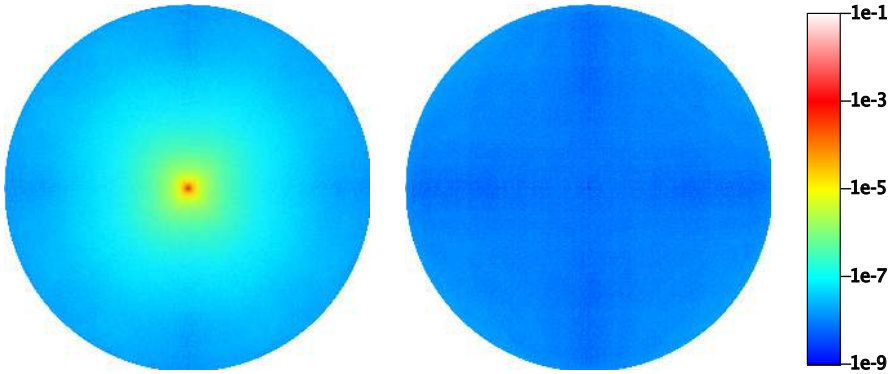
Building an Orthonormal Basis, Revisited

Tom Duff, James Burgess, Per Christensen, Christophe Hery, Andrew Kensler, Max Liani, and Ryusuke Villemin

Pixar



(a) Frisvad's method **(b)** Revised method scale

Figure 1. RMS deviation from orthogonality, for Frisvad's method and our revised method. The two discs have $x^2 + y^2 \leq 1$. At each point of each disc we calculate a unit vector $(x, y, -\sqrt{1 - x^2 - y^2})$, and construct an orthonormal frame, using Frisvad's method [Frisvad 2012b] in (a) and our revised method in (b). Colors are proportional to the RMS deviation of the frame from orthogonality, as in the scale on the right.

Abstract

Frisvad [2012b] describes a widely-used computational method for efficiently augmenting a given single unit vector with two other vectors to produce an orthonormal frame in three dimensions, a useful operation for any physically based renderer. However, the implementation has a precision problem: as the z component of the input vector approaches -1, floating point cancellation causes the frame to lose all precision. This paper introduces a solution to the precision problem and shows how to implement the resulting function in C++ with performance comparable to the original.

 ISSN 2331-7418

1. Introduction

When rendering using Monte Carlo integration, we typically generate orthonormal coordinate frames billions of times per image, so even tiny performance gains can produce significant savings. At the same time, we need to produce results with good precision.

For example, we often need to choose a random sample from a distribution of outgoing ray directions in response to an incoming ray arriving at some surface point. The sampling is often easiest to do in a canonical coordinate frame where the normal vector at the surface point is a fixed direction, for example $(0, 0, 1)$. Transforming the incoming and outgoing directions to and from the canonical coordinate frame requires that we be able to extend a given unit vector n into an orthonormal basis with that vector as one of its axes.

The most obvious way to do that is to select some vector perpendicular to n and normalize it to get the second vector of the basis. Then the third vector is just the cross-product of the first two. Hughes and Möller [1999] offer a particularly efficient way of choosing a vector perpendicular to n. This was the state of the art until Frisvad's method [Frisvad 2012b] appeared, improving on the efficiency of Hughes and Möller by a factor of 2 or more, using a computation expressed in terms of quaternions that requires no normalization and so no expensive square roots. Frisvad's code is reproduced in Listing 1.

```
1  #include <Vec3f.h>
2
3  void frisvadONB(const Vec3f& n, Vec3f& b1, Vec3f& b2)
4  {
5      if(n.z < -0.9999999f) // Handle the singularity
6      {
7          b1 = Vec3f( 0.0f, -1.0f, 0.0f);
8          b2 = Vec3f(-1.0f,  0.0f, 0.0f);
9          return;
10     }
11     const float a = 1.0f / (1.0f + n.z);
12     const float b = -n.x*n.y*a;
13     b1 = Vec3f(1.0f - n.x*n.x*a, b, -n.x);
14     b2 = Vec3f(b, 1.0f - n.y*n.y*a, -n.y);
15 }
```

Listing 1. Frisvad's orthonormal basis code

2. The Problem

When we evaluated Frisvad's code, we liked its speed, but we discovered that it occasionally failed, producing coordinate frames that are not even remotely orthogonal.

Frisvad's paper claims extreme accuracy – the RMS deviation from orthonormal of its results is given as 4.8×10^{-10}. But, if you run Frisvad's code on the vector

$$(0.00038527316, 0.00038460016, -0.99999988079),$$

the resulting frame matrix is

$$\begin{bmatrix} \text{-0.24516642094} & \text{-1.24299144745} & \text{-0.00038527316} \\ \text{-1.24299144745} & \text{-0.24082016945} & \text{-0.00038460016} \\ \text{0.00038527316} & \text{0.00038460016} & \text{-0.99999988079} \end{bmatrix} \quad (1)$$

The product of this and its transpose, which should be the identity, is not even close:

$$\begin{bmatrix} \text{1.60513436794} & \text{0.60407733917} & \text{-0.00018723766} \\ \text{0.60407733917} & \text{1.60302221775} & \text{-0.00018691065} \\ \text{-0.00018723766} & \text{-0.00018691065} & \text{1.00000000000} \end{bmatrix}$$

The RMS deviation from orthogonality is 0.29.[1]

A possibly worse case can be seen by running Frisvad's code on

$$(-0.00019813581, -0.00008946839, -0.99999988079)$$

In this case, the resulting frame is

$$\begin{bmatrix} \text{0.67068171501} & \text{-0.14870394766} & \text{0.00019813581} \\ \text{-0.14870394766} & \text{0.93285262585} & \text{0.00008946839} \\ \text{-0.00019813581} & \text{-0.00008946839} & \text{-0.99999988079} \end{bmatrix}$$

Again, the RMS error is large (0.16), but the determinant of the frame matrix is negative – the coordinate frame has the wrong handedness, rendering it useless for many purposes.

These large errors suggest two questions. First, how can the deviation be so large, even occasionally, if the average error is less than 5×10^{-10}? Even if only one in a

[1] We calculate deviations from orthogonality the same way that Frisvad does. If n, u and v are the three vectors of an orthonormal basis, $|n|$, $|u|$ and $|v|$ should all be 1, and $n\cdot u$, $n\cdot v$ and $u\cdot v$ should all be zero, so the average squared error in a calculated basis is

$$\frac{(|n|-1)^2+(|u|-1)^2+(|v|-1)^2+(n\cdot u)^2+(n\cdot v)^2+(u\cdot v)^2}{6}$$

and the RMS error is the square root of the average of this over a set of test of bases.

million deviations were as large as 0.29 and the other 999,999 were zero, the average error would be 2.9×10^{-7}, nearly three orders of magnitude larger than Frisvad reports. And second, what is causing these errors?

The first question is easily answered. Frisvad [2012a] is an archive containing the program used to compute the error values reported in Frisvad [2012b], and it has a bug.[2]

The total error over a sequence of test runs is divided by 6 at each step of the test loop, but the division should just apply to the error of the test run being accumulated or, equivalently, be moved outside the loop. That is, at step `i` of the loop, the code computes `sum = (sum + err[i])/6` instead of `sum = sum + (err[i]/6)`.

After correction, the program calculates an average RMS error of 6.7×10^{-7}, which sounds much more plausible. Even with this correction, Frisvad's numbers differ substantially from what we report below in Table 1, mostly because our averages include many more samples (one billion vs. ten thousand) and the averages depend strongly on extremely poor performance in rare cases that Frisvad's tests may not sample well.

But how did we get a result as bad as that shown in Equation (1)? Look at line 11 of Listing 1. When $n.z$ is close to -1, the subtraction in the denominator can suffer catastrophic cancellation. Figure 2a shows $1/(1+n.z)$ calculated in single precision (the red line) and double precision (blue) for values of $n.z$ varying from $-1+(3\times10^{-8})$ to $-1+(5\times10^{-7})$. The stairsteps in the single precision plot mark the points at which we move from one representable number to the next. Figure 2b shows the relative error — the double-precision plot of Figure 2a divided by the single-precision plot. You can see that at the left end, where $n.z$ is closest to -1, the single-precision value of $1/(1+n.z)$ can be wrong by up to a factor of 2. That is to say, even its most-significant bit can be wrong, rendering the rest of the calculation meaningless.

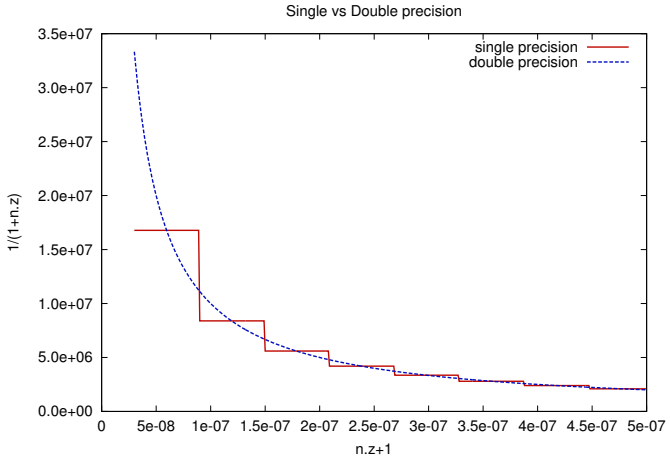
Recent work by Nelson Max [2017], independent of ours, goes into a much more detailed analysis of these errors.

3. The Solution

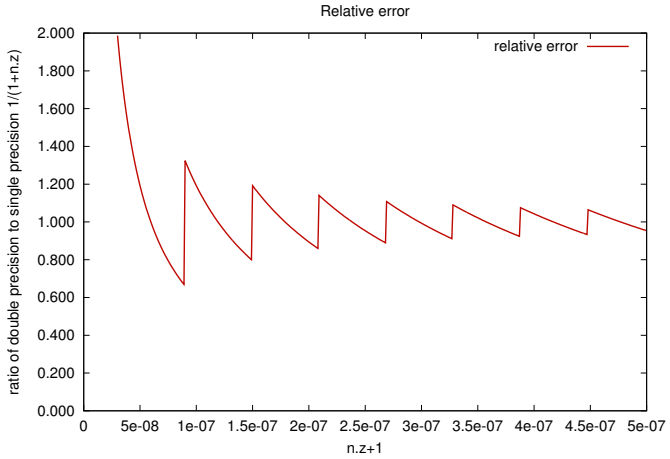
What are we to do about this? The code in Listing 1 is actually pretty well-behaved when $n.z$ is not near -1. This suggests rewriting it so that if $n.z < 0$ we compute a frame for $-n$ and negate the result, as in Listing 2. (In fact, we negate `b2` but not `b1` in the first branch of the `if` so that the handedness of the frames it produces matches the second branch.) With this change, the RMS error is reduced to 2.13×10^{-8} and the worst error we see, when testing a billion random inputs, is 1.04×10^{-7}.

Figure 1 illustrates the error magnitude of Frisvad's method and our revision. The

[2] Since this was written, Frisvad's online archive has been updated, fixing the error.



(a) *Quantization near* $n.z = -1$



(b) *Relative error near* $n.z = -1$

Figure 2. Floating point error near $n.z = -1$. The red line in (a) is the single precision error in the calculation of $1/(1 + n.z)$, the blue line is double precision. The stairsteps occur as $n.z$ moves from one representable single precision number to the next. (b) shows the ratio of those two curves. The deviation can be as much as a factor of 2, indicating a single precision result not accurate to even one bit of precision.

two discs represent $n.x$ and $n.y$ values with $(n.x)^2 + (n.y)^2 \leq 1$, with $n.z$ chosen to be negative (that is $n.z = -\sqrt{1-(n.x)^2-(n.y)^2}$). Colors in each plot are a rainbow proportional to the log of the RMS error: blue when the error is 10^{-9} or less, interpolating to green when the error is 10^{-7}, yellow at 10^{-5}, red at 10^{-3} and white when the error is 10^{-1} or more.

The principal problem with the version in Listing 2 is that it runs about twice as slow as Listing 1. That seems surprising, since the computations are very similar.

```
void revisedONB(const Vec3f &n, Vec3f &b1, Vec3f &b2)
{
    if(n.z<0.){
        const float a = 1.0f / (1.0f - n.z);
        const float b = n.x * n.y * a;
        b1 = Vec3f(1.0f - n.x * n.x * a, -b, n.x);
        b2 = Vec3f(b, n.y * n.y*a - 1.0f, -n.y);
    }
    else{
        const float a = 1.0f / (1.0f + n.z);
        const float b = -n.x * n.y * a;
        b1 = Vec3f(1.0f - n.x * n.x * a, b, -n.x);
        b2 = Vec3f(b, 1.0f - n.y * n.y * a, -n.y);
    }
}
```

Listing 2. The code of Listing 1, revised to avoid catastrophic cancellation.

```
void branchlessONB(const Vec3f &n, Vec3f &b1, Vec3f &b2)
{
    float sign = copysignf(1.0f, n.z);
    const float a = -1.0f / (sign + n.z);
    const float b = n.x * n.y * a;
    b1 = Vec3f(1.0f + sign * n.x * n.x * a, sign * b, -sign * n.x);
    b2 = Vec3f(b, sign + n.y * n.y * a, -n.y);
}
```

Listing 3. A branchless version of Listing 2, using `copysignf` to eliminate the test.

The main difference is that while the test in Listing 1 almost always takes the false branch, and so will usually be correctly predicted, the test in Listing 2 takes each branch roughly 50% of the time at random, and so will often be mispredicted. We can eliminate the test by first calculating `sign = copysignf(1.0f, n.z)`, for which all our compilers generate code that transfers the sign bit without branching, and using multiplies by `sign` to merge the two branches of Listing 2, as shown in Listing 3. This regains almost all of the time lost by the Listing 2 version, running between 5% and 12% slower than Frisvad's original, depending on which compiler we use. The branchless version is also amenable to vectorization.

The C99 and C++11 standards include `copysignf`, but compiler support often lags the standards by many years. Other alternatives, such as explicitly writing

```
sign = n.z>=0.0f ? 1:0f : -1.0f,
```

can work just as well – if the whole function is inlined, the compiler can move the test up far enough to avoid pipeline stalls.

routine	icc time	clang time	g++ time	RMS error	largest error
Hughes-Möller	18.59 ns	22.92 ns	22.87 ns	3.00×10^{-8}	1.15×10^{-7}
Frisvad	8.38 ns	8.28 ns	8.53 ns	3.91×10^{-5}	2.91×10^{-1}
Revised	15.67 ns	15.05 ns	15.31 ns	2.13×10^{-8}	1.04×10^{-7}
With copysign	9.35 ns	9.62 ns	10.02 ns	2.13×10^{-8}	1.04×10^{-7}

Table 1. Speed and accuracy

4. Performance

Table 1 summarizes our timing and accuracy data for the various routines discussed in this paper. All statistics are based on runs using one billion random unit vectors. All timings are for a single core on one of our desktop machines, a 17 core Intel Xeon E5-2699 (2.30 GHz) virtual machine with 128 GiB of memory, running Red Hat Enterprise Linux 7.2. We give times for the three compilers available to us (icc 15.0.3, clang++ 3.4.1, and g++ 4.8.5), all using the `-O3` optimization level. Errors were the same for all compilers, so we only report them once.

The RMS error of our revised method is three orders of magnitude better than Frisvad's, at a cost of 17% or less in speed. The maximum error, in our tests on a large random sample of inputs, is almost six orders of magnitude better. Our accuracy is even slightly better than the Hughes-Möller method, the standard algorithm before the appearance of Frisvad's paper.

Acknowledgements

Conversations with Marc Bannister and Philippe Leprince added important details to our understanding.

References

FRISVAD, J. R., 2012. onb (code for building an orthonormal basis from a 3D unit vector without normalization). URL: http://people.compute.dtu.dk/jerf/code/. 4

FRISVAD, J. R. 2012. Building an orthonormal basis from a 3D unit vector without normalization. *J. Graphics Tools 16*, 3, 151–159. URL: http://orbit.dtu.dk/files/126824972/onb_frisvad_jgt2012_v2.pdf. 1, 2, 4

HUGHES, J. F., AND MÖLLER, T. 1999. Building an orthonormal basis from a unit vector. *J. Graph. Tools 4*, 4, 33–35. URL: http://dx.doi.org/10.1080/10867651.1999.10487513. 2

MAX, N. 2017. Improved accuracy when building an orthonormal basis. *Journal of Computer Graphics Techniques (JCGT) 6*, 1 (March), 60–66. URL: http://jcgt.org/published/0006/01/02/. 4

Author Contact Information

Tom Duff	td@pixar.com	James Burgess	jrb@pixar.com
Per Christensen	per@pixar.com	Christophe Hery	chery@pixar.com
Andrew Kensler	aek@pixar.com	Max Liani	maxl@pixar.com
Ryusuke Villemin	rvillemin@pixar.com		

Pixar
1200 Park Avenue
Emeryville, CA 94608

Tom Duff, James Burgess, Per Christensen, Christophe Hery, Andrew Kensler, Max Liani, and Ryusuke Villemin, Building an Orthonormal Basis, Revisited, *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 1, 1–8, 2017
http://jcgt.org/published/0006/01/01/

Received:	2016-11-01		
Recommended:	2016-12-06	Corresponding Editor:	Peter Shirley
Published:	2017-03-27	Editor-in-Chief:	Marc Olano