Vol. 13, No. 3: 53–68

Fast Equal-Area Mapping of the (Hemi)Sphere using SIMD

Petrik Clarberg
Lund University

Abstract. We present a fast vectorized implementation of a transform that maps points in the unit square to the surface of the sphere, while preserving fractional area. The mapping uses the octahedral map combined with an equal-area parameterization and has many desirable features such as low distortion, straightforward interpolation, and fast inverse and forward transforms. Our SIMD implementation completely avoids branching and uses polynomial approximations for the trigonometric operations, along with other tricks. This results in up to 9 times speed-up over a traditional scalar implementation. Source code is available online.

1. Introduction

Spherical and hemispherical functions are abundant in computer graphics. Examples include environment maps, BRDFs, visibility data, surface maps for spherically parameterized objects, and so on. To handle such data, we need a mapping from the (hemi)spherical domain to the plane. The best choice of mapping depends on the application. For example, the *cube map* is convenient because of its simplicity and hardware support, but it is not area-preserving (pixels near the corners represent a smaller solid angle).

In many cases, it is desirable to use an *equal-area* mapping, i.e., a mapping that preserves fractional area. In applications integrating functions over the (hemi)sphere, area preservation significantly simplifies the implementation as we do not have to take the solid angle of each pixel into account. The prime

1086-7651/08 $0.50 per page

example is the evaluation of the rendering equation [Kajiya 86], which involves an integration over the hemisphere. Another desirable property is *low distortion*, i.e., the aspect ratio of pixels on the sphere should be close to one. Otherwise, square pixels can be mapped to long, thin segments on the sphere, which causes aliasing and reduces the useful resolution. It is also desirable to have as few discontinuities as possible in order to simplify interpolation.

In this paper, we describe a fast implementation of a mapping with all these properties: equal area, low distortion, and support for straightforward interpolation across edges. The mapping uses the octahedral map [Praun and Hoppe 03] combined with an area-preserving parameterization [Shirley and Chiu 97]. The described mapping has been successfully used for importance-sampling purposes [Clarberg and Akenine-Möller 08].

The current trend is microprocessors with many cores and wide data paths. By exploiting data parallelism using SIMD (single instruction, multiple data) vectorization, the performance of many applications can be greatly improved. We provide a SIMD implementation (using Intel SSE) of the described mapping that is up to 9× faster than a straightforward scalar implementation and roughly 4× faster than an optimized scalar version. Most of this paper deals with the technical details of our implementation, such as avoiding branching, polynomial approximations of the trigonometric operations, etc. We believe this is of interest to a wide audience, as knowledge about how to write SIMD code is becoming increasingly important to fully utilize the enormous performance available in modern CPUs.

2. Equal-Area Mapping

2.1. Hemisphere

For mapping the hemisphere, we use the *concentric map* [Shirley and Chiu 97], which maps concentric squares to concentric circles on the hemisphere, while preserving fractional area (see Figure 1). For the first sector, i.e., where $\phi \in [-\frac{\pi}{4}, \frac{\pi}{4}]$, a point (s,t) in the unit square $\mathcal{P} = [0,1]^2$ is transformed to a point on the hemisphere $\mathcal{H} = \{(x,y,z) \mid x^2+y^2+z^2=1, z \geq 0\}$ as follows:

$$(s,t) \rightarrow \begin{array}{l} u = 2s-1 \\ v = 2t-1 \end{array} \rightarrow \begin{array}{l} r = u \\ \phi = \frac{\pi}{4}\frac{v}{u} \end{array} \rightarrow \begin{array}{l} x = \cos\phi \cdot r\sqrt{2-r^2} \\ y = \sin\phi \cdot r\sqrt{2-r^2} \\ z = 1-r^2 \end{array} . \qquad (1)$$

Similar transforms apply to the other sectors. The z-coordinate in the last step is equal to $z = 1-r^2 = \cos\theta$, where θ is the angle from the z-axis. Hence, $\sin\theta = \sqrt{1-\cos^2\theta} = r\sqrt{2-r^2}$, which explains the equations for x and y.

As noted by Shirley and Chiu, this simple mapping has a number of desirable properties. Most importantly, it preserves fractional area, which means a

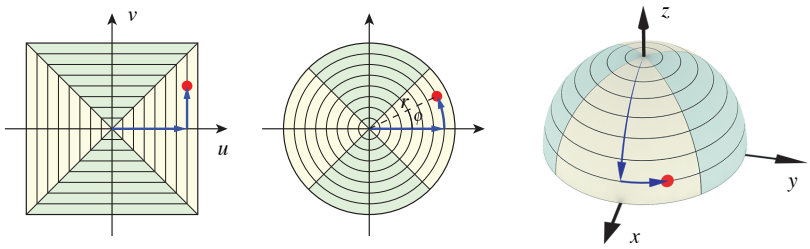

Figure 1. The concentric map [Shirley and Chiu 97] transforms concentric squares in the plane to concentric circles on the hemisphere. The mapping preserves fractional area and has a relatively low distortion.

uniform point distribution in the square will map to a uniform distribution on the hemisphere. Second, the mapping preserves adjacency, i.e., nearby points in the square map to nearby points on the hemisphere. Last, the distortion is relatively well-behaved, which is important in order to reduce aliasing when sampling functions over the hemisphere.

2.2. Sphere

To get an equal-area mapping of the sphere, we combine the concentric map with the *octahedral map* [Praun and Hoppe 03], which is a clever way to "fold" a square over the sphere. The square is divided into eight triangles, where the four innermost triangles are mapped to the northern hemisphere, while the outer four are folded down to cover the southern hemisphere. Thus, each triangle maps to a quadrant in one of the two hemispheres, as illustrated in Figure 2. We rotate the concentric map by 45° and insert it into the inner quad of the octahedral map. To the best of our knowledge, this combination of the two mappings was first used in [Clarberg and Akenine-Möller 08].

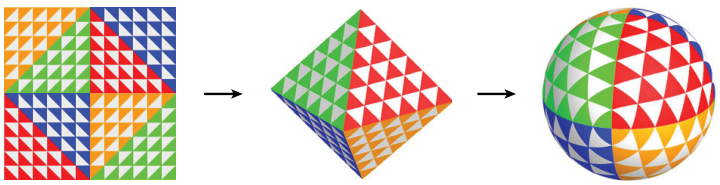

Figure 2. The octahedral map [Praun and Hoppe 03] is obtained by folding a quad into an octahedron, which is projected onto the sphere using an arbitrary parameterization. Image courtesy of Emil Praun and Hugues Hoppe.

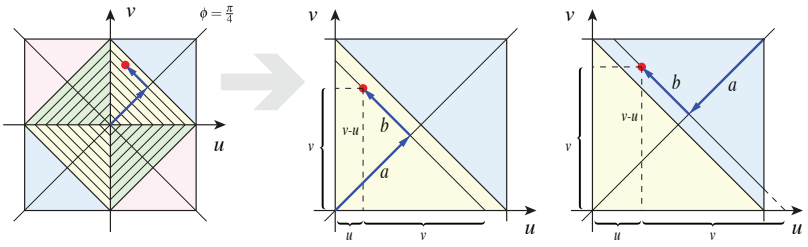

Figure 3. The concentric map applied to the first quadrant of the octahedral map. The inner region (yellow) maps to the northern hemisphere, while the outer (blue) maps to the southern. The lengths a and b are found using simple trigonometry.

The transform from the unit square $\mathcal{P}$ to the sphere $\mathcal{S} = \{(x, y, z) \mid x^2 + y^2 + z^2 = 1\}$ is easy to derive. As before, the first step is to transform (s, t) to a point $(u, v) \in [-1, 1]^2$. We start by considering the innermost triangle in the first quadrant, as shown in Figure 3. The lengths of a and b are $a = (u+v)/\sqrt{2}$ and $b = (v-u)/\sqrt{2}$, and the transform to the unit disk is given by

$$\begin{aligned} r &= \sqrt{2}a = u + v, \\ \phi &= \frac{\pi}{4}\frac{b}{a} + \frac{\pi}{4} = \frac{\pi}{4}\left(\frac{v-u}{r} + 1\right), \qquad \text{with } 0 \le \phi \le \frac{\pi}{2}, \end{aligned} \tag{2}$$

where ϕ is measured from the positive u-axis (hence the addition of $\frac{\pi}{4}$). The outermost triangle maps to the southern hemisphere, and its parameterization is obtained by mirroring the innermost triangle about the diagonal. Here, $b = (v-u)/\sqrt{2}$ as before, but a differs slightly and is given as $a = (2-u-v)/\sqrt{2}$. The transform to the unit disk is

$$\begin{aligned} r &= \sqrt{2}a = 2 - u - v, \\ \phi &= \frac{\pi}{4}\frac{b}{a} + \frac{\pi}{4} = \frac{\pi}{4}\left(\frac{v-u}{r} + 1\right), \qquad \text{with } 0 \le \phi \le \frac{\pi}{2}. \end{aligned} \tag{3}$$

Note that the computation of ϕ is the same in both these cases, only r differs. The mapping from the disk to the sphere is the same as the last step of Equation (1), except that the z-component is negated, i.e., $z = -(1-r^2)$, if we are in the outer triangle. The final mapping is shown in Figure 4, and a proof of area preservation is given in Section 6.

So far, we have ignored the remaining quadrants. However, following a similar reasoning as above, the transforms are easily derived. A straightforward implementation, similar to Shirley and Chiu's version for the hemisphere, results in three levels of `if`-statements as there are eight different cases (four quadrants, each divided into two triangles). The main execution cost lies in this branching, together with the trigonometric operations.

3. SIMD Implementation

We will now describe how each part of the algorithm can be efficiently written using the x86 streaming SIMD extensions (SSE). SSE code can be written directly using inline assembly language instructions, or using compiler intrinsics (which we use). Carefully hand-optimized assembly code can be faster but is tedious and error-prone to write. Intrinsics, which is a C/C++ mapping of the assembly instructions, enable features like compiler optimizations and automatic register allocation, which make them easier to use. Intrinsics are also likely to be more forward compatible (e.g., if more registers are added, the code can, after recompilation, automatically benefit). With SSE, four floating-point values are processed in parallel, but future-generation CPUs will likely have wider data paths. As we do not use any horizontal operations, it is trivial to adapt our implementation to such architectures.

3.1. The Square-to-Sphere Transform

3.1.1. Avoiding Branching

In SIMD code, branching is handled using conditional instructions which set a bit mask based on the outcome of some comparison. By executing both branches and selecting the correct result based on the mask using logical operations (`and`, `or`, `not`), the equivalent of a "parallel" `if`-statement is created.

In our case, there are eight different ways to compute (r, ϕ), which makes this approach inefficient. Hence, we take an alternative route and map the problem to the first quadrant by taking the absolute values of u and v. A similar approach is used to find r without using any conditional instructions (we will come back to this in a moment). We use the following:

$$\phi' = \frac{\pi}{4}\left(\frac{|v| - |u|}{r} + 1\right), \qquad \text{where } \phi' \in [0, \frac{\pi}{2}]. \tag{4}$$

In the last step of Equation (1), we need to compute the sine and cosine of ϕ, but we only have ϕ' to work with. Using standard trigonometric rules, we find the following expressions for $\sin\phi$ and $\cos\phi$ in each of the four quadrants:

quadrant	ϕ	$\sin\phi$	$\cos\phi$
1	ϕ'	$\sin\phi'$	$\cos\phi'$
2	$\pi - \phi'$	$\sin\phi'$	$-\cos\phi'$
3	$\phi' - \pi$	$-\sin\phi'$	$-\cos\phi'$
4	$-\phi'$	$-\sin\phi'$	$\cos\phi'$

Based on this, we realize that

$$\begin{array}{rcl} \sin\phi & = & \text{sign}(v)\cdot\sin\phi', \\ \cos\phi & = & \text{sign}(u)\cdot\cos\phi', \end{array} \qquad \text{where } \text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0. \end{cases} \tag{5}$$

Fortunately, both the absolute operator needed in Equation (4), and the sign operator used in Equation (5), can be efficiently implemented using simple logic instructions. In the IEEE-754 standard for floating-point numbers, the most significant bit (MSB) represents the sign, where 0 means positive and 1 means negative. Hence, taking the absolute value is simply a matter of `and`'ing by the bit mask $01\ldots1_{\text{b}} = \text{0x7F}\ldots\text{F}$, and changing the sign can be done by `xor`'ing with the sign-bit: $10\ldots0_{\text{b}} = \text{0x80}\ldots0$. More formally,

$$\begin{array}{rcl} |u| & = & u \;\&\; \text{0x7FFFFFFF}, \\ \text{sign}(u)\cdot x & = & (u \;\&\; \text{0x80000000}) \oplus x, \end{array}$$

where $\oplus$ represents `xor` and & means `and`. In practice, we use the `andnps` instruction (`and` a value with the logical inverse of another) to avoid loading two different constants from memory as $\text{0x80}\ldots0$ is the inverse of $\text{0x7F}\ldots\text{F}$.

We apply a similar reasoning to find r (Equations (2) and (3)) and the z-coordinate (Equation (1)) without branching. First, we compute the signed distance d along the diagonal in the first quadrant, so that $d{=}1$ at the origin, $d{=}{-}1$ at the upper-right corner, and $d = 0$ halfway between. By taking the absolute value, we can thus compute r as $1 - |d|$. Also note that a positive distance (inner triangle) represents points on the northern hemisphere, while negative values map to the southern hemisphere. Hence, the sign of d can be used to set the sign of z. We arrive at the following:

$$d = 1 - (|u| + |v|), \qquad \text{and} \qquad \begin{array}{rcl} r & = & 1 - |d|, \\ z & = & \text{sign}(d)\cdot(1 - r^2). \end{array}$$

3.1.2. Avoiding Division-by-Zero

In the center and the four corners of the square, r is exactly zero, and Equation (4) results in a division-by-zero. To get a robust solution, we set $\phi' = 0$ if $r = 0$. This is a valid behavior as these points map to the south and north poles, respectively, where the value of ϕ' does not matter. Using intrinsics, the test can be compactly written as follows:

```
mask = _mm_cmpneq_ps(r, ZERO);   // compare r to zero
phi = _mm_and_ps(phi, mask);     // clear phi to 0..0 if r=0
```

The `cmpneqps` (compare not equal) instruction compares r with a predefined zero constant, and based on the result, ϕ' is either cleared to zero or kept unchanged. This is valid since the bit sequence $0\ldots0$ also represents a floating-point zero in the IEEE-754 standard.

3.1.3. Approximating the Trigonometric Operations

Using simple logical operations, we map the problem to the first quadrant, and from there, sign extension moves the result back to the correct quadrant. The only remaining difficulty is the trigonometric operations needed to compute x and y in Equation (1). As `sin` and `cos` are not part of the SSE instruction set, polynomial approximations have to be used. Table-based approaches are not recommended as multiple values are computed in parallel, which means multiple memory accesses at different locations and bad cache performance. The most straightforward approach is to truncate the Taylor series for sine and cosine to an arbitrary number of terms:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad \text{and} \quad \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots .$$

The Taylor series are obtained by expanding the power series around the point $x = 0$. However, the absolute error grows the further from x we get. In our case, the input is in the range $\phi' \in [0, \frac{\pi}{2}]$. If we truncate the expression for $\sin x$ after, e.g., the seventh-degree term, we get a maximum absolute error of $1.57 \cdot 10^{-4}$, which is quite large. The power series can be expanded around a point somewhere in the middle of the interval, but the basic problem remains—the error increases quickly as we move away from the chosen point.

A better approach is to use the *minimax* polynomial approximation, which is the polynomial that minimizes the maximum approximation error. This has a number of interesting properties. The Chebyshev equioscillation theorem (see, e.g., [Hart 78]) states that for a minimax polynomial of degree n over an interval $a \le x \le b$, there exist at least $n+2$ points in the domain where the error between the polynomial and the approximated function oscillate in sign and are of equal magnitude. Thus, the minimax polynomial gives us a well-controlled error over the entire range. The Remez algorithm [Fraser 65] can be used to find the polynomial, but as the details are quite technical, we use the implementation provided by the `numapprox` package in Maple. As an example, the seventh-order minimax approximation of $\sin x$ for $x \in [0, \frac{\pi}{2}]$ is

$$\begin{aligned} \sin x \;\approx\; & -1.947 \cdot 10^{-8} + 1.00000155x - 0.000020227x^2 - 0.16657x^3 \\ & -0.00023977x^4 + 0.0086393x^5 - 0.00020575x^6 - 0.00013731x^7, \end{aligned}$$

with a maximum error of only $2.00 \cdot 10^{-8}$, as compared to $1.57 \cdot 10^{-4}$ for the Taylor approximation of the same order. However, note that the evaluation is more expensive, as every term has a nonzero coefficient (seven vs. three `mul`'s). To reduce the complexity, we set coefficients for terms of even power to zero, as these are relatively small anyway. This is done by rewriting the problem as

$$\sin x \approx x p(x^2) \quad \overset{y=x^2}{\Longleftrightarrow} \quad \frac{\sin\sqrt{y}}{\sqrt{y}} \approx p(y).$$

After a change of variables, $y = x^2$, we can find a third-order polynomial $p(y)$, which after insertion on the left side gives a seventh-order approximation of $\sin x$ with all even coefficients set to zero. Note that the range used for the minimax optimization must be updated to $y \in [0, \frac{\pi^2}{4}]$ as $x \in [0, \frac{\pi}{2}]$. The maximum error is now $1.18 \cdot 10^{-6}$ at a cost of 4 `mul`'s, which is a good compromise.

In our case, we need to compute both $\sin\phi'$ and $\cos\phi'$. The first thing that comes to mind is to approximate only the sine, and then apply the rule $\cos x = \sqrt{1 - \sin^2 x}$ to find the cosine. However, the approximation of $\sin x$, which we call $f(x)$, results in an error $\varepsilon_s(x) = \sin x - f(x)$. By analyzing $\varepsilon_s(x)$ over the range $0 \leq x \leq \frac{\pi}{2}$, we find that the largest error is $\varepsilon_s = 1.18 \cdot 10^{-6}$ at $x = \frac{\pi}{2}$. The error in the cosine approximaton at this point would be

$$|\varepsilon_c| = |\cos x - \sqrt{1 - (\sin x - \varepsilon_s)^2}| = \sqrt{\varepsilon_s(2 - \varepsilon_s)} \approx \sqrt{2\varepsilon_s} = 0.0015,$$

which is too large. Hence, we have to perform two separate polynomial approximations (sine and cosine) in order to reach an acceptable precision.

Last, we note that the computation of ϕ' in Equation (4) includes a multiplication by $\frac{\pi}{4}$. By rescaling the coefficients of the minimax polynomials so that we approximate $\sin(\frac{\pi}{4}x)$ rather than $\sin x$, and similarly for the cosine, this extra multiplication can be avoided (without changing the approximation error as we only rescale the input). Finally, we arrive at

$$\begin{aligned}
\sin(\frac{\pi}{4}x) &\approx 0.785398x - 0.0807407x^3 + 0.00248440x^5 - 0.0000341486x^7, \\
\cos(\frac{\pi}{4}x) &\approx 0.999993 - 0.308371x^2 + 0.0157863x^4 - 0.000298371x^6,
\end{aligned}$$

where $x \in [0, 2]$ and the coefficients have been rounded to 6 significant digits (see the source code for full precision).

3.2. The Sphere-to-Square Transform

The inverse transform, i.e., the mapping of vectors on the sphere to points in the unit square, is useful in a number of applications. The main steps are as follows.

1. Map the problem to the first quadrant by taking the absolute of (x, y).
2. Compute ϕ' from (x, y) using `atan`, and compute r from z.
3. Convert (r, ϕ') to (u, v) in the first quadrant.
4. Flip the sign bits of (u, v) to move the point to the correct quadrant.
5. Map points from $[-1, 1]^2$ to the unit square $[0, 1]^2$.

As many parts of the implementation resemble those described in Section 3.1, we will only briefly go over the details.

3.2.1. Approximating the Arctan Function

Our input is a normalized 3D vector (x, y, z). To compute ϕ', which is the rotation in the first quadrant of the xy-plane, we start by computing the absolute values $|x|$ and $|y|$. The rationale for this is the same as before, i.e., to move the problem to the first quadrant. Then, the rotation is found using $\phi' = \text{atan}\ \frac{|y|}{|x|}$. Since there is no SSE version of `atan`, we have to use a polynomial approximation here as well. However, it proved hard to find an approximation that yields enough precision for all inputs, as $\frac{|y|}{|x|} \to +\infty$ as $|x| \to 0$. Therefore, we apply the rule

$$\text{atan}\ \alpha = \begin{cases} \text{atan}\ \alpha & \text{if} \quad \alpha < 1, \\ \frac{\pi}{2} - \text{atan}\ \frac{1}{\alpha} & \text{if} \quad \alpha \geq 1, \end{cases} \tag{6}$$

to reduce the input range to $[0, 1]$, i.e., we let $|x|$ and $|y|$ switch places if $|x| < |y|$. Using the SSE instructions `minps` (minimum of two values) and `maxps` (maximum of two values), α can be efficiently computed as

$$\alpha = \frac{\min(|x|, |y|)}{\max(|x|, |y|)}, \qquad 0 \leq \alpha \leq 1.$$

With this reduced range, it is easier to find a good minimax approximation. We strive for about the same precision as in the approximations of sine and cosine. For the atan function, *rational* minimax approximations are known to give low errors. In our case, a third- or second-order approximation would be sufficient (maximum error of $7.28 \cdot 10^{-6}$). However, the necessary division is rather slow and has a long latency. Thus, we opted for a sixth-order polynomial approximation, which avoids the division and uses the same number of coefficients. As we will see later, it is useful to include a multiplication by $\frac{2}{\pi}$ so that the approximation returns an angle in $[0, \frac{1}{2}]$ rather than $[0, \frac{\pi}{4}]$. Our final approximation is (note $\alpha \in [0, 1]$)

$$\begin{aligned} \frac{2}{\pi}\ \text{atan}\ \alpha \ \approx\ & 4.06531 \cdot 10^{-6} + 0.636227\alpha + 0.00615523\alpha^2 - 0.247326\alpha^3 + \\ & 0.0881627\alpha^4 + 0.0419157\alpha^5 - 0.0251427\alpha^6, \end{aligned}$$

and the maximum error is $4.07 \cdot 10^{-6}$. Last, we need to evaluate Equation (6) (scaled by $\frac{2}{\pi}$). This can be done using a compare instruction followed by four logic/arithmetic instructions, as follows:

```
__m128 mask = _mm_cmplt_ps(x, y);     // mask = x<y ? 11..1 : 0
__m128 c = _mm_and_ps(mask, ONE);     // c = x<y ? 1.0 : 0.0
mask = _mm_and_ps(mask, SIGNBIT);     // mask = x<y ? 10..0 : 0
phi = _mm_xor_ps(phi, mask);          // phi = x<y ? -phi : phi
phi = _mm_add_ps(c, phi);             // phi = x<y ? 1-phi : phi
```

3.2.2. Finding the Radius

The computation of r differs depending on whether we are in the northern or the southern hemisphere ($z \geq 0$ or $z < 0$). By rearranging the terms in the equations for z given in Section 2.2, we find that

$$r = \begin{cases} \sqrt{1-z} & \text{if } z \geq 0 \\ \sqrt{1+z} & \text{if } z < 0 \end{cases} \qquad \Longleftrightarrow \qquad r = \sqrt{1-|z|}.$$

Again, by taking the absolute value, we avoid branching.

3.2.3. Mapping from Disc to Square

The computation of the point (u, v) in the square, corresponding to the point (r, ϕ') in the disc, is relatively straightforward. We assume, for now, that the point lies in the "inner" triangle (northern hemisphere). By inverting the expressions for r and ϕ (Equation (2)), we can compute (u, v) as

$$\begin{cases} r = u + v \\ \phi' = \frac{\pi}{4}\left(\frac{v-u}{r} + 1\right) = \ldots = \frac{\pi}{2}\frac{v}{r} \end{cases} \qquad \Longleftrightarrow \qquad \begin{cases} v = r \cdot \frac{2}{\pi}\phi' \\ u = r - v. \end{cases}$$

Then, if we are in the southern hemisphere ($z < 0$), the point (u, v) is reflected about the diagonal in the square as follows:

$$u' = 1 - v \qquad \text{and} \qquad v' = 1 - u.$$

This is implemented using a compare instruction followed by logic/arithmetic instructions similar to what we did in Section 3.2.1. The next step is to sign-extend (u, v) based on the original signs of x and y, i.e., we take the point from the first quadrant to its correct position. Finally, we transform the point to the unit square, $[-1, 1]^2 \rightarrow [0, 1]^2$, which completes the transform.

3.3. Precision

The approximations of the trigonometric operations were chosen to provide sufficient precision for all but the most demanding applications. To measure the approximation errors, we transformed a large number (10^9) of random points in the square to the sphere, using both the "exact" scalar version of the algorithm (using built-in trigonometric instructions) and our SSE-optimized version. The error was measured as the Euclidean distance in 3D between the resulting points on the sphere. As the error is very small, this measure is approximately the same as the arc length in radians on the unit sphere.

For the inverse transform, a large number of points over the sphere were mapped to points in the square. To measure the error, these 2D points were

transform	maximum error	average error
square→sphere	$7.49 \cdot 10^{-6}$	$3.37 \cdot 10^{-6}$
sphere→square	$2.43 \cdot 10^{-4}$	$3.19 \cdot 10^{-6}$

Table 1. The maximum and average approximation errors of the forward and inverse transforms. The errors were measured as the Euclidean distance between the points on the unit sphere representing the exact and approximated directions.

transformed back to the sphere using the exact algorithm, and the Euclidean distance was measured as before. The maximum and average errors are given in Table 1. The average error is on the order of $3 \cdot 10^{-6}$ in both directions. As a comparison, the diagonal of a single pixel in a 4096×4096 image has a shortest length of $7.7 \cdot 10^{-4}$ when mapped to the sphere.[1] The average error is thus only about 0.3% of the edge length of a pixel in a 4k map. The maximum error is well-behaved in the square-to-sphere transform. In its inverse, there are a few bad cases where the error goes up to about 1/3 of a pixel (again assuming a 4k map), even though the average error is low. If a higher precision is required, it is easy to increase the order of the arctan approximation.

4. Boundary Symmetry

4.1. Tiling

Each edge of the octahedral map is folded about its midpoint so that its two endpoints meet. This *boundary symmetry* [Praun and Hoppe 03] is useful as it means the map can be tiled by mirroring the mapping about both its axes for every other occurrence,[2] as shown in Figure 4(c). Hence, look-ups for coordinates outside the $[0, 1)^2$ range are trivial, and interpolation across the edges of the map is well-defined and without singularities.

For a map with $N \times N$ pixels, where $N = 2^k$ is a power of two, the coordinate transform from a point (s, t) to integer pixel coordinates (x, y) can be efficiently done using logic operations. We scale the input by N and truncate:

$$x = \lfloor s \cdot N \rfloor \qquad \text{and} \qquad y = \lfloor t \cdot N \rfloor,$$

where $\lfloor . \rfloor$ is the floor operator. The bits $0, \ldots, k$–1 of the binary representation of x and y hold the pixel coordinates within the square, while bits $k, k+1, \ldots$ indicate which repetition of the tiling we are in. Mirroring needs to be done when either x *or* y is at an odd number of repetitions. We can thus `xor` x

[1] With a map of $N \times N$ pixels, there are $2N$ pixels crossing the equator, while the circumference of the unit sphere is 2π. Hence, the shortest diagonal is of length π/N.

[2] This is a variant of the "mirrored repeat" tiling used by OpenGL.

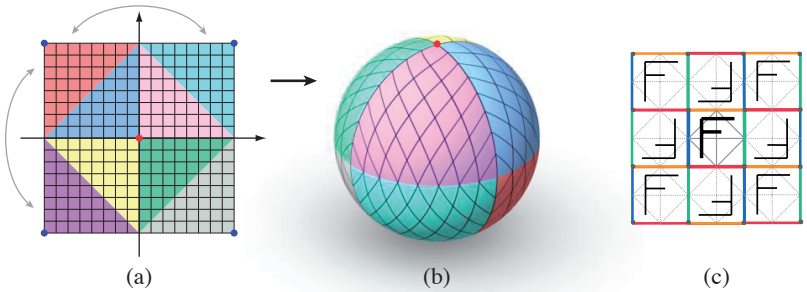

Figure 4. Our mapping transforms square pixels (a) into curvilinear quads on the sphere (b), while preserving fractional area. The boundary symmetry [Praun and Hoppe 03] of the octahedral map is shown in (c). The rightmost image is courtesy of Emil Praun and Hugues Hoppe.

and y and look at the kth bit to decide whether to mirror or not:

$$m = (x \oplus y)\ \&\ N, \qquad \text{if} \begin{cases} m = 0 & \rightarrow \text{ do not mirror,} \\ m \neq 0 & \rightarrow \text{ mirror.} \end{cases}$$

The final positions are computed as $(x, y) \mod N$, followed by $x = (N-1) - x$ (and similar for y) if $m \neq 0$. The mod-operator translates to an `and` by $N-1$, which is the bit mask with 1's at the positions $0, \ldots, k-1$ and 0's elsewhere.

4.2. Bilinear Interpolation

For bilinear interpolation, the four pixels nearest to the point (s, t) must be accessed, and their respective interpolation weights computed. In a map of $N \times N$ pixels, we define each pixel's center to be at $(x+0.5, y+0.5)/N$, where (x, y) are the integer coordinates of the pixel. To find the coordinates of the top-left pixel, we scale (s, t) by N and offset by 0.5, as follows:

$$x = \lfloor s \cdot N - 0.5 \rfloor \qquad \text{and} \qquad y = \lfloor t \cdot N - 0.5 \rfloor. \tag{7}$$

The fractional distance (α_x, α_y) from the top-left pixel's center is given by $\alpha_x = (s \cdot N - 0.5) - x$ (similar for α_y). Based on this, the coordinates and the weights for the bilinear interpolation are given in Table 2.

As the coordinates may lie outside the $\{0, \ldots, N-1\}$ range, we perform the wrapping described in Section 4.1 on each of the four pixels. For this purpose, we have written a code snippet that computes all four pixel positions, and their respective weights, in parallel using SIMD instructions. We store x and y as 16-bit integers and pack all eight combinations into a single 128-bit

pixel	x-coordinate	y-coordinate	weight
0	x	y	$(1-\alpha_x)\cdot(1-\alpha_y)$
1	$x+1$	y	$\alpha_x\cdot(1-\alpha_y)$
2	x	$y+1$	$(1-\alpha_x)\cdot\alpha_y$
3	$x+1$	$y+1$	$\alpha_x\cdot\alpha_y$

Table 2. Coordinates and weights for the bilinear interpolation.

XMM register. Using SSE2 integer instructions, we perform the wrapping on all eight values in parallel. Finally, the coordinates are unpacked into 32-bit integers and the pixel addresses computed using bit shifts.

The lack of a floor operator in the SSE/SSE2 instruction set presents a minor difficulty. One option is to use `cvttps2dq` (convert float to int with truncation), but this gives unexpected results for negative inputs as it truncates towards zero. Instead, we rewrite the floor operator using rounding:

$$\lfloor x \rfloor = \text{round}(x - 0.5).$$

The `cvtps2dq` (convert float to int) instruction performs correct rounding (assuming the rounding control bits in the MXCSR register have been correctly set up). Thus, Equation (7) can be rewritten:

$$x = \text{round}(s \cdot N) - 1 \qquad \text{and} \qquad y = \text{round}(t \cdot N) - 1.$$

In total, we use 27 SSE/SSE2 instructions to compute all pixel indices and weights. As we use 16-bit integers, the map size is limited to 64k $\times$ 64k pixels.

5. Performance

We have implemented three different versions of the forward and inverse transforms: 1) a straightforward scalar version with branching and trigonometric operations, 2) an optimized scalar version using the tricks described here, and 3) a vectorized version using SSE instructions to transform four points/vectors in parallel. To evaluate the performance, we have run the algorithms on sets of N random 2D points in the unit square and N random 3D vectors on the sphere, respectively, using a large number of iterations.

As the computations are performed repeatedly on the same data, we largely avoid cache effects and measure pure computational performance. For the larger datasets, the memory bandwidth limits the performance slightly. The total memory use for each test is $20N$ bytes, and all timings are reported as *clock cycles per single transform*. Thus, transforming a set of N points/vectors takes approximately Nt clock cycles, where t is the reported timing. Note that N has to be a multiple of four, as we use 4-wide SIMD code. All tests were executed on an Intel 45nm quad-core "Penryn" CPU running at 3.2GHz using one core, and the code was compiled using gcc 4.0.1 on Mac OS X 10.5.2.

N	scalar standard	scalar optimized	SSE optimized
256	121.9	77.9	18.8
4k	156.2	77.9	18.7
64k	160.7	78.0	18.7
1M	162.4	80.4	22.8
16M	162.5	80.5	22.8

Table 3. Execution times of our three different implementations of the square-to-sphere transform, using datasets of N 2D points as input. The timings are reported as number of clock cycles per transformed point.

The performance of the square-to-sphere transform is summarized in Table 3. The vectorized version is a factor of 3.5×–4.2× faster than the optimized scalar code and 6.5×–8.6× faster than the standard version. The numbers for the inverse transform (sphere to square) are listed in Table 4. Here, the speed-up is a factor of 4.6×–5.0× compared to the optimized scalar code and 5.0×–6.4× compared to the standard version. Theoretically, SSE-optimized code should be up to a factor of 4× faster than a corresponding scalar implementation, as it uses 4-wide instructions. However, the differences in the instruction sets make it possible to exceed this limit. For example, branching is avoided by using compare instructions together with logical operations.

The achieved speed-up can make a significant difference. As an example, consider ray tracing with 256 rays/pixel, with sampling directions computed in the unit square and mapped to the sphere using our transform. With a fast ray tracer capable of 5 million rays/second on a single core, the total rendering time for a 1 megapixel image would be about 61.0 seconds, using the standard scalar transform. With the SIMD version, the rendering time goes down to 52.7 seconds, a 13.5% improvement.

N	scalar standard	scalar optimized	SSE optimized
256	114.7	106.9	23.0
4k	145.1	115.8	23.2
64k	147.8	115.7	23.2
1M	149.1	117.7	25.6
16M	149.0	117.5	25.6

Table 4. Execution times for the inverse transform, i.e., sphere-to-square. The timings are reported as number of clock cycles per transformed vector.

6. Proof of Area Preservation

Here, we present a formal proof that the described mapping, $P : (u, v) \rightarrow (x, y, z)$, from the square to the sphere, indeed preserves fractional area. The

magnitude of the vector product of the partial derivatives of P with respect to u and v gives the area-distortion, dA, of the transform:

$$dA = \left\| \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v} \right\| = \left\| \begin{array}{c} y_u z_v - z_u y_v \\ z_u x_v - x_u z_v \\ x_u y_v - y_u x_v \end{array} \right\|,$$

where we have used the shorthand notation x_u for $\partial x/\partial u$, and so on. We consider the inner triangle of the first quadrant in the square $(u, v) \in [-1, 1]$ (see Figure 3) and expand the expressions for the partial derivatives of (x, y, z) with respect to u and v. Due to symmetry, the same proof applies to the other parts of the map. Using the chain rule, and noting that $r_u = r_v = 1$ (as $r = u + v$), we get

$$\begin{array}{rclcl} x_u & = & x_r r_u + x_\phi \phi_u & = & x_r + x_\phi \phi_u, \\ x_v & = & x_r r_v + x_\phi \phi_v & = & x_r + x_\phi \phi_v, \end{array} \tag{8}$$

with similar expressions for the y and z components. Combining Equation (6) and (8) and simplifying, we obtain

$$dA = \left\| \begin{array}{c} (\phi_v - \phi_u)(y_r z_\phi - z_r y_\phi) \\ (\phi_v - \phi_u)(z_r x_\phi - x_r z_\phi) \\ (\phi_v - \phi_u)(x_r y_\phi - y_r x_\phi) \end{array} \right\|. \tag{9}$$

Further, the partial derivatives of ϕ (Equation (2)) can be written as

$$\phi_u = -\frac{\pi}{2} \frac{v}{(u+v)^2} \qquad \text{and} \qquad \phi_v = \frac{\pi}{2} \frac{u}{(u+v)^2},$$

which gives

$$\phi_v - \phi_u = \frac{\pi}{2} \frac{1}{u+v} = \frac{\pi}{2r}. \tag{10}$$

Now, we are ready to write the expressions for the partial derivatives of (x, y, z) with respect to r and ϕ:

$$\begin{array}{rclcl} x_r & = & \frac{2(1-r^2)}{\sqrt{2-r^2}} \cos\phi & \qquad x_\phi & = & -r\sqrt{2-r^2} \sin\phi, \\ y_r & = & \frac{2(1-r^2)}{\sqrt{2-r^2}} \sin\phi & \qquad y_\phi & = & r\sqrt{2-r^2} \cos\phi, \\ z_r & = & -2r, & \qquad z_\phi & = & 0. \end{array} \tag{11}$$

Applying Equation (10) and (11), Equation (9) reduces to

$$dA \;=\; \pi \left\| \begin{array}{c} y_\phi \\ -x_\phi \\ 1 - r^2 \end{array} \right\| \;=\; \pi \sqrt{y_\phi^2 + (-x_\phi)^2 + (1-r^2)^2} \;=\; \ldots \;=\; \pi.$$

Hence, the fractional area grows by a factor of π when we go from the square $(u, v) \in [-1, 1]$ to the sphere $\mathcal{S}$. This is consistent with our expectations as the area of the square is 4, and the surface area of the unit sphere is 4π.

Acknowledgments. The author was supported by the Swedish Foundation for Strategic Research and by Intel Corporation. Thanks also to Tomas Akenine-Möller and Jacob Munkberg for proofreading and many valuable suggestions, and to Jacob for help with Figure 4.

References

[Clarberg and Akenine-Möller 08] Petrik Clarberg and Tomas Akenine-Möller. "Practical Product Importance Sampling for Direct Illumination." In *Proceedings of Eurographics*, pp. 681–690. Aire-la-Ville, Switzerland: Eurographics Association, 2008.

[Fraser 65] W. Fraser. "A Survey of Methods of Computing Minimax and Near-Minimax Polynomial Approximations for Functions of a Single Independent Variable." *Journal of the ACM* 12:3 (1965), 295–314.

[Hart 78] J. F. Hart. *Computer Approximations*. Malabar, FL: Krieger Pub. Co., 1978.

[Kajiya 86] James T. Kajiya. "The Rendering Equation." *Proceedings SIGGRAPH '86, Computer Graphics* 20:4 (1986), 143–150.

[Praun and Hoppe 03] Emil Praun and Hugues Hoppe. "Spherical Parametrization and Remeshing." *ACM Transactions on Graphics* 22:3 (2003), 340–349.

[Shirley and Chiu 97] Peter Shirley and Kenneth Chiu. "A Low Distortion Map between Disk and Square." *journal of graphics tools* 2:3 (1997), 45–52.

Web Information:

Source code is available at http://jgt.akpeters.com/Clarberg08/.

Petrik Clarberg, Department of Computer Science, Lund University, Box 118, SE-22100 Lund, Sweden (petrik@cs.lth.se)

Received April 24, 2008; accepted November 7, 2008.