Building an Orthonormal Basis from a 3D Unit Vector Without Normalization

Frisvad, Jeppe Revall

Published in:
Journal of Graphics Tools

Link to article, DOI:
10.1080/2165347X.2012.689606

Publication date:
2012

Document Version
Peer reviewed version

Vol. 16, No. 3: 151–159

Building an Orthonormal Basis from a 3D Unit Vector Without Normalization

Jeppe Revall Frisvad
Technical University of Denmark

Abstract. I present two tools that save the computation of a dot product and a reciprocal square root in operations that are used frequently in the core of many rendering programs. The first tool is a formula for rotating a direction sampled around the z-axis to a direction sampled around an arbitrary unit vector. This is useful in Monte Carlo rendering techniques, such as path tracing, where directions are usually sampled in spherical coordinates and then transformed to a Cartesian unit vector in a local coordinate system where the zenith direction is the z-axis. The second tool is a more general result extracted from the first formula, namely a faster way of building an orthonormal basis from a 3D unit vector. These tools require fewer arithmetic operations than other methods I am aware of, and a performance test of the more general tool confirms that it is faster.

1. Introduction

It often happens in Monte Carlo simulation that we would like to sample a direction from a probability density function (pdf) that depends on the angle with a specific direction. The specific direction could be a surface normal or the forward direction of a ray traveling through a scattering material [Pharr and Humphreys 10]. The classical example is the sampling of a direction on a cosine-weighted hemisphere. This is useful for sampling the cosine dependency of diffuse reflections if we would like to evaluate the rendering

ISSN: 2165-347X print/2165-3488 online

equation using Monte Carlo integration. Directions are typically sampled in spherical coordinates, and these are transformed to Cartesian coordinates by assuming that the zenith direction is the z-axis. This means that we get local Cartesian coordinates that must be transformed to coordinates in the usual basis where the zenith direction is an arbitrary (user-specified) direction vector. This paper provides an inexpensive formula (Equation 3) for rotating a sampled direction from being sampled around the z-axis to being sampled around an arbitrary direction.

The rotation used for this type of sampling corresponds to a change of basis. Another way to do it is to take the arbitrary direction (the surface normal, for example), build an orthonormal basis from it, and use the three basis vectors to specify a rotation matrix. This means that the solution found for rotating directions sampled in spherical coordinates also solves a more general problem, namely the problem of building an orthonormal basis from an arbitrary 3D unit vector. My solution (Equations 4–6) has the advantage that it does not require the dot product and reciprocal square root computations needed for vector normalization. The number of other arithmetic operations corresponds to little more than what is needed for a cross product, so the normalization is saved nearly without adding extra cost, as compared to other methods.

2. Notation and Background

Let us use bold face to denote arbitrary vectors (e.g. $\boldsymbol{s}$) and arrow overline to denote unit vectors (e.g. $\vec{\omega}$). With respect to the problem at hand, we let $\vec{n}$ denote the specific direction that we would like to sample a new direction $\vec{\omega}$ around. The new direction is sampled in spherical coordinates (θ, ϕ), where θ is the inclination angle and ϕ is the azimuthal angle. We can translate these spherical coordinates to a Cartesian unit vector by

$$_{\perp}\vec{\omega} = (x, y, z) = (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \ ,$$

where the subscript $_{\perp}$ signals that these are coordinates with respect to a basis $\{\vec{b}_1, \vec{b}_2, \vec{n}\}$, where $\vec{n}$ is the direction of the z-axis. We have no explicit knowledge about the other two vectors of this basis. Therefore, we must build an orthonormal basis from $\vec{n}$ in order to find $\vec{\omega}$ in the usual basis. Once the two other basis vectors have been chosen, the change of basis is

$$\vec{\omega} = x\vec{b}_1 + y\vec{b}_2 + z\vec{n} \ .$$

There are several ways to build the vectors $\vec{b}_1$ and $\vec{b}_2$ from $\vec{n}$. For the basis to be orthonormal, the requirement is that all three vectors are orthogonal

Listing 1. Common ways of finding an orthonormal basis from a unit 3D vector.

```
void naive(const Vec3f& n, Vec3f& b1, Vec3f& b2)
{
  // If n is near the x-axis, use the y-axis. Otherwise use the x-axis.
  if(n.x > 0.9f) b1 = Vec3f(0.0f, 1.0f, 0.0f);
  else           b1 = Vec3f(1.0f, 0.0f, 0.0f);
  b1 -= n*dot(b1, n);              // Make b1 orthogonal to n
  b1 *= rsqrt(dot(b1, b1));        // Normalize b1
  b2 = cross(n, b1);               // Construct b2 using a cross product
}

void hughes_moeller(const Vec3f& n, Vec3f& b1, Vec3f& b2)
{
  // Choose a vector orthogonal to n as the direction of b2.
  if(fabs(n.x) > fabs(n.z)) b2 = Vec3f(-n.y, n.x, 0.0f);
  else                      b2 = Vec3f(0.0f, -n.z, n.y);
  b2 *= rsqrt(dot(b2, b2));        // Normalize b2
  b1 = cross(b2, n);               // Construct b1 using a cross product
}
```

and of unit length. The usual approach is to find a vector orthogonal to $\vec{n}$, normalize it, and take the cross product of this vector and $\vec{n}$ to find the third vector in the basis. C++ code for doing this both in a naïve way and using the faster Hughes-Möller method [Hughes and Möller 99] is provided in Listing 1. The vector normalization is expensive in these methods as it involves a dot product and a reciprocal square root. All the methods I have been able to find use vector normalization. In the following section, I use quaternions [Hamilton 44] to find a method that does not need normalization.

3. Rotation of the z-axis to an Arbitrary Direction

A change of basis corresponds to rotation. A quaternion is a different way to express a three-dimensional rotation. The rotation we need is the rotation from the z-axis to the arbitrary unit vector $\vec{n}$. This is the rotation that we would like to apply to the sampled direction ${}_{\perp}\vec{\omega}$ in order to get $\vec{\omega}$. The following formula finds a unit quaternion $\hat{\boldsymbol{q}}$ that specifies the rotation from a vector $\boldsymbol{s}$ to a vector $\boldsymbol{t}$ [Akenine-Möller et al. 08, Section 4.3]:

$$\hat{\boldsymbol{q}} = (\boldsymbol{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+\boldsymbol{s}\cdot\boldsymbol{t})}} (\boldsymbol{s}\times\boldsymbol{t}), \frac{\sqrt{2(1+\boldsymbol{s}\cdot\boldsymbol{t})}}{2} \right) . \tag{1}$$

If a point or a vector is given in homogeneous coordinates $(\boldsymbol{p}, p_w)$, we can represent it by a quaternion $\hat{\boldsymbol{p}}$ defined by the same four coordinates. The rotation defined by a unit quaternion $\hat{\boldsymbol{q}}$ is then applied using the mapping

$$\hat{\boldsymbol{p}} \mapsto \hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^{-1} ,$$

and $\hat{\boldsymbol{q}}^{-1} = \hat{\boldsymbol{q}}^* = (-\boldsymbol{q}_v, q_w)$ for unit quaternions, where the asterisk (*) denotes the quaternion conjugate. Multiplication of quaternions is associative but not commutative, and it is defined by

$$\hat{\boldsymbol{p}}\hat{\boldsymbol{q}} = (\boldsymbol{p}_v \times \boldsymbol{q}_v + q_w \boldsymbol{p}_v + p_w \boldsymbol{q}_v, \quad p_w q_w - \boldsymbol{p}_v \cdot \boldsymbol{q}_v) \ .$$

To rotate from the direction of the z-axis $(0, 0, 1)$ to $\vec{n} = (n_x, n_y, n_z)$, the rotation formula (1) simplifies to

$$\hat{\boldsymbol{q}} = \left(\frac{(-n_y, n_x, 0)}{\sqrt{2(1+n_z)}}, \frac{1}{2}\sqrt{2(1+n_z)} \right) \ . \tag{2}$$

A direction vector has a zero as its w-coordinate, thus $\hat{\boldsymbol{p}} = (_\perp\vec{\omega}, 0)$, and we have

$$(\vec{\omega}, 0) = \hat{\boldsymbol{q}}\,(_\perp\vec{\omega}, 0)\,\hat{\boldsymbol{q}}^* = \hat{\boldsymbol{q}}\,(x, y, z, 0)\,\hat{\boldsymbol{q}}^* \ .$$

After some algebraic manipulation and using the fact that $\vec{n}$ is of unit length (see Appendix A), we get the result

$$\begin{aligned} \vec{\omega} &= (\hat{\boldsymbol{q}}\,(_\perp\vec{\omega}, 0)\,\hat{\boldsymbol{q}}^*)_v \\ &= \left(x \begin{pmatrix} 1 - n_x^2/(1+n_z) \\ -n_x n_y/(1+n_z) \\ -n_x \end{pmatrix} + y \begin{pmatrix} -n_x n_y/(1+n_z) \\ 1 - n_y^2/(1+n_z) \\ -n_y \end{pmatrix} + z \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \right) . \end{aligned} \tag{3}$$

The equation has a singularity if $1 + n_z = 0$, which happens only when $\vec{n} = (0, 0, -1)$. We can handle this case by setting $\vec{\omega} = (-y, -x, -z)$, which is what the formula would go to for n_z going to -1. Note that this end result involves no square root and that $(1 + n_z)^{-1}$ is used repeatedly and should be stored as a temporary variable. Storing $-n_x n_y/(1 + n_z)$ as a temporary also seems to give a slight advantage.

4. Building an Orthonormal Basis from a 3D Unit Vector

Looking at the formula (3) for rotation of a direction sampled around the z-axis to a direction sampled around an arbitrary direction $\vec{n}$, the three vectors

$$\vec{b}_1 = (1 - n_x^2/(1+n_z), \quad -n_x n_y/(1+n_z), \quad -n_x) \tag{4}$$

$$\vec{b}_2 = (-n_x n_y/(1+n_z), \quad 1 - n_y^2/(1+n_z), \quad -n_y) \tag{5}$$

$$\vec{n} = (n_x, \quad n_y, \quad n_z) \tag{6}$$

are a basis with $\vec{n}$ as the z-axis. We should check that this basis is orthonormal, such that we are certain that we get a unit vector result if the sampled direction ${}_{\perp}\vec{\omega} = (x, y, z)$ is a unit vector. We check this by ensuring that the vectors are of unit length and that the dot products between them are zero (see Appendix B). Because this is true, Equations 4–6 provide a tool for building an orthonormal basis from an arbitrary unit vector $\vec{n}$. The singularity $1 + n_z = 0$ is handled by setting $\vec{b}_1 = (0, -1, 0)$ and $\vec{b}_2 = (-1, 0, 0)$, which is the limit the equations would go to. Code implementing this new method is provided in Listing 2. If we want a function that builds a right-handed orthonormal basis $\{\vec{x}, \vec{y}, \vec{z}\}$ with $\vec{n}$ as the x-axis, we should use

$$\vec{x} = \vec{n} \quad , \quad \vec{y} = \vec{b}_2 \quad , \quad \vec{z} = -\vec{b}_1 \ .$$

5. Performance

The operations needed for the new method are six multiplications, three additions/subtractions, three negations, one division, and a conditional. For comparison, the Hughes-Möller method uses six multiplications, three subtractions, three negations/absolute values (fabs), a normalization, and a conditional. This means that, if we use the new method instead of the Hughes-Möller method, we trade a normalization for a division. Normalization is multiplying the reciprocal square root (rsqrt) of a dot product onto a vector, that is, six multiplications, two additions, and one rsqrt. Thus the new method should nearly double performance.

Table 1 provides a test of the performance of the new method (Listing 2) compared with the methods in Listing 1. To ensure that the test provides a fair indication of the performance gain that we can expect from using the new method, the previous methods have been tested using the different implementations of the rsqrt function in Listing 3. If we want comparable precision, the new method really is almost twice as fast as the Hughes-Möller method.

Listing 2. New way of finding an orthonormal basis from a unit 3D vector.

```
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);
}
```

onb function	rsqrt function	no. of trials (millions)	RMSE
naive	rsqrt	22.6	$5.6 \cdot 10^{-8}$
	InvSqrt	33.8	$7.3 \cdot 10^{-4}$
	SSE_rsqrt	40.8	$6.6 \cdot 10^{-5}$
	SSE_rsqrt_1N	34.3	$6.2 \cdot 10^{-8}$
	no normalization	64.6	$1.5 \cdot 10^{-1}$
hughes_moeller	rsqrt	26.0	$3.2 \cdot 10^{-8}$
	InvSqrt	41.4	$7.9 \cdot 10^{-4}$
	SSE_rsqrt	53.1	$6.7 \cdot 10^{-5}$
	SSE_rsqrt_1N	41.0	$4.0 \cdot 10^{-8}$
	no normalization	73.1	$5.5 \cdot 10^{-2}$
frisvad	not needed	78.5	$7.2 \cdot 10^{-7}$

Table 1. Number of times that we can find an orthonormal basis (onb) in 1 second on a laptop computer with a 2.4 GHz Intel Core2 Duo CPU (using OpenMP to employ both cores). RMSE is root mean square error with respect to length and orthogonality of the vectors.

Listing 3. Different options for the rsqrt function used for normalization.

```
// Using the C++ standard library sqrt function
inline float rsqrt(float x) { return static_cast<float>(1.0/sqrt(x)); }

// See http://en.wikipedia.org/wiki/Fast_inverse_square_root
inline float InvSqrt(float x)
{
  float xhalf = 0.5f*x;
  int i = *(int*)&x;
  i = 0x5f3759df - (i>>1);
  x = *(float*)&i;
  x = x*(1.5f - xhalf*x*x); // (repeat to improve precision)
  return x;
}

// Streaming SIMD Extension (SSE) scalar rsqrt function
inline float SSE_rsqrt(float x)
{
  // The following compiles to movss, rsqrtss, movss
  _mm_store_ss(&x, _mm_rsqrt_ss(_mm_load_ss(&x)));
  return x;
}

// Include a Newton iteration with the SSE rsqrt to improve precision
inline float SSE_rsqrt_1N(float x)
{
  float y = SSE_rsqrt(x);
  return y*(1.5f - 0.5f*x*y*y);
}
```

Surprisingly, the Hughes-Möller method is not faster than the new method if we exclude normalization entirely and accept a large error. Considering the number of operations, it should be faster by a division. My guess is that

the new method is still faster because the condition is almost always false. This should lead to only very few branch mispredictions.

In conclusion, the first tool discussed in this paper is an inexpensive way of transforming a direction sampled in spherical coordinates (with the z-axis as the zenith direction) to a Cartesian unit vector $\vec{\omega}$ specifying the direction as if it were sampled around a specific unit vector $\vec{n}$. Use Equation 3 to do this. The more general result is a tool (Equations 4–6 and Listing 2) for building an orthonormal basis from a unit vector $\vec{n}$ without having to use a dot product and a reciprocal square root for vector normalization.

Acknowledgment. Thanks to an anonymous reviewer for suggesting the naïve way of finding an orthonormal basis (first function in Listing 1). Thanks to Nelson Max for pointing out that the RMSE numbers in the publisher's version of this paper are faulty. The rightmost column of Table 1 has been fixed in this author's version.

A. Applying the Unit Quaternion $\hat{q}$

Using $\hat{\boldsymbol{p}} = (x, y, z, 0)$ and Equation 2, we first find the quaternion product

$$\begin{aligned}
\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^* &= (x, y, z, 0)\left(\frac{(n_y, -n_x, 0)}{\sqrt{2(1+n_z)}}, \frac{1}{2}\sqrt{2(1+n_z)}\right) \\
&= \left(\frac{(zn_x, zn_y, -xn_x - yn_y)}{\sqrt{2(1+n_z)}} + \frac{1}{2}\sqrt{2(1+n_z)}(x, y, z), -\frac{xn_y - yn_x}{\sqrt{2(1+n_z)}}\right) \\
&= \frac{1}{\sqrt{2(1+n_z)}}\left(x\begin{pmatrix}1+n_z\\0\\-n_x\end{pmatrix} + y\begin{pmatrix}0\\1+n_z\\-n_y\end{pmatrix} + z\begin{pmatrix}n_x\\n_y\\1+n_z\end{pmatrix}, yn_x - xn_y\right).
\end{aligned}$$

Completing the application of the unit quaternion $\hat{\boldsymbol{q}}$, we get

$$\begin{aligned}
&2(1+n_z)(\hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^*)_v \\
&= \begin{pmatrix} -xn_x^2 - yn_xn_y + zn_x(1+n_z) + (xn_y - yn_x)n_y + (1+n_z)(x(1+n_z) + zn_x) \\ -xn_xn_y - yn_y^2 + zn_y(1+n_z) - (xn_y - yn_x)n_x + (1+n_z)(y(1+n_z) + zn_y) \\ -xn_x(1+n_z) - yn_y(1+n_z) - z(n_x + n_y^2) - (1+n_z)(xn_x + yn_y - z(1+n_z)) \end{pmatrix} \\
&= \left(x\begin{pmatrix} n_y^2 - n_x^2 + (1+n_z)^2 \\ -2n_xn_y \\ -2n_x(1+n_z)\end{pmatrix} + y\begin{pmatrix} -2n_xn_y \\ n_x^2 - n_y^2 + (1+n_z)^2 \\ -2n_y(1+n_z)\end{pmatrix} + z\begin{pmatrix} 2n_x(1+n_z) \\ 2n_y(1+n_z) \\ (1+n_z)^2 - n_x^2 - n_y^2\end{pmatrix}\right).
\end{aligned}$$

Note that the square root disappears. Now we can use the fact that $\vec{n}$ is a unit vector to simplify the equation further. Because $\vec{n}$ is of unit length, we have:

$$\begin{aligned}
n_x^2 + n_y^2 + n_z^2 &= 1 \\
n_y^2 - n_x^2 + (1+n_z)^2 &= 1 - n_x^2 + n_y^2 + n_z^2 + 2n_z = 2(1+n_z) - 2n_x^2 \\
n_x^2 - n_y^2 + (1+n_z)^2 &= 1 + n_x^2 - n_y^2 + n_z^2 + 2n_z = 2(1+n_z) - 2n_y^2 \\
(1+n_z)^2 - n_x^2 - n_y^2 &= (1+n_z)^2 - (1+n_z)(1-n_z) \ .
\end{aligned}$$

Thus, when the equation is divided on both sides by $2(1+n_z)$, we have

$$\begin{aligned}
\vec{\omega} &= (\hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^*)_v \\
&= \left(x \begin{pmatrix} 1 - n_x^2/(1+n_z) \\ -n_x n_y/(1+n_z) \\ -n_x \end{pmatrix} + y \begin{pmatrix} -n_x n_y/(1+n_z) \\ 1 - n_y^2/(1+n_z) \\ -n_y \end{pmatrix} + z \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \right) .
\end{aligned}$$

B. Proof That the Basis We Build Is Orthonormal

Using that $\vec{n}$ is a unit vector, the squared length of $\vec{b}_1$ from Equations 4–6 is

$$\begin{aligned}
\vec{b}_1 \cdot \vec{b}_1 &= \left(1 - \frac{n_x^2}{1+n_z}\right)^2 + \left(\frac{n_x n_y}{1+n_z}\right)^2 + n_x^2 \\
&= 1 + \frac{n_x^4 - n_x^2(2(1+n_z) - n_y^2 - (1+n_z)^2)}{(1+n_z)^2} \\
&= 1 + \frac{n_x^4 - n_x^2(n_x^2)}{(1+n_z)^2} = 1 \ .
\end{aligned}$$

Swapping the x and y subscripts in this calculation, we also get $|\vec{b_2}| = 1$, thus all three basis vectors are of unit length. The cosine of the angle between $\vec{b}_1$ and $\vec{n}$ is

$$\begin{aligned}
\vec{b}_1 \cdot \vec{n} &= \frac{n_x(1+n_z) - n_x^3 - n_x n_y^2 - n_x n_z(1+n_z)}{1+n_z} \\
&= \frac{n_x(1 - n_x^2 - n_y^2 - n_z^2)}{1+n_z} = 0 \ ,
\end{aligned}$$

and similarly $\vec{b}_2 \cdot \vec{n} = 0$. Finally,

$$\vec{b}_1 \cdot \vec{b}_2 = \frac{n_x^3 n_y - 2n_x n_y(1+n_z) + n_x n_y^3 + n_x n_y (1+n_z)^2}{(1+n_z)^2}$$

$$= \frac{n_x n_y (n_x^2 + n_y^2 + (1+n_z)^2 - 2(1+n_z))}{(1+n_z)^2} = 0 \ ,$$

which proves that the presented formulae build an orthonormal basis from a unit vector without using vector normalization explicitly.

References

[Akenine-Möller et al. 08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*, Third edition. Natick, MA: A K Peters, 2008.

[Hamilton 44] William Rowan Hamilton. "On Quaternions; or on a new System of Imaginaries in Algebra." *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science (3rd series)* 25 (1844), 10–13.

[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.

[Pharr and Humphreys 10] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*, Second edition. Burlington, MA: Morgan Kaufmann/Elsevier, 2010.

Web Information:

Supplementary source code data: http://www2.imm.dtu.dk/pubdb/p.php?6303

Jeppe Revall Frisvad, Technical University of Denmark, DTU Informatics, Asmussens Alle, Building 305, 2800 Kgs. Lyngby, Denmark. (jrf@imm.dtu.dk)

Submitted July 21, 2011; Recommended April 10, 2012; Accepted April 25, 2012. Edited by Morgan McGuire.

This is an electronic version of an article published in Journal of Graphics Tools 16(3):151–159, August 2012. The Journal of Graphics Tools is available online at: http://www.tandfonline.com/doi/abs/10.1080/2165347X.2012.689606