normal)
{  aVector ruling = Q - V;
   double d = ruling.normalize();
   if (d < tolerance)
   // Q is at the vertex. We
   // cannot proceed.
       return false;
   aVector wParallel, wPerpendicular;
   ruling.decompose( w,
          wParallel, wPerpendicular );
   d = wPerpendicular.normalizeToCopy
                              (normal);
   if (d < tolerance)
   // Q is on the cone axis -OR-
   // w is the zero vector. We
   // cannot proceed in either case.
       return false;
   // Invert the normal if necessary.
   // (The outward pointing normal
   // must point away from the cone
   // axis.)
   if (aVector::dot(w,ruling) > 0.0)
       normal  =  -normal;
   return true;
}
```

Conclusion

Somewhat more detailed derivations of the examples presented here as well as those for other transformations and operations appear in my earlier technical report.[2] That report also describes coordinate-based schemes for those situations they better suit.

We have seen that vector geometric approaches prove most appropriate in situations where we can make no assumptions about how geometry relates to coordinate systems. Vector geometric methods operate by expressing coordinate-system-independent relationships between points and vectors. Not only did these expressions not rely on special configurations of the geometry with respect to each other or to the coordinate system, but also their implementations in computer code didn't require special-case handling in the event that such special relationships occurred. This proved a powerful analytical technique that led to highly efficient and robust computer implementations.

However, some situations benefit from transforming geometry into special coordinate systems where we can exploit the system's orientation to dramatically simplify and accelerate certain types of imaging operations. Examples include clipping algorithms, visible line and visible surface determination, and intensity depth cueing. You can find technical details on implementing those operations in standard references.[4,6] ■

Acknowledgments

Ron Goldman first showed me the derivation behind Example 2 when we were working together in the early 1980s, and his expression for the matrix later appeared in the first of the *Graphics Gems* series.[5] I have since seen similar expressions for the final matrix of Example 2.[4] I would also like to thank Ron for his comments on an earlier draft of this material. The development of the software implementing the various techniques described here was performed in DesignLab, a multidisciplinary research laboratory at the University of Kansas funded in part by NSF Grant CDA-94-01021. Valuable suggestions were also provided by the reviewers.

References

1. J.R. Miller, "Vector Geometry for Computer Graphics," *IEEE CG&A*, Vol. 19, No. 3, May/June 1999, pp. 66-73.
2. J.R. Miller, "The Mathematics of Graphical Transformations: Vector Geometric and Coordinate-Based Approaches," DesignLab Tech. Report DL-1997-03, Univ. of Kansas, Lawrence, Ks., Jan. 1997.
3. T.L.J. Howard et al., *Practical Introduction in PHIGS and PHIGS Plus*, Addision-Wesley, Reading, Mass., 1991.
4. M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Version 1.1, 2d edition, Addison-Wesley, Reading, Mass., 1997.
5. R.N. Goldman, "Matrices and Transformations," in *Graphics Gems*, A.S. Glassner, ed., Academic Press, San Diego, 1990, pp. 472-475.
6. J.D. Foley et al., *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, Mass., 1990.
7. A.S. Glassner, "Useful 3D Geometry," in *Graphics Gems*, A.S. Glassner, ed., Academic Press, San Diego, 1990, pp. 297-300.
8. P. Georgiades, "Signed Distance from Point to Plane," in *Graphics Gems III*, D. Kirk, ed., Academic Press, San Diego, 1992, pp. 223-224.

***James R. Miller** is an associate professor of computer science in the Department of Electrical Engineering and Computer Science at the University of Kansas. His current research interests include computer graphics; geometric modeling; visualization; the use of computers and technology in education; and programming paradigms, especially object-oriented techniques and their applicability to problems in graphics, modeling, and visualization. He earned a BS in computer science from Iowa State University, and holds MS and PhD degrees in computer science from Purdue University.*

Readers may contact Miller at Dept. of Electrical Engineering and Computer Science, University of Kansas, 415 Snow Hall, Lawrence, KS 66045-2228, e-mail miller@eecs.ukans.edu.