Improved accuracy when building an orthonormal basis

Nelson Max
University of California, Davis

Abstract

Frisvad's method for building a 3D orthonormal basis from a unit vector has accuracy problems in its published floating point form. These problems are investigated and a partial fix is suggested, by replacing the threshold 0.9999999 by the threshold -0.999805696, which decreases the maximum error in the orthonormality conditions from 0.623 to 0.0062.

1. Introduction

When shading a volume with an SGGX microflake distribution for another paper [Max et al. 2017], in an implementation of Heitz et al. [2015], I used the method of Frisvad [2012] it recommended to extend a unit surface normal n to an orthonormal basis. This is needed for importance sampling the BRDF of a microflake. The method of Frisvad constructs two perpendicular unit vectors b_1 and b_2 which are perpendicular to n, by explicit formulas in the components of n (see the last two lines of the code in Listing 1), and proves that these three vectors form an orthonormal basis.

However, when using these formulas, I noticed glitches in the shading, which I eventually traced back to inaccuracies in the vectors b_1 and b_2. I analyze these inaccuracies in the rest of this paper, in terms of the orthogonality errors $|n \cdot b_1|$, $|n \cdot b_2|$, and $|b_1 \cdot b_2|$, and the normality errors $|\|b_1\| - 1|$ and $|\|b_2\| - 1|$. (The input vector n is not modified, and is assumed to be correctly normalized.) A simultaneously submitted paper by Duff et al. [2017] discusses the same problems with Frisvad's algorithm, and presents a more accurate and efficient correction than those discussed below. An earlier paper by Hughes and Möller [1999] presents a method of extending a unit vector n to an orthonormal basis using cross products. It involves an *if* test, which can be inefficient in a SIMD machine such as a GPU, and Stark [2009] gives a related method without branching. As functions of the input unit vector n, the vector fields b_1 and b_2 from these other three methods all have discontinuities along fairly large curves on the unit sphere. A potential advantage of the Frisvad method is that

 ISSN 2331-7418

```
void frisvad ( const Vec3f & n, Vec3f & b1 , Vec3f & b2)
{
    if(n.z < -0.9999999f) // Handle the singularity
    {
        b1 = Vec3f ( 0.0f, -1.0f, 0.0f);
        b2 = Vec3f ( -1.0f, 0.0f, 0.0f);
        return ;
    }
    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 1. The algorithm from Frisvad [2012], reprinted with permission.

the resulting fields of tangent vectors on the unit sphere have discontinuities only on a tiny circle near the south pole (see Frisvad [2014]).

2. Accuracy investigation

Once I discovered that for some values of n, the computed b_1 and b_2 did not create an orthonormal basis with n, I looked for the maximum deviation from orthonormality. I made a grid of input vectors n on the unit sphere by regularly sampling spherical coordinate values (θ, ϕ), for θ in a range near π radians, to get values spanning the threshold 0.9999999 for the z component $n.z$ of n, and ϕ in the range from 0 to $\pi/4$, which is sufficient due to the symmetry of the formulas for b_1 and b_2. I computed the coordinates of n accurately as doubles, and then converted them to floats to define n as a float vector of type Vec3f. As might be expected, I found the greatest deviation from orthonormality when the z component $n.z$ of n was exactly 0.9999999. Considering only the three dot products defining the orthogonality condition, the greatest inaccuracy was at $\phi = \pi/4$, where n = (0.0003860202, 0.0003860202, -0.9999998808). Note that the z coordinate $n.z$ = -0.9999998808 is the representable 32 bit floating point number closest to the real number threshold 0.9999999 in Listing 1. For this input, the algorithm in Listing 1 using floats gives output b_1 = (-0.2499998808, -1.249999881, -0.0003860202) and b_2 = (-1.249999881, -0.2499998808, -0.0003860202), and the dot product $b_1 \cdot b_2$ = 0.624999762, while it should be zero. Note also that the lengths $|b_1| = |b_2|$ = 1.27475488186 instead of 1. Considering only the normality (unit vector) condition, the worst inaccuracy was for ϕ = 0, and n = (0, 0.000545915, -0.9999998808), with b_1 = (1, 0, 0) and b_2 = (0, -1.499999523, -0.000545915), and the length $|b_2|$ = 1.49999964. These errors are much larger than the RMS errors reported in Frisvad [2012], so they must occur infrequently. But the visual processing in the human eye and brain has spatial and

temporal contrast enhancement steps that amplify small glitches in otherwise smooth shading, especially those that flash in animation, so even infrequent errors can be a problem for computer graphics. In fact, the RMS errors reported in Frisvad [2012] are much less than the 7.22 decimal digit precision limit of 32 bit IEEE floats, which is impossible. One bug in Frisvad's computation of the RMS errors from the errors of the individual executions is discussed in Duff et al. [2017], and another problem in the random distribution of the input vectors is discussed in section 3 below.

To determine the source of these inaccuracies, I implemented the algorithm in Listing 1, with doubles as well as floats for all vectors and variables, and compared the variables a and b, and the final results b_1 and b_2. For the input to the implementation with doubles, I used the double version of the vector n as initially computed from θ and ϕ, before conversion to a float. I discovered that the discrepancy propagated from an error in the intermediate variable $a = 1/(1 + n.z)$, which was $6.71089 \cdot 10^6$ using doubles, but was $8.38861 \cdot 10^6$ using floats. The error in the float version of a came from loss of significance, since $n.z$ is very close to -1, so almost all precision is lost when 1 is added to it.

The proof in appendix B of Frisvad [2012] that $\{n, b_1, b_2\}$ is an orthonormal set of vectors relies on the fact that n is a unit vector, so that $1 - n.x^2 - n.y^2 - n.z^2 = 0$, but when the real-valued components of n are represented as floats, this is not exactly true. For example, in the $\phi = \pi/4$ case above, $1 - n.x^2 - n.y^2 - n.z^2$ = - $5.960462 \cdot 10^{-8}$, which is about the error that could be expected from the precision of IEEE floats. When this quantity is multiplied by $a = 1/(1 + n.z)$ which is of order 10^7, or by $1/(1+n.z)^2$, as it is in this proof, and then assumed to be zero, a significant error can arise.

3. Possible fixes

The choice of the threshold t, set to -0.9999999 in Listing 1, involves a trade off between two kinds of errors, the floating point inaccuracy discussed above when $n.z \geq t$, and the errors in $n \cdot b_1$ and/or $n \cdot b_2$ when $n.z < t$. The size of these latter errors can be significant, even when $n.z$ is very close to -1. Suppose $n.z = -1 + e < t$, for a small positive quantity e, and that ϕ = 0, so that $n.y$ = 0. Then, from the formula for b_2 in Listing 1,

$$n \cdot b_2 = -n.x = -\sqrt{1 - n.z^2} = -\sqrt{1 - (-1 + e)^2} = -\sqrt{2e - e^2} \qquad (1)$$

which is of order $\sqrt{e}$ and can therefore be significant even when e is small.

To explore this trade off, I looked at the maximum errors in the orthogonality or normality conditions for a grid of vectors on the unit sphere spanning the circle $n.z = t$, as in section 2 above, for each of the representable floating point values for thresholds $t \geq -1$ in a range near -1, and then minimized this maximum error

over the range of t values. As t increases away from -1, the maximum errors for vectors n with $n.z \geq t$ will decrease, and those errors for $n.z < t$ will increase, so there should be an optimum trade-off value of t where they are equal. When considering only the normality conditions, the maximum normality error was minimized at 0.00222844 when t = -0.9999805689, in which case the maximum orthogonality error was 0.00622611. When considering only the orthogonality conditions, the maximum error was minimized to the value 0.004901 when t = -0.99998796, in which case the maximum normality condition error was 0.00180161. This same value of t minimized the maximum errors when both the orthogonality and normality conditions were considered together. So if these errors of about a half of one percent are acceptable, this threshold of -0.99998796 would work in Frisvad's method.

If more accuracy is required, one could use the double precision version of Listing 1. With the threshold -0.9999998808 used implicitly there, the maximum error in the dot products is then 0.0004879, achieved, as could be expected, when z is just less than the threshold. The maximum error in the lengths is when z is greater than the threshold, and is $2.3500435 \cdot 10^{-10}$. With double precision, however, the loss of precision discussed above when $n.z$ is greater than but close to the threshold t is less serious, so t can be set closer to -1 to reduce the errors when $z < t$. The optimal threshold is t = -0.9999999999776 where the maximum error in the dot products is $6.82204 \cdot 10^{-6}$ and the maximum error in the lengths is $2.4121 \cdot 10^{-10}$. Computing with doubles after conversion from floats takes more time, mainly because to get the full benefit of the higher floating point accuracy, the double precision vector must be normalized, which requires a square root. Table 1 with timing and accuracy data includes rows for both double precision vector inputs, and for floating point vector inputs converted to doubles and then normalized.

According to an e-mail conversation with Jeppe Frisvad, the RMS errors reported in Frisvad [2012] are smaller than those in Table 1 not only because of (a) the bug discussed in Duff et al. [2017], but also because (b) the random numbers were from rand()/RAND_MAX in stdlib.h. On Frisvad's system RAND_MAX was 32767, so that the only unit vectors generated near the threshold circle at $n.z$ = -0.9999998808, where the error should be largest, were exactly at (0, 0, -1), where the error was zero. Frisvad has a link to his revised code which corrects problem (a) but not problem (b) at the web page `http://www2.compute.dtu.dk/pubdb/p.php?6303`, so it still produces RMS error estimates which are too low. In my tests, I used drand48() in Linux, which can generate $2^{48} - 1$ different random numbers, and thus gives a more uniform sampling on the unit sphere. The maximum errors in Table 1 are less than those reported in text of this section, because the tests giving the error values in the text chose sample input vectors very close the threshold circle, coming closer than the randomly generated vectors did. Note that the tests reported in Table 1, and those used to determine the thresholds, all used normal vectors computed from sampling

Method	precision	executions/sec.	RMS error	max error
Frisvad	float	98,678,205	$4.1761 \cdot 10^{-5}$	0.52189
revised Frisvad	float	98,800,581	$5.1782 \cdot 10^{-6}$	0.004895
revised Frisvad	double	91,729,015	$5.5940 \cdot 10^{-12}$	$3.95365 \cdot 10^{-7}$
revised Frisvad	converted	35,632,021	$2.1628 \cdot 10^{-8}$	$3.95365 \cdot 10^{-7}$
reviewer7	mixed	77,453,148	$1.0989 \cdot 10^{-7}$	$4.17233 \cdot 10^{-7}$
reviewer9	mixed	90,495,879	$1.3668 \cdot 10^{-7}$	$8.34465 \cdot 10^{-7}$
Hughes and Möller	float	60,597,533	$2.6174 \cdot 10^{-8}$	$2.3842 \cdot 10^{-7}$
Hughes and Möller	double	67,712,211	$4.9992 \cdot 10^{-17}$	$4.4409 \cdot 10^{-16}$
Stark	float	41,437,081	$2.6174 \cdot 10^{-8}$	$2.3842 \cdot 10^{-7}$

Table 1. Average execution speed and RMS orthonormality error. The three methods labeled revised Frisvad use the two optimized thresholds described here, with the "converted" in the precision column meaning float vectors converted to doubles and normalized, while the "double" entry means the initial input was already a double precision unit vector. The method called reviewer7 was that in Listing 2, with `rthreshold = -0.7`, while reviewer9 used `rthreshold = -0.9`. Speed timing was done on one core processor of a 3.4GHz Intel i7-6700 CPU. Error results were found using one billion (10^9) random vectors on the unit sphere.

θ and ϕ to generate points on the unit sphere. Because of the limited precision of floating point arithemetic, there may be other vectors n with $\|n\| = 1$ up to floating point precision that cannot be generated in this way.

One reviewer of this submission suggested a combination of the single and the double precision versions, with single precision used when z is far from -1, and double precision used when z is closer. In the code for this in Listing 2, `rthreshold` is the threshold for switching to double precision. There is a speed/accuracy trade off, since making `rthreshold` more negative (closer to -1) will decrease the accuracy, but increase the average speed, because fewer inputs are converted to doubles and normalized. Two possible values of `rthreshold` sampling this trade off are shown in Table 1. Note that in Listing 2, the *if* and *else* clauses from Listing 1 have been switched, making the execution slightly faster, probably because when the more frequent option is in the *if* clause, less time is spent reloading the instruction pipeline. The timings in the first six rows of Table 1 are based on this clause reordering.

```
void buildOrthonormalBasis(const Vec3f & n, Vec3f & b1 , Vec3f & b2)
{
   double dthreshold = -0.9999999999776;
   float rthreshold = -0.7f;
   if (omega_3.z >= rthreshold)
   {
      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 );
   }
   else
   {
      double x = (double) n.x;
      double y = (double) n.y;
      double z = (double) n.z;
      const double d = 1./sqrt(x*x + y*y + z*z);
      x *= d;
      y *= d;
      z *= d;
      if(z >= dthreshold)
      {
        const double a = 1.0 /(1.0 + z );
        const double b = -x*y*a ;
        b1 = Vec3f (1.0f - (float)(x*x*a) , (float)b , (float)(-x) );
        b2 = Vec3f ((float)b , 1.0f - (float)(y*y*a) , (float)(-y) );
      }
      else
      {
        b1 = Vec3f ( 0.0f , -1.0f , 0.0f );
        b2 = Vec3f ( -1.0f , 0.0f , 0.0f );
      }
   }
}
```

Listing 2. The combination algorithm, suggested by a reviewer.

References

DUFF, T., BURGESS, J., CHRISTENSEN, P., HERY, C., KENSLER, A., LIANI, M., AND VILLEMIN, R. 2017. Building an orthonormal basis, revisited. *Journal of Computer Graphics Techniques (JCGT) 6*, 1 (March), 52–59. URL: http://jcgt.org/published/0006/01/02/. 9, 11, 12

FRISVAD, J. R. 2012. Building an orthonormal basis from a 3D unit vector without normalization. *Journal of Graphics Tools 16*, 3, 151 – 159. URL: http://dx.doi.org/10.1080/2165347X.2012.689606. 9, 10, 11, 12

FRISVAD, J. R. 2014. Combing a hairy ball. URL: http://people.compute.dtu.dk/jerf/code/hairy/. 10

HEITZ, E., DUPUY, J., CRASSIN, C., AND DACHSBACHER, C. 2015. The SGGX microflake distribution (see supplemental files). *ACM Transactions on Graphics 34*, 4. URL: http://dl.acm.org/citation.cfm?doid=2809654.2766988. 9

HUGHES, J. F., AND MÖLLER, T. 1999. Building an orthonormal basis from a unit vector. *Journal of Graphics Tools 4*, 4, 33 – 35. doi:10.1080/10867651.1999.10487513. 9

MAX, N., DUFF, T., MILDENHALL, B., AND YAN, Y. 2017. Approximations for the distribution of microflake normals. *The Visual Computer*. URL: http://dx.doi.org/10.1007/s00371-017-1352-2. 9

STARK, M. M. 2009. Efficient construction of perpendicular vectors without branching. *Journal of Graphics, GPU, and Game Tools 14*, 4, 55 – 61. URL: http://dx.doi.org/10.1080/2151237X.2009.10129274. 9

Author Contact Information

Nelson Max
University of California, Davis
1 Shields Ave.
Davis, CA 95616
max@cs.ucdavis.edu
http://faculty.engineering.ucdavis.edu/max/

Nelson Max, Improved accuracy when building an orthonormal basis, *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 1, 9–16, 2017
http://jcgt.org/published/0006/01/02/

Received:	2016-10-24		
Recommended:	2016-12-12	Corresponding Editor:	Matt Pharr
Published:	2017-03-27	Editor-in-Chief:	Marc Olano