Foundations
of
Game Engine Development

Foundations
of
Game Engine Development

Volume 1
Mathematics

by Eric Lengyel

Terathon Software LLC
Lincoln, California

Foundations of Game Engine Development
Volume 1: Mathematics

ISBN-13: 978-0-9858117-4-7

Fifth printing

Published by Terathon Software LLC
www.terathon.com

Series website: foundationsofgameenginedev.com

About the cover: The cover image is a scene from a game entitled *The 31st*, developed with the Tombstone Engine. Artwork by Javier Moya Pérez.

3.2 Normal Vectors

A *normal vector*, or just *normal* for short, is a vector that is perpendicular to a surface, and the direction in which it points is said to be *normal* to the surface. A flat plane has only one normal direction, but most surfaces aren't so simple and thus have normal vectors that vary from point to point. Normal vectors are used for a wide variety of reasons in game engine development that include surface shading, collision detection, and physical interaction.

3.2.1 Calculating Normal Vectors

There are a few ways in which normal vectors can be calculated, and the best method in any particular case really depends on how a surface is described from a mathematical standpoint. In the case that a surface is defined implicitly by a scalar function $f(\boldsymbol{p})$, the normal vector at $\boldsymbol{p} = (x, y, z)$ is given by the gradient $\nabla f(\boldsymbol{p})$ because it is perpendicular to every direction tangent to the level surface of f at $\boldsymbol{p}$. For example, suppose we have the ellipsoid shown in Figure 3.2, defined by the equation

$$f(\boldsymbol{p}) = x^2 + \frac{y^2}{4} + z^2 - 1 = 0. \tag{3.2}$$

The point $\boldsymbol{p} = \left(\frac{\sqrt{6}}{4}, 1, \frac{\sqrt{6}}{4}\right)$ lies on the surface of this ellipsoid, and the normal vector $\mathbf{n}$ at that point is given by

$$\mathbf{n} = \nabla f(\boldsymbol{p}) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)\Bigg|_p = \left(2x, \frac{y}{2}, 2z\right) = \left(\frac{\sqrt{6}}{2}, \frac{1}{2}, \frac{\sqrt{6}}{2}\right). \tag{3.3}$$

Calculating normal vectors with the gradient is something that's usually done only in the process of constructing a triangle mesh to approximate an ideal surface described by some mathematical formula. The normal vectors are typically scaled to unit length and stored with the vertex coordinates that they're associated with. Most of the time, a game engine is working with a triangle mesh having an arbitrary shape that was created in a modeling program, and the only information available is the set of vertex coordinates and the list of indices that tell how vertices are grouped into triangles.

As illustrated in Figure 3.3, the normal vector $\mathbf{n}$ for a single triangular face can be calculated by taking the cross product between vectors aligned with two of the

triangle's edges. Let $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$ be the vertices of a triangle wound in the counterclockwise direction. An outward-facing normal vector is then given by

$$\mathbf{n} = (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0). \tag{3.4}$$

Any permutation of the subscripts that keeps them in the same cyclic order produces the same normal vector. It doesn't matter which vertex is chosen to be subtracted from the other two as long as the first factor in the cross product involves the next vertex in the counterclockwise order. If the order is reversed, then the calculated normal vector still lies along the same line, but it points in the opposite direction.

To calculate per-vertex normal vectors for a triangle mesh, it is typical for a game engine's model processing pipeline to calculate all of the per-face normal vectors and then take an average at each vertex over all of the faces that use that vertex. The average may be weighted based on triangle area or other factors to create a smooth field of normal vectors over a curved surface. In cases in which a hard edge is desired, such as for a cube or the pyramid in Figure 3.3, vertex positions are typically duplicated, and different normal vectors corresponding to different faces are associated with the various copies of the vertex coordinates.

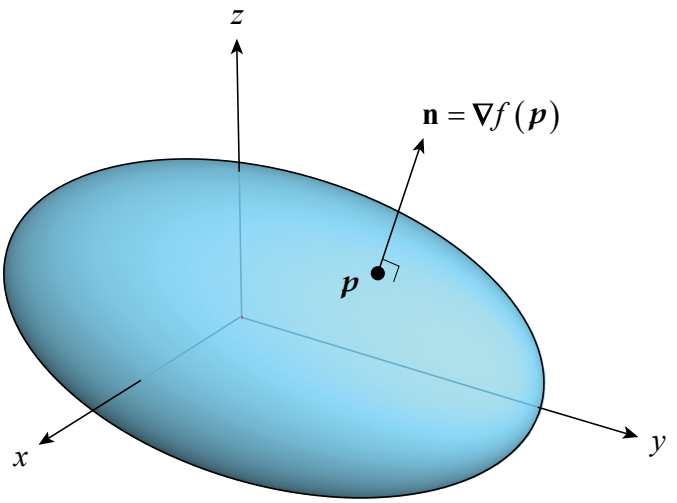

Figure 3.2. The normal vector $\mathbf{n}$ at a particular point $\boldsymbol{p}$ on a surface implicitly defined by the equation $f(\boldsymbol{p}) = 0$ is given by the gradient $\nabla f(\boldsymbol{p})$.

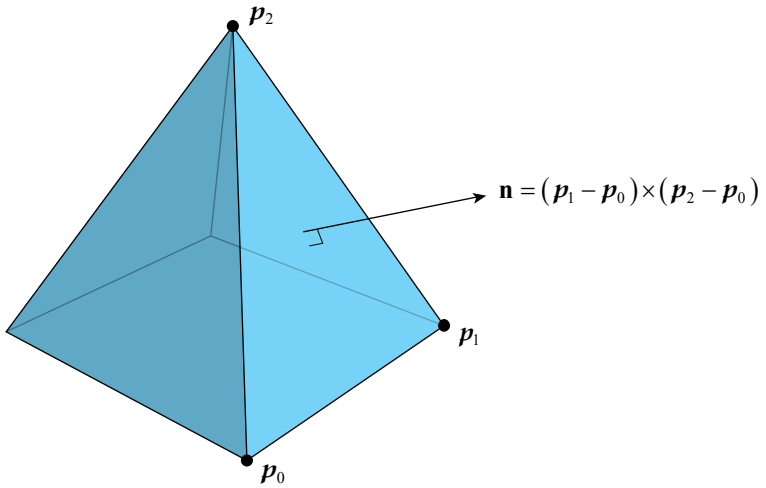

Figure 3.3. The normal vector **n** for a triangular face having vertices $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$ is given by the cross product between vectors corresponding to two edges of the triangle.

3.2.2 Transforming Normal Vectors

When a model is transformed by a matrix **M** in order to alter its geometry, every point $\boldsymbol{p}$ belonging to the original model becomes a point $\mathbf{M}\boldsymbol{p}$ in the transformed model. Since a tangent vector **t** can be approximated by the difference of points $\boldsymbol{p}$ and $\boldsymbol{q}$ on a surface, or is often exactly equal to such a difference, it is transformed in the same way as a point to become **Mt** because the difference between the new points $\mathbf{M}\boldsymbol{p}$ and $\mathbf{M}\boldsymbol{q}$ is tangent to the new surface. Problems arise, however, if we try to apply the same transformation to normal vectors.

Consider the shape shown in Figure 3.4 that has a normal vector **n** on its slanted side. Let **M** be a transformation matrix that scales by a factor of two in the horizontal direction but does not scale in the vertical direction. If the matrix **M** is multiplied by **n**, then the resulting vector **Mn** is stretched horizontally and, as clearly visible in the figure, is no longer perpendicular to the surface. This indicates that something is inherently different about normal vectors, and if we want to preserve perpendicularity, then we must find another way to transform them that produces the correct results. Taking a closer look at how a matrix transforms a vector provides some insight. We restrict our discussion to 3×3 matrices here since normal vectors are not affected by translation, but the same conclusions will apply to 4×4 matrices in the discussion of planes later in this chapter.

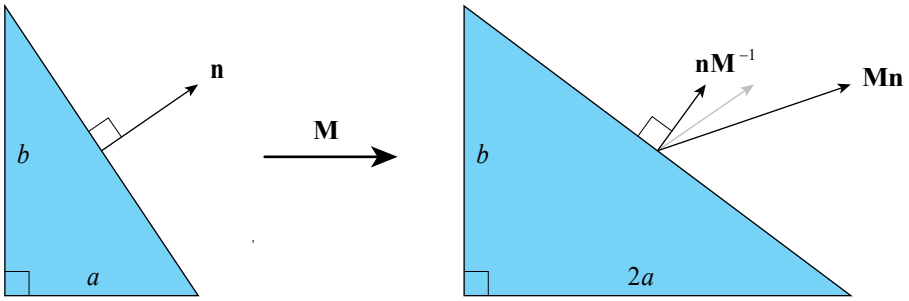

Figure 3.4. A shape is transformed by a matrix **M** that scales by a factor of two only in the horizontal direction. The normal vector **n** is perpendicular to the original surface, but if it is treated as a column vector and transformed by the matrix **M**, then it is not perpendicular to the transformed surface. The normal vector is correctly transformed by treating it as a row vector and multiplying by $\mathbf{M}^{-1}$. (The original normal vector is shown in light gray on the transformed surface.)

Let $\mathbf{v}^A$ be a vector whose coordinates are expressed in coordinate system A as indicated by its superscript, and consider the fact that the components of $\mathbf{v}^A$ measure distances along the coordinate axes. A distance Δx^A along the x direction in coordinate system A is equivalent to the sum of distances Δx^B, Δy^B, and Δz^B along the axis-aligned directions in another coordinate system B. The x component of $\mathbf{v}^A$ can therefore be expressed as the vector

$$\left(\frac{\Delta x^B}{\Delta x^A} v_x^A, \frac{\Delta y^B}{\Delta x^A} v_x^A, \frac{\Delta z^B}{\Delta x^A} v_x^A \right) \tag{3.5}$$

in coordinate system B. Similar expressions with Δy^A and Δz^A in the denominators can be used to express the y and z components of $\mathbf{v}$ in coordinate system B. Adding them up gives us a transformed vector $\mathbf{v}^B$ in coordinate system B that corresponds to the original vector $\mathbf{v}^A$ in its entirety, and this can be written as the matrix transformation

$$\mathbf{v}^B = \begin{bmatrix} \frac{\Delta x^B}{\Delta x^A} & \frac{\Delta x^B}{\Delta y^A} & \frac{\Delta x^B}{\Delta z^A} \\ \frac{\Delta y^B}{\Delta x^A} & \frac{\Delta y^B}{\Delta y^A} & \frac{\Delta y^B}{\Delta z^A} \\ \frac{\Delta z^B}{\Delta x^A} & \frac{\Delta z^B}{\Delta y^A} & \frac{\Delta z^B}{\Delta z^A} \end{bmatrix} \mathbf{v}^A. \tag{3.6}$$

Each entry of this matrix multiplies a component of $\mathbf{v}^A$ by a ratio of axis-aligned distances, and the axis appearing in the denominator of each ratio corresponds to the component of $\mathbf{v}^A$ by which the ratio is multiplied. This has the effect of cancelling the distances in coordinate system A and replacing them with distances in coordinate system B.

Now let us consider a normal vector that was calculated as a gradient. The key to understanding how such normal vectors transform is realizing that the components of a gradient do not measure distances along the coordinate axes, but instead measure *reciprocal* distances. In the partial derivatives that compose a vector $(\partial f/\partial x, \partial f/\partial y, \partial f/\partial z)$, distances along the x, y, and z axes appear in the denominators. This is fundamentally different from the measurements made by the components of an ordinary vector, and it's the source of the problem exemplified by the nonuniform scale shown in Figure 3.4. Whereas an ordinary vector $\mathbf{v}$ is treated as a *column* matrix with components (v_x, v_y, v_z), we write a normal vector $\mathbf{n}$ as the *row* matrix

$$\mathbf{n} = \begin{bmatrix} \frac{1}{n_x} & \frac{1}{n_y} & \frac{1}{n_z} \end{bmatrix}. \tag{3.7}$$

It then becomes apparent that multiplying this vector on the right by the matrix in Equation (3.6) has the effect of cancelling reciprocal distances in coordinate system B and replacing them with reciprocal distances in coordinate system A. Calling the matrix $\mathbf{M}$, we can state that it is *simultaneously* the transform that takes ordinary vectors from A to B through the product $\mathbf{v}^B = \mathbf{M}\mathbf{v}^A$ *and* the transform that takes normal vectors, in the opposite sense, from B to A through the product $\mathbf{n}^A = \mathbf{n}^B\mathbf{M}$. Inverting the matrix reverses both of these transformations, so we conclude that the correct transformation from A to B for a normal vector, at least one calculated with a gradient, is given by

$$\boxed{\mathbf{n}^B = \mathbf{n}^A\mathbf{M}^{-1}.} \tag{3.8}$$

The correctness of Equation (3.8) can be verified by demonstrating that a transformed normal vector remains perpendicular to any transformed tangent vector. Suppose that $\mathbf{n}^A$ and $\mathbf{t}^A$ are normal and tangent to a surface at some point in coordinate system A. By definition, they are perpendicular, and we must have $\mathbf{n}^A \cdot \mathbf{t}^A = 0$. (Since $\mathbf{n}^A$ is a row vector and $\mathbf{t}^A$ is a column vector, the matrix product $\mathbf{n}^A\mathbf{t}^A$ is actually what we're calculating here, but the dot is still included by convention, even though the notation is not technically correct, to make it clear that we are producing a scalar quantity.) Let $\mathbf{M}$ be a matrix that transforms ordinary

vectors from coordinate system A to coordinate system B. Then the transformed normal $\mathbf{n}^B$ is given by $\mathbf{n}^A\mathbf{M}^{-1}$, and the transformed tangent $\mathbf{t}^B$ is given by $\mathbf{M}\mathbf{t}^A$. Their product is

$$\mathbf{n}^B \cdot \mathbf{t}^B = \mathbf{n}^A\mathbf{M}^{-1}\mathbf{M}\mathbf{t}^A = \mathbf{n}^A \cdot \mathbf{t}^A = 0, \tag{3.9}$$

and this establishes the fact that they are still perpendicular in coordinate system B after the transformation by $\mathbf{M}$.

Getting back to the example in Figure 3.4, the transformation matrix $\mathbf{M}$ is

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

when we align the x axis with the horizontal direction and the y axis with the vertical direction. We can take the normal vector before the transformation to be $\mathbf{n} = [b \quad a \quad 0]$. Transforming this with Equation (3.8) gives us a new normal vector equal to $[b/2 \quad a \quad 0]$, which is perpendicular to the transformed surface. The important observation to make is that the matrix $\mathbf{M}$ scales x values by a factor of two, but because normal vectors use reciprocal coordinates as shown in Equation (3.7), multiplying n_x by two is equivalent to multiplying the x component of $\mathbf{n}$ by a factor of one half, which is exactly what $\mathbf{M}^{-1}$ does.

In the case that a normal vector $\mathbf{n}^A$ is calculated as the cross product $\mathbf{s} \times \mathbf{t}$ between two tangent vectors $\mathbf{s}$ and $\mathbf{t}$, the transformed normal vector $\mathbf{n}^B$ should be equal to the cross product between the transformed tangent vectors. Again, let $\mathbf{M}$ be a matrix that transforms ordinary vectors from coordinate system A to coordinate system B. Then $\mathbf{n}^B = \mathbf{M}\mathbf{s} \times \mathbf{M}\mathbf{t}$, but we need to be able to calculate $\mathbf{n}^B$ without any knowledge of the vectors $\mathbf{s}$ and $\mathbf{t}$. Expanding the matrix-vector products by columns with Equation (1.28), we can write

$$\mathbf{n}^B = \left(s_x\mathbf{M}_{[0]} + s_y\mathbf{M}_{[1]} + s_z\mathbf{M}_{[2]}\right) \times \left(t_x\mathbf{M}_{[0]} + t_y\mathbf{M}_{[1]} + t_z\mathbf{M}_{[2]}\right), \tag{3.11}$$

where we are using the notation $\mathbf{M}_{[j]}$ to mean column j of the matrix $\mathbf{M}$ (matching the meaning of the [ ] operator in our matrix data structures). After distributing the cross product to all of these terms and simplifying, we arrive at

$$\begin{aligned} \mathbf{n}^B = &\left(s_y t_z - s_z t_y\right)\left(\mathbf{M}_{[1]} \times \mathbf{M}_{[2]}\right) \\ &+ \left(s_z t_x - s_x t_z\right)\left(\mathbf{M}_{[2]} \times \mathbf{M}_{[0]}\right) \\ &+ \left(s_x t_y - s_y t_x\right)\left(\mathbf{M}_{[0]} \times \mathbf{M}_{[1]}\right). \end{aligned} \tag{3.12}$$

The cross product $\mathbf{n}^A = \mathbf{s} \times \mathbf{t}$ is clearly visible here, but it may be a little less obvious that the cross products of the matrix columns form the rows of $\det(\mathbf{M})\mathbf{M}^{-1}$, which follows from Equation (1.95). We conclude that a normal vector calculated with a cross product is correctly transformed according to

$$\boxed{\mathbf{n}^B = \mathbf{n}^A \det(\mathbf{M})\mathbf{M}^{-1}.} \tag{3.13}$$

Using the adjugate of $\mathbf{M}$, defined in Section 1.7.5, we can also write this as

$$\boxed{\mathbf{n}^B = \mathbf{n}^A \operatorname{adj}(\mathbf{M}).} \tag{3.14}$$

This is not only how normal vectors transform, but it's how *any* vector resulting from a cross product between ordinary vectors transforms.

Equation (3.13) differs from Equation (3.8) only by the additional factor of $\det(\mathbf{M})$, showing that the two types of normal vectors are closely related. Since normal vectors are almost always rescaled to unit length after they're calculated, in practice, the size of $\det(\mathbf{M})$ is inconsequential and often ignored, making the two normal vector transformation equations identical. However, there is one situation in which $\det(\mathbf{M})$ may have an impact, and that is the case when the transform performed by $\mathbf{M}$ contains a reflection. When the vertices of a triangle are reflected in a mirror, their winding orientation is reversed, and this causes a normal vector calculated with the cross product of the triangle's edges to reverse direction as well. This is exactly the effect that a negative determinant of $\mathbf{M}$ would have on a normal vector that is transformed by Equation (3.13).

The code in Listing 3.1 multiplies a normal vector, stored in a `Vector3D` data structure and treated as a row matrix, by a `Transform4D` data structure on the right, but it ignores the fourth column of the transformation matrix. When positions and normals are being transformed by a 4×4 matrix $\mathbf{H}$, a point $\boldsymbol{p}$ is transformed as $\mathbf{H}\boldsymbol{p}$, but a normal $\mathbf{n}$ has to be transformed as $\mathbf{n}\mathbf{M}^{-1}$, where $\mathbf{M}$ is the upper-left 3×3 portion of $\mathbf{H}$. Being able to multiply by a `Transform4D` data structure is convenient when both $\mathbf{H}$ and $\mathbf{H}^{-1}$ are already available so that the matrix $\mathbf{M}^{-1}$ doesn't need to be extracted.

In the general case, both $\mathbf{H}$ and $\mathbf{M}^{-1}$ are needed to transform both positions and normals. If $\mathbf{M}$ happens to be orthogonal, which is often the case, then its inverse is simply equal to its transpose, so the transformed normal is just $\mathbf{n}\mathbf{M}^{\mathrm{T}}$, but this is equivalent to $\mathbf{M}\mathbf{n}$ if we treat $\mathbf{n}$ as a column matrix. Thus, it is common to see game engines treat ordinary vectors and normal vectors as the same kind of mathematical element and use multiplication by the same matrix $\mathbf{H}$ on the left to transform both kinds among different coordinate systems.

Listing 3.1. This multiplication operator multiplies a `Vector3D` data structure representing a normal vector as a row matrix on the right by a `Transform4D` data structure to transform a normal vector from coordinate system *B* to coordinate system *A*. The transformation matrix is treated as a 3×3 matrix, ignoring the fourth column. Note that this transforms a normal vector in the opposite sense in relation to how the same matrix would transform an ordinary vector from coordinate system *A* to coordinate system *B*.

```
Vector3D operator *(const Vector3D& n, const Transform4D& H)
{
   return (Vector3D(n.x * H(0,0) + n.y * H(1,0) + n.z * H(2,0),
                    n.x * H(0,1) + n.y * H(1,1) + n.z * H(2,1),
                    n.x * H(0,2) + n.y * H(1,2) + n.z * H(2,2)));
}
```