Vol. 14, No. 1: 55–62

Efficient Construction of Perpendicular Vectors without Branching

Michael M. Stark
Boise State University

Abstract. This paper presents a novel formula for computing an arbitrary vector perpendicular to a given 3D vector. The formula, as well as the provided C language implementation, are unusual in that they require no conditional branching. The formula involves little arithmetic and is amenable to hardware implementation.

1. Introduction

Finding an arbitrary vector perpendicular to a given (nonzero) 3D vector is a common task in computer graphics. That is, given a vector $\mathbf{v} \neq \mathbf{0}$, what $\mathbf{n} \neq \mathbf{0}$ satisfies $\mathbf{n} \cdot \mathbf{v} = 0$? The problem arises, for example, in the construction of a coordinate system (i.e., an orthonormal basis) from a single axis vector.

While simple enough to state, the vector perpendicular problem is more complicated than it might seem. There is no unique choice for the perpendicular vector $\mathbf{n}$: any nonzero vector in the plane perpendicular to $\mathbf{v}$ suffices, yet no particular choice is readily apparent. As it happens, the construction of a unit perpendicular cannot be a continuous function of $\hat{\mathbf{v}}$. To see why, notice that the problem can be formulated as the development of a (unit) vector-valued function P on the the space of all unit vectors, i.e., the unit sphere, having the property that $\hat{\mathbf{v}} \cdot P(\hat{\mathbf{v}}) = 0$. As such, P is a transformation of the unit sphere. The Brouwer fixed-point theorem [Rotman 98] (in a special case) asserts that any *continuous* transformation of the sphere must have a fixed point. So if P is continuous, there must be some $\hat{\mathbf{u}}_0$ such that $P(\hat{\mathbf{u}}_0) = \hat{\mathbf{u}}_0$ and therefore $\hat{\mathbf{v}} \cdot P(\hat{\mathbf{v}}) = 1 \neq 0$.

1086-7651/09 $0.50 per page

The vector perpendicular problem has an interesting interpretation as "combing" a sphere. Consider a "hairy" sphere, which has a fine hair sticking straight out at each point. To "comb" the sphere is to lay each hair down flat, making it tangent to the sphere and therefore perpendicular to the original orientation. It cannot be done: there will always be a point where the hairs all bunch up toward an outward-pointing "cowlick" or a "whorl" around a single hair sticking straight out. The Brouwer fixed point theorem on the sphere is sometimes called the "hairy ball theorem" and has some other interesting consequences.

The innate discontinuity in finding perpendicular vectors has some practical consequences. First, any method for constructing a vector perpendicular has at least one singular point. In addition, there is no formula for a vector perpendicular in terms of combinations of *continuous* elementary functions. Discontinuities suggest a need for an if/else structure in an implementation, which usually results in conditional branch instructions in generated machine code. Conditional branches on contemporary hardware are comparatively costly due to the trend toward deep pipelining—when a branch instruction is executed the pipeline may have to be stalled to determine the outcome of the branch. Hardware branch prediction mitigates this, but branch prediction is imperfect and costly in terms of power requirements and, to a lesser extent, CPU efficiency. The formula we present can be implemented without conditional branch instructions, which is advantageous by itself, and runs about 30% faster than the conditional formula on which is it based.

2. Perpendicular Vector

One approach for constructing a perpendicular to a given vector $\mathbf{v}$ is to use the cross product of $\mathbf{v}$ with one of the coordinate axis vectors [Shirley and Morley 03]. There are three choices:

$$(x, y, z) \times (1, 0, 0) = (0, z, -y), \tag{1}$$

$$(x, y, z) \times (0, 1, 0) = (-z, 0, x), \tag{2}$$

$$(x, y, z) \times (0, 0, 1) = (y, -x, 0). \tag{3}$$

Of course, if $\mathbf{v}$ happens to line up with the axis vector, the result is the zero vector, so no single choice of axis works for all inputs. A choice of axis thus depends on the input $\mathbf{v}$ (therein lies the discontinuity). Crossing by an axis vector essentially projects the vector onto the principal plane perpendicular to that axis, then takes a natural perpendicular in that plane. It is therefore sensible to project onto the principal plane from which $\mathbf{v}$ deviates the least,

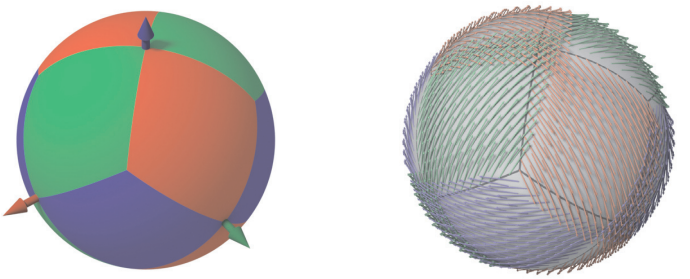

Figure 1. The perpendicular to $\mathbf{v}$ is computed from the cross product with the axis corresponding to the principal plane from which $\mathbf{v}$ deviates the least. This partitions the sphere of directions into 12 regions in the shape of a rhombic dodecahedron (left). As a function on the sphere, the perpendicular is discontinuous (right).

which corresponds to the axis on which $\mathbf{v}$ has the smallest coordinate:

$$\mathbf{n}(x,y,z) = \begin{cases} (\ 0, & z, -y) & \text{if } |x| < |y| \text{ and } |x| < |z|, \\ (-z, & 0, & x) & \text{if } |y| < |x| \text{ and } |y| < |z|, \\ (\ y, -x, & 0) & \text{otherwise.} \end{cases} \tag{4}$$

Figure 1 illustrates the discontinuity structure. Equation (4) is very similar to the formula given by Hughes and Möller [Hughes and Möller 99], which is essentially

$$\mathbf{n}_0(x,y,z) = \begin{cases} (\ 0, -z, & y) & \text{if } |x| < |y| \text{ and } |x| < |z|, \\ (-z, & 0, & x) & \text{if } |y| < |x| \text{ and } |y| < |z|, \\ (-y, & x, & 0) & \text{otherwise.} \end{cases} \tag{5}$$

However, Equation (4) has the advantage that $\mathbf{v}$, the chosen axis vector, and $\mathbf{n}$ form a positively-oriented basis (Figure 2).

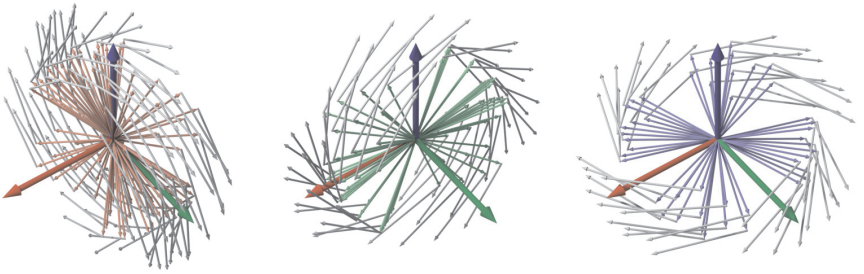

Figure 2. The chosen axis, the vector, and the normal form a positively-oriented basis.

Equations (4) and (5) are well suited to an "if/else" sequence or a related construct. Equation (4) can be written as a simpler expression without the explicit cases:

$$\mathbf{n} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \times \begin{pmatrix} [x \text{ is smallest}] \\ [y \text{ is smallest}] \\ [z \text{ is smallest}] \end{pmatrix} = \begin{pmatrix} y[z \text{ is smallest}] - z[y \text{ is smallest}] \\ z[x \text{ is smallest}] - x[z \text{ is smallest}] \\ x[y \text{ is smallest}] - y[x \text{ is smallest}] \end{pmatrix} \tag{6}$$

where the "is smallest" condition is 1 if the corresponding coordinate is minimal, and 0 otherwise. In terms of the Heaviside step function[1] and absolute value, the condition $[x \text{ is smallest}]$ can be expressed as

$$[x \text{ is smallest}] = u(|y| - |x|)u(|z| - |x|). \tag{7}$$

If the others were expressed analogously, more than one of the "is smallest" conditions could be 1 if any of the coordinates are equal. In particular, if $\mathbf{v}$ is a multiple of $(1, 1, 1)$, all the "is smallest" values would be 1 and Equation (6) would produce the zero vector. The simplest way around this problem is to encode the "is smallest" for y and z to have mutual exclusion:

$$[y \text{ is smallest}] = (1 - [x \text{ is smallest}])u(|z| - |y|) \tag{8}$$

$$[z \text{ is smallest}] = (1 - [x \text{ is smallest}])(1 - [y \text{ is smallest}]). \tag{9}$$

Combining Equations (6)–(9) and substituting to remove data dependencies yields the following explicit formula for a perpendicular vector:

$$\mathbf{n} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \times \begin{pmatrix} u\left(|y| - |x|\right) u\left(|z| - |x|\right) \\ \left(1 - u\left(|y| - |x|\right) u\left(|z| - |x|\right)\right) u\left(|z| - |y|\right) \\ \left(1 - u\left(|y| - |x|\right) u\left(|z| - |x|\right)\right) \left(1 - u\left(|z| - |y|\right)\right) \end{pmatrix} \tag{10}$$

Equation (10) has the innate discontinuities of the perpendicular function pushed into the Heaviside step function. At first glance it might appear that the expression actually involves more "if"s than the piecewise expression of Equation (6); indeed, a direct implementation of Equation (10) using a library function for $u(x)$ does run significantly slower, even when the common values are precomputed. However, $u(x)$ can be easily computed without any conditional branch instructions.

[1]For the purposes of this paper, the Heaviside step function is defined as

$$u(x) \equiv \begin{cases} 0 & \text{if } x < 0; \\ 1 & \text{if } x \geq 0, \end{cases}$$

but the choice of the value at $x = 0$ is not universal. Sometimes, the value at 0 is left undefined, which is a useful model for the signed zero anomaly of `signbit`.

3. Implementation

Floating-point values conforming to the IEEE 754 standard [IEEE Task P754 85] have the sign stored in a dedicated sign bit; the value is negative if the bit is set. The standard *recommends* a "signbit" function to extract this bit; $u(x)$ is 1 minus the signbit of x, except in the case of "negative zero" (the IEEE 754 standard allows zero to have a sign). Negative zero is not distinguished from "positive" zero in floating-point comparisons, but it does present a problem in implementing $u(x)$. Using a sign bit implementation, $u(|x| - |y|)$ can erroneously return 0 when $|x| = |y|$ and the floating-point difference $|x| - |y|$ is computed as "negative" zero (although this can only occur in "round toward $-\infty$ mode, which is not the default [Goldberg 91]).

Fortunately, this $u(0)$ quirk does not affect the validity of Equation (10). The issue only arises when two coordinates of the input vector are equal. For example, if $|x| = |y|$ and $|x| < |z|$, $u(|y| - |x|) = 0$ causes y to be chosen as the "smallest" coordinate. If $u(|y| - |x|)$ instead returns 1 (incorrectly), x will be chosen. The resulting perpendicular is different, but valid nonetheless. The remaining cases are detailed in Table 1.The case where $|x| = |y| = |z|$ is omitted; any axis choice works in this case.

A C/C++ implementation of Equation (10) looks something like this:

```
const unsigned int uyx = SIGNBIT(ABS(x) - ABS(y));
const unsigned int uzx = SIGNBIT(ABS(x) - ABS(z));
const unsigned int uzy = SIGNBIT(ABS(y) - ABS(z));

const unsigned int xm = uyx & uzx;
const unsigned int ym = (1^xm) & uzy;
const unsigned int zm = 1^(xm & ym);

n.x = zm*y - ym*z;
n.y = xm*z - zm*x;
n.z = ym*x - xm*y;
```

condition	incorrect u	correct axis	axis chosen, and why
$\|x\| = \|y\|,\ \|y\| < \|z\|$	$u(\|y\| - \|x\|) \to 0$	x or y	x: $u(\|z\| - \|x\|) = 1$
$\|x\| = \|y\|,\ \|y\| > \|z\|$	$u(\|y\| - \|x\|) \to 0$	z	z: $u(\|z\| - \|x\|) = 0$, $u(\|z\| - \|y\|) = 0$
$\|x\| = \|z\|,\ \|z\| < \|y\|$	$u(\|z\| - \|x\|) \to 0$	x or z	x: $u(\|y\| - \|x\|) = 1$
$\|x\| = \|z\|,\ \|z\| > \|y\|$	$u(\|z\| - \|x\|) \to 0$	y	y: $u(\|y\| - \|x\|) = 0$, $u(\|z\| - \|y\|) = 1$
$\|y\| = \|z\|,\ \|z\| < \|x\|$	$u(\|z\| - \|y\|) \to 0$	y or z	y: $u(\|y\| - \|x\|) = 0$
$\|y\| = \|z\|,\ \|z\| > \|x\|$	$u(\|z\| - \|y\|) \to 0$	x	x: $u(\|y\| - \|x\|) = 1$, $u(\|z\| - \|x\|) = 1$

Table 1. Potentially troublesome cases involving the value of $u(0)$. When computed from the sign bit, $u(0)$ can erroneously return 0. In all cases an appropriate axis ends up getting chosen.

where the ABS macro is set to the appropriate absolute value function, e.g., fabsf if the vector components are of type float; fabs if they are of type double. SIGNBIT is trickier. The C99 standard math library includes a signbit macro, but it returns "nonzero" if the sign bit is set while the code above expects the value to be 1. The sign bit can be extracted directly, by forcing the C compiler to treat a floating-point value as an integer type. The (machine-dependent) macro

```
#define SIGNBIT(X) (((union  float x; unsigned long n; )(X)).n >> 31)
```

works for float X on Intel IA32 architectures under gcc and on any system in which unsigned long is a 32-bit unsigned integer type, the sign bit is the most significant bit in the floating-point representation (the position of the sign bit is not specified by the IEEE 754 standard), and >> is a logical (vs. arithmetic) right bitshift operation. The ABS macro can be implemented similarly by clearing the sign bit directly; e.g., for float type

```
#define ABS(X) (((union  float x; unsigned long n; )(X)).n & 2147483647ul)
```

although fabs and fabsf are typically implemented in a similar fashion and are likely to be more efficient.

3.1. Remarks

The final three lines of the code implement the cross product of Equation (10) and can be replaced with a cross product:

```
n = CROSS(x, y, z, xm, ym, zm).
```

This could be useful if CROSS represented a fast hardware cross product operation.

The last three lines can also be viewed as an indirect implementation of the piecewise-defined formula of Equation (4). Replacing these lines with

```
n.x = -zm*y - ym*z;
n.y = xm*z - zm*x;
n.z = ym*x + xm*y;
```

implements Equation (5), the formula of Hughes and Möller, also without branching, although the computation is no longer a proper cross product.

3.2. Performance

We tested our signbit-based implementation of Equation (10) on several Intel, AMD, and Sun platforms, usually with gcc (version 4.3). The benchmark was simple: an iterative computation of a large set of (precomputed) random vectors. We found that it runs at least 30% faster than a direct implementation of the piecewise formula of Equation (4). The running times are highly dependent on the implementation of the `SIGNBIT` macro, and also on the placement and ordering of the instructions. The compiler version is also significant: under gcc version 3.2, our branchless implementation runs more like 200% faster.

Because exactly one of `xm`, `ym`, and `zm` is unity (the others are zero), the arithmetic at the end of the code can be replaced with bitwise logical operations. The absolute value is normally computed by simply clearing the sign bit, so the only true floating-point operations in the implementation are the three subtractions at the beginning. The remaining operations involve only bitshifting and bitwise logic, which makes it naturally suited to hardware implementation. Note that `SIGNBIT(ABS(x) - ABS(y))` can be replaced with `(ABS(x) - ABS(y) < 0)`, but in our tests this runs significantly slower, probably because it requires a full-blown floating-point comparison, which is more complicated than a subtraction.

A final remark: if a floating-point hardware Heaviside step function instruction were available, the entire implementation could be done in floating-point arithmetic. This could provide a significant performance increase, because the cost of moving the operands between the integer and floating-point units would be eliminated.

Acknowledgments. The author wishes to thank Peter Shirley for posing the problem of a "branchless" perpendicular vector formula, Gang-Ryung Uh for his assistance with compiler optimization and hardware details, and James Arvo for pointing out the interpretation of the problem as "combing a sphere." An earlier version of the result was contained in a submission to the 2006 IEEE Symposium on Interactive Ray Tracing. Although it was not accepted to that venue, the anonymous reviewers provided useful comments and suggested submitting portions of the work to the *journal of graphics tools*.

References

[Goldberg 91] David Goldberg. "What Every Computer Scientist Should Know About Floating-point Arithmetic." *ACM Comput. Surv.* 23:1 (1991), 5–48.

[Hughes and Möller 99] John F. Hughes and Tomas Möller. "Building an Orthonormal Basis from a Unit Vector." *journal of graphics tools* 4:4 (1999), 33–35.

[IEEE Task P754 85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. New York: IEEE, 1985. Reprinted in *SIGPLAN* 22:2, 9–25.

[Rotman 98] Joseph J. Rotman. *An Introduction to Algebraic Topology*. New York: Springer, 1998.

[Shirley and Morley 03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*, Second edition. Natick, MA: A K Peters, Ltd., 2003.

Web Information:

http://jgt.akpeters.com/papers/Stark09/

Michael Stark, Boise State University, Computer Science Department, 1910 University Drive, Boise, ID 83725 (stark@cs.boisestate.edu)

Received February 7, 2009; accepted in revised form April 24, 2009.