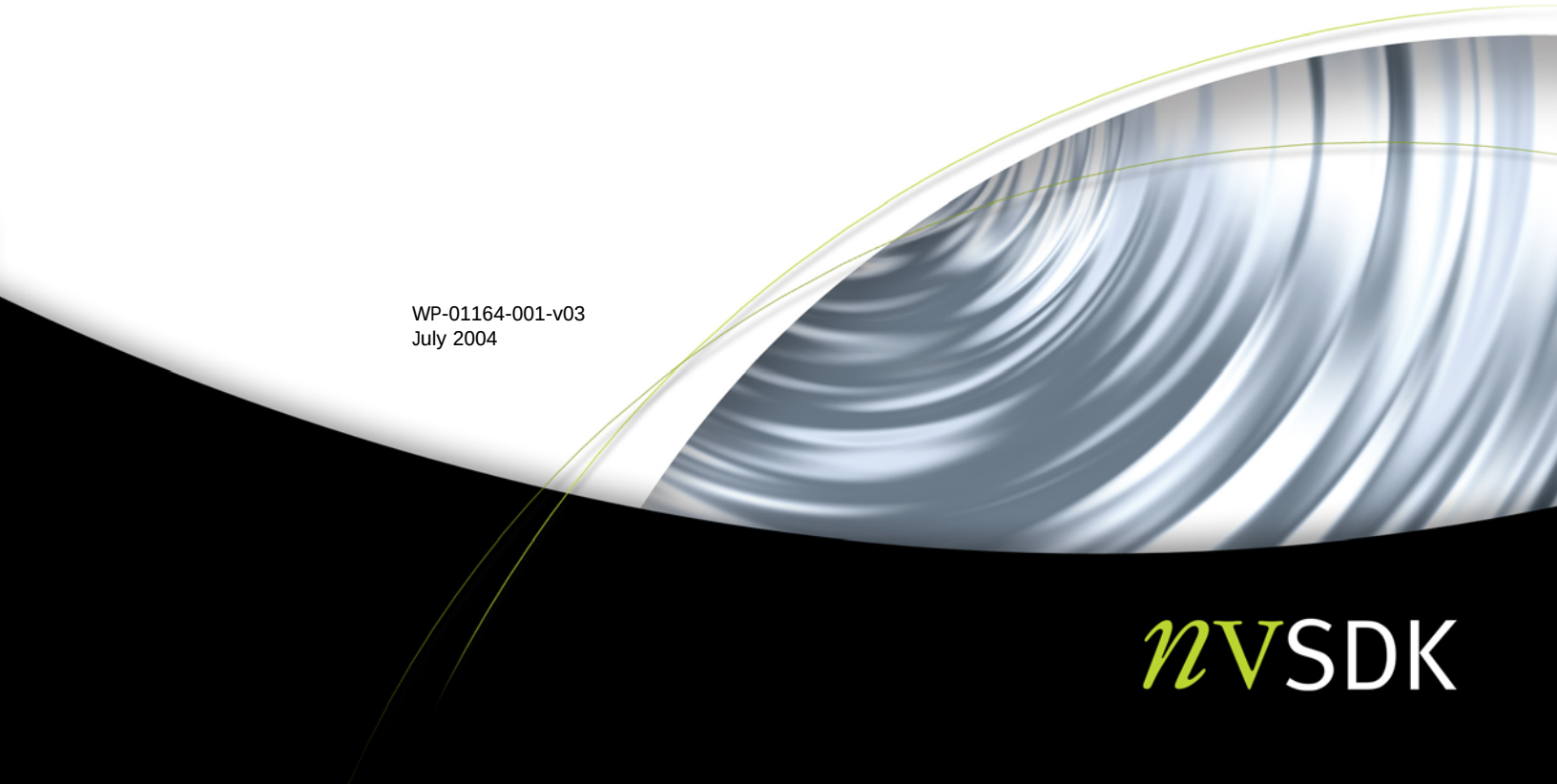
SDK White Paper

Normalization Heuristics

Performance vs. Quality on GeForce FX

WP-01164-001-v03
July 2004



nVSDK

Abstract

About Normalization Heuristics

This white paper answers the question, "When is cube-map normalization faster than **normalize()**?" It describes experiments performed with a non-trivial pixel shader, and uses the experimental results to derive useful rules of thumb regarding the performance and quality of normalization in pixel shaders. These heuristics provide *tuning dials* that developers can use to trade quality for performance, and vice versa, in 3D applications. To gain an intuitive understanding of these performance-quality tradeoffs, a demonstration application is provided so that the user has access to the experiments described in this white paper.

Mark J. Harris
mharris@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

Discussion

NVIDIA often encourages developers to use cube-map normalization in pixel shaders on GeForce FX. Although this is often faster than the standard **`normalize()`** pixel shader function, there are cases where it is not. This white paper addresses the question of when to choose cube-map normalization over **`normalize()`** and vice versa for faster performance. With the help of experiments, this paper presents two heuristics—one based on performance and one on quality—for the use of normalization cube maps.

Consider a non-trivial per-pixel lighting *skin* shader with tangent-space bump mapping as shown in **Error! Reference source not found.**. This pixel shader is described later on in Appendix A and consists of the following four **normalize** operations:

1. Normalize the eye space position of the fragment. This gives the **`V`** vector.
2. Normalize the eye space light vector, **`L`**.
3. Compute the half angle vector, **`H = normalize(V + L)`**.
4. Normalize the normal vector, **`N`** after it is scaled and transformed into eye space.

These normalizations are all essential[1]. You cannot compute **`H`** from non-normalized vectors because if **`V`** and **`L`** have different lengths, then their sum has a different direction than the sum of the normalized vectors[2].

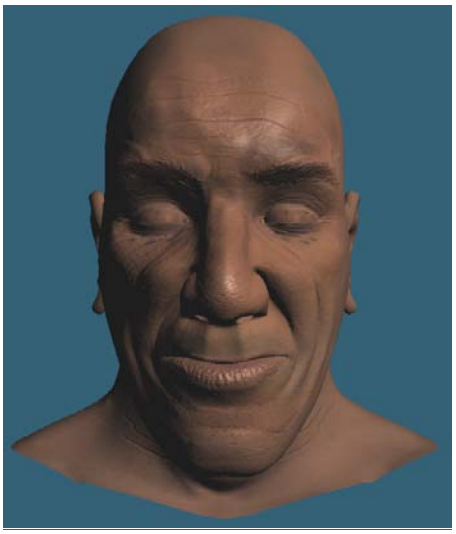


Figure 1. Model, Shader, and View Used in Experiments

The standard **`normalize()`** pixel shader function compiles to three instructions on GeForce FX GPUs: **`dp3`**, **`rsq`**, and **`mul`** which requires multiple cycles. The goal is to determine if and when a cube map lookup is faster but not at the expense of quality; indiscriminate use of cube-map normalization can result in shading artifacts.

[1] See Appendices B and C for a discussion of situations where normalization may not be necessary.

[2] Even with the optimization at the end of Appendix B, this shader requires four square roots.

Exhaustive tests of different configurations of the pixel shader are performed. The first test performs all normalization using **`normalize();`** the last test performs all normalization using cube map lookups with the remaining 14 tests covering other combinations of **`normalize()`** and cube map lookups. A single 8-bit RGBA normalization cube map with face resolution of 256x256 is used. A discussion of how cube-map normalization affects rendering quality follows.

Performance

You might assume that the slowest case would be four calls to **`normalize()`**, and the fastest to be four cube map lookups. These are both incorrect assumptions! Table 1 shows performance for all 16 combinations of normalization methods on three GPUs: GeForce FX 5200, 5700, and 5950. The leftmost four columns indicate which of **V**, **L**, **H**, and **N** are normalized with a cube map in each test.

Table 1. Performance Comparison for all Combinations of Normalize() and Normalization via Cube Map Lookups

Active Cube Maps					Performance (FPS)			
N	H	L	V	# inst	FX 5950	FX 5700	FX 5200	Comment
				34	132.5	61.4	17.7	No cube maps
			●	33	132.3	61.4	18.6	
		●		32	137.3	63.4	18.2	
		●	●	31	146.8	68.2	21.6	Speed / quality sweet spot
	●			32	138.6	68.0	19.3	
	●		●	31	145.2	71.1	21.5	
	●	●		30	142.0	66.1	19.9	
	●	●	●	29	155.9	75.9	23.4	Highest performance
●				32	119.3	57.9	18.2	Lowest performance
●			●	31	121.4	60.3	18.9	
●		●		30	118.3	57.6	18.0	
●		●	●	29	130.0	63.9	20.6	
●	●			30	129.3	60.5	17.9	
●	●		●	29	133.3	62.9	19.3	
●	●	●		28	134.2	63.3	19.5	
●	●	●	●	27	148.3	68.8	22.1	All cube maps

Table 1 yields the following observations:

1. The shortest shader is not the fastest, and the longest shader is not the slowest
2. Performance when using a cube map to normalize **N** is always lower than the corresponding case using **`normalize()`.**

The second observation is important; the cause is texture cache incoherence. The cube map is used to normalize the normal obtained from the normal map, which contains detailed regions over which the surface normal varies rapidly. This causes large strides across the normalization cube map, resulting in many texture cache misses. You can be verify this by reducing the resolution of the cube map textures. With a cube map composed of 1x1 textures, cube-map normalization always increases performance over **`normalize()`**. However, low-resolution cube maps can cause blocky lighting.

The fastest shader uses normalization cube maps for all vectors except **N**. This is because the first three vectors vary smoothly (because the polygonal surface is not as rough as the normal map), meaning that the lookups have good spatial locality in the cube map texture. This results in more texture cache hits and better performance. This leads to the first normalization heuristic.

Normalization Performance Heuristic: If a vector to be normalized varies smoothly, it is generally faster to use cube-map normalization than to call **`normalize()`**. If the vector varies rapidly, **`normalize()`** is likely to be faster.

Quality

If all four normalization cube maps are used, there are noticeable artifacts. There are two sources of these artifacts. The first is precision. The 8-bit RGBA textures used in the cube maps result in only 256 different values for each normal component. This results in visible bands in smooth gradients in the lighting[3]. This is most noticeable in specular highlights. The second problem is the resolution of the cube map. A lower resolution cube map tends to be faster because more of it can fit in the texture cache. However, lower resolution maps represent fewer vectors resulting in blocky artifacts in the lighting. These artifacts are most noticeable in variable, high frequency lighting, such as view-dependent specular highlights. In the shader used here, these artifacts show when a cube map is used to normalize the halfway vector, **H**, as demonstrated by the picture on the right in Figure 2.

Normalization Quality Heuristic: If a vector to be normalized is used for view-dependent or time-varying lighting or other effects, low-precision cube-map normalization is likely to result in visible artifacts.

[3] It is possible to use cube maps that have two 16-bit channels for higher-precision normalization. Unfortunately, this requires two cube map lookups per normalization: one for x and y, the other for z. See Table 6 and Table 7.

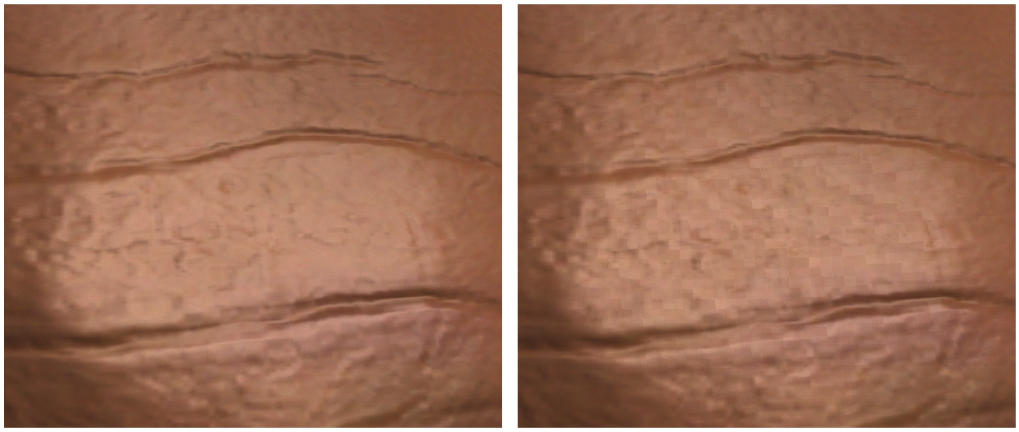


Figure 2. Artifacts (right) caused by cube-map normalization of **H**

Mipmapping

The strong dependence of performance on texture cache coherence leads to Mipmapping as another option for balancing performance and quality. Mipmapping is a practice commonly used to reduce texture aliasing, but its benefits extend to performance, too. Texture minification without mipmapping leads to samples falling far apart in the texture and poor cache coherence which in turn means reduced performance. Textures that will undergo minification should almost always be mipmapped.

This leads to the question, "Why not mipmap the normalization cube maps?" They will undergo minification as well, and you want to avoid texture cache thrashing. Before you pursue this approach, however, you should think about the purpose of the cube map. The goal is to replace an expensive **`normalize()`**computation with a cheaper texture access, while keeping the error in the normalization to a minimum. As mentioned in the previous section lower-resolution cube maps can represent fewer vectors because the sphere of directions is sampled in fewer directions. When you use mipmaps, you are effectively using a lower-resolution cube map in areas of texture minification. Neighboring vectors in lower-resolution maps will point in significantly different directions. Bilinear filtering of these vectors can result in significant denormalization (shortening) of the vectors resulting in vectors that were intended to be normalized not being normalized at all.

Nonetheless, that makes for an interesting experiment. Using the same shader, you can normalize **N**, **L**, and **V** using cube maps, because this configuration maintains high rendering quality, and has a lot to gain from mipmapping due to the poor texture cache coherence of normalizing **N**. The results are shown in Table 2.

Table 2. Performance Comparison using Mipmapped Normalization Cube Maps for **N**, **L**, and **V** Vectors

Max Mip Level	Performance (FPS)		
	FX 5950	FX 5700	FX 5200
0	130	61.8	20.6
1	143.3	64.5	21.6
2	145.8	66	22.3
3	152.2	67	22.8
4	152.2	67.7	23
5	152.2	67.9	23
6	152.2	68	23
7	152.2	68	23
8	152.2	68	23

The next experiment enables mipmapping on the normalization cube map, but the maximum mip level accessed[4] is limited. Table 2 shows that performance increases up to 17 percent as the maximum mip level is raised, but only up to a point. Beyond level 3, the performance gain is negligible. Equally of interest is that in this experiment, most of the visible error introduced by mipmapping occurs after level 3. Figure 3 shows the results of error introduced by mipmapping on a model viewed at a distance, so that it projects to an area of roughly 50 pixels square. With a maximum mip level of 3 there is very little visible error, whereas, the error is quite noticeable at a mip level of 8.

This experiment shows that you can use mipmapping to squeeze more performance out of normalization cube maps, as long as you are careful to constrain the error by limiting the maximum mipmap level. Table 3 is a revision of Table 1, with the normalization cube map mipmapped to a maximum level of 3.

4 The maximum mipmap level is controlled in OpenGL by setting the GL_TEXTURE_MAX_LOD texture parameter, and in DirectX by setting the D3DSAMP_MAXMIPLEVEL sampler state.

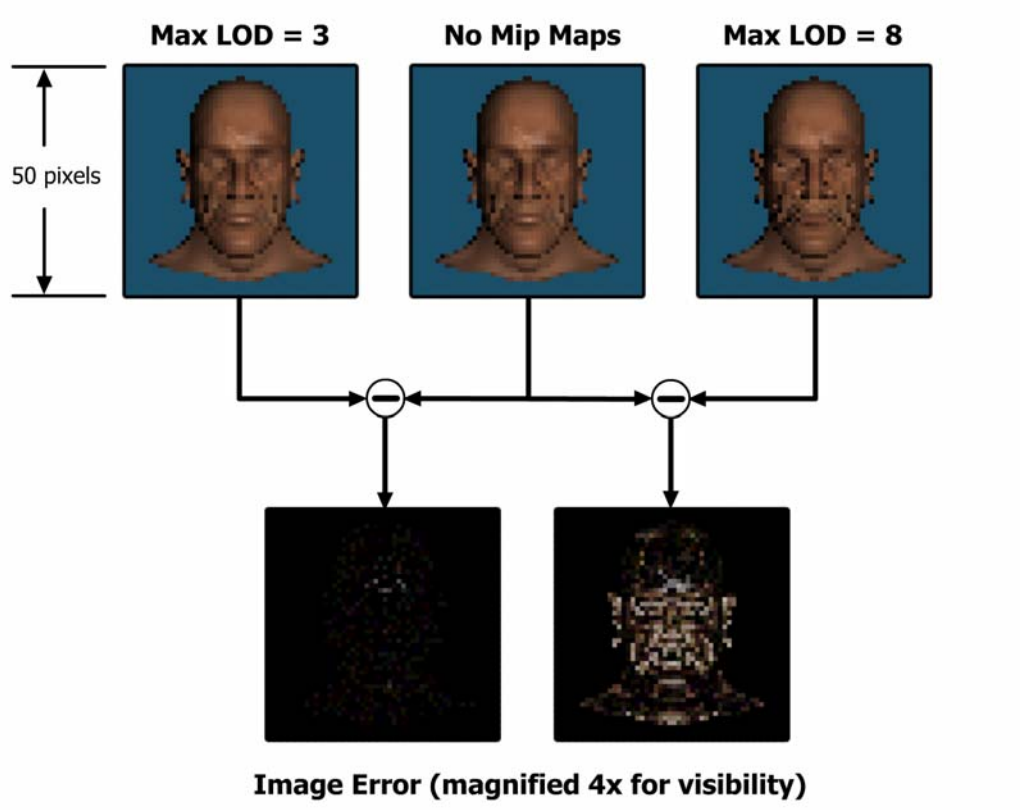


Figure 3. Comparison of Error Induced by using Maximum Mipmap Levels of 3 (bottom left) and 8 (bottom right)

The Speed/Quality Sweet Spot

The best quality is obtained by using **`normalize()`** for all normalization. These experiments show that using cube maps to normalize the light, view, and normal vectors results in very little visible difference. The half angle vector **H** on the other hand, directly affects specular lighting. Artifacts are noticeable due to the higher frequency and view dependence of specular lighting. Performance when using cube maps for only the view and light vectors is nearly as good as when using them for view, light, and half angle vectors, so this is a good balance between performance and quality. You can get even higher performance by also normalizing **N** with a mipmapped cube map, but you must be careful to clamp the maximum mipmap level used to avoid noticeable artifacts.

A working application with OpenGL source code is available at **http://developer.nvidia.com**. Appendix A details the Cg shader source code used in these experiments. Appendices B and C provide additional information about normalization. Appendix D contains detailed tables of all results obtained from these experiments, including a 16-bit HILO cube map experiment not described here.

Table 3 Performance Comparison for all Combinations of normalize() and Normalization via Mipmapped Cube Map Lookups

Active Cube Maps					Performance (FPS)			
N	H	L	V	# inst	FX 5950	FX 5700	FX 5200	Comment
				34	132.5	61.4	17.7	Lowest performance
			●	33	132.3	61.4	18.6	
		●		32	137.3	63.4	18.2	
		●	●	31	146.8	68.2	21.6	
	●			32	138.6	68	19.3	
	●		●	31	145.2	71.1	21.5	
	●	●		30	142	66.1	19.9	
	●	●	●	29	155.9	75.9	23.4	
●				32	139.3	63.5	19.9	
●			●	31	142.3	66.4	20.8	
●		●		30	139	63.8	19.7	
●		●	●	29	152.2	70.9	22.8	Speed / quality sweet spot
●	●			30	105.9	66.9	19.6	
●	●		●	29	150.3	69.2	21	
●	●	●		28	152.5	70.3	21.4	
●	●	●	●	27	172.9	76.8	24.5	Highest Performance

Note: The maximum mipmap level is constrained to 3 (32x32).

Appendix A
Cg / HLSL Shader Source

```
// Note: this code depends on the use of signed RGB textures
// for normalization cube maps.  These are available on
// NVIDIA GeForce 3 and higher GPUs, through the GL_SIGNED_RGB_NV
// texture internal format (The DirectX equivalent is
// D3DFMT_Q8W8V8U8). If unsigned textures are used, care must be
taken
// to range expand the vectors obtained from the cube map lookups:
// vec = 2 * h3texCUBE() – 1.

struct fragin
{
    half2 texcoords            : TEXCOORD0;
    half4 shadowcoords         : TEXCOORD1;
    half4 tangentToEyeMat0     : TEXCOORD4;
    half3 tangentToEyeMat1     : TEXCOORD5;
    half3 tangentToEyeMat2     : TEXCOORD6;
    half3 eyeSpacePosition     : TEXCOORD7;
};

half4 main(fragin In,
   uniform sampler2D normalTexture,
   uniform sampler2D diffuseTexture,
   uniform sampler2D glossyTexture,
   uniform samplerCUBE normCubeTexture,
   uniform half3 eyeSpaceLightPosition) : COLOR
{
    // diffuse and specular colors
    half4 kd = h4tex2D(diffuseTexture, In.texcoords);
    half4 ks = h4tex2D(glossyTexture, In.texcoords);

    half3 n,h,l,v;

    // Get eye-space eye vector.
#ifdef CUBEMAP_V
    v = h3texCUBE(normCubeTexture, -In.eyeSpacePosition);
#else
    v = normalize(-In.eyeSpacePosition);
#endif

    // Get eye-space light and halfangle vectors.
#ifdef CUBEMAP_L
    l = h3texCUBE(normCubeTexture,
                          eyeSpaceLightPosition -
In.eyeSpacePosition);
#else
    l = normalize(eyeSpaceLightPosition - In.eyeSpacePosition);
```

```
#endif

#ifdef CUBEMAP_H
    h = h3texCUBE(normCubeTexture, v + l);
#else
    h = normalize(v + l);
#endif

    // Get tangent-space normal vector from normal map.
    half3 bumpScale = {In.tangentToEyeMat0.ww, 1};
    half3 tangentSpaceNormal = bumpScale *
 h3tex2D(normalTexture, In.texcoords);

    // Transform it into eye-space.
    n.x = dot(In.tangentToEyeMat0.xyz, tangentSpaceNormal);
    n.y = dot(In.tangentToEyeMat1, tangentSpaceNormal);
    n.z = dot(In.tangentToEyeMat2, tangentSpaceNormal);

#ifdef CUBEMAP_N
    n = h3texCUBE(normCubeTexture, n);
#else
    n = normalize(n);
#endif

    static const half m = 34; // specular exponent
    half4 coeffs;
    coeffs.y = dot(n,l);
    coeffs.z = dot(n,h);
    coeffs = lit(coeffs.y, coeffs.z, m);

    // Compute lighting.
    return coeffs.y * kd + coeffs.z * ks;
}
```

Appendix B
When to Normalize

Depending on the situation, normalization may not always be necessary. In the lighting example given in this white paper, all four normalizations are essential. Because the vectors are used for lighting, they must have unit length or the results of the lighting computation will be incorrect. In the case of environment mapping, however, normalization is sometimes overused. The formula typically given for the reflection vector is:

R = 2 * dot(N,V) * N – V,

where **N** is the unit-length normal vector, and **V** is the vector from the viewpoint to the reflection point.

Textbooks sometimes claim that **V** must be normalized, too. In the common case of hardware cube map reflection, this is not true; texture coordinates for cube map lookups can represent any three-dimensional vector. Only **N** need be unit length to get the correct reflected value. In the case of a non-unit-length **N**, you can use another formulation of reflection:

R = 2 * dot(N,V) * N – dot(N,N) * V.

This formulation results in an **R** with the correct direction, regardless of the lengths of **N** and **V**. Note, however that **R** and **V** do not necessarily have the same length. If preserving the length of **V** is necessary, then the following formulation can be used.

R = (2 * dot(N,V) * N) / dot(N,N) – V.

Dot Product Optimization

In lighting computations, vectors are typically used to compute dot products. There is a trick that can sometimes be used to reduce the computational cost of normalizing vectors. To compute diffuse lighting you must compute the dot product of the unit-length normal and light vectors. This typically requires two reciprocal square roots instructions:

dot(N / ||N||,L / ||L||) = dot(N * rsq(dot(N,N)), L * rsq(dot(L,L))).

However, you can reduce this to a single reciprocal square root because

dot(N / ||N||, L / ||L||) = dot(N,L) / (||N|| * ||L||).

Thus, a more efficient computation is

dot(N/||N||, L/||L||) = dot(N,L) * rsq(dot(N,N) * dot(L,L)).

Appendix C
Approximate Normalization

A useful optimization for normalization is based on the fact that vectors to be interpolated are usually close to unit length. For a nearly-unit-length vector **V**, you can approximate `1 / ||V||` by the first terms of the Taylor expansion of **1 / sqrt(x) at x = 1:**

$$1 / \mathrm{sqrt}(x) \approx 1 + (1 - x) / 2.$$

The approximation for **V** is therefore

$$V / ||V|| = V / \mathrm{sqrt}(||V||^2) \approx V + V * (1 - ||V||^2) / 2.$$

This computation can be implemented using the following two assembly instructions.

```
dp3_sat r1, r0, r0
mad_d2 r1, r0, 1-r1, r0_d2
```

Appendix D
Results

Table 4 contains the complete performance comparison for all configurations with signed RGB cubemaps, float (32-bit) registers, with and without mipmaps. In the mipmap case, the maximum mipmap level is set to 3

Table 4. Signed RGB Cube Maps, 32-bit Float Registers

Active Cubemaps				# inst	# R regs	FX 5950 Performance		FX 5700 Performance		FX 5200 Performance	
						no mipmaps	mipmaps*	no mipmaps	mipmaps*	no mipmaps	mipmaps*
N	H	L	V								
0	0	0	0	34	4	94.2 fps	94.2 fps	44.1 fps	44.1 fps	17.9 fps	17.9 fps
0	0	0	1	33	3	97.1 fps	97.1 fps	45.6 fps	45.6 fps	19.1 fps	19.1 fps
0	0	1	0	32	5	100 fps	100 fps	47.9 fps	47.9 fps	19.1 fps	19.1 fps
0	0	1	1	31	4	107.3 fps	107.3 fps	50.5 fps	50.5 fps	20.7 fps	20.7 fps
0	1	0	0	32	5	98.3 fps	98.3 fps	46.5 fps	46.5 fps	19.1 fps	19.1 fps
0	1	0	1	31	4	103.1 fps	103.1 fps	48.9 fps	48.9 fps	20.6 fps	20.6 fps
0	1	1	0	30	5	103.7 fps	103.7 fps	49 fps	49 fps	21.2 fps	21.2 fps
0	1	1	1	29	5	112.8 fps	112.8 fps	53.7 fps	53.7 fps	22.2 fps	22.2 fps
1	0	0	0	32	5	87.6 fps	98.2 fps	42.5 fps	45.8 fps	17.2 fps	18.7 fps
1	0	0	1	31	4	93.5 fps	104.4 fps	45.5 fps	49.8 fps	18.2 fps	19.9 fps
1	0	1	0	30	6	94.2 fps	103.2 fps	44 fps	48 fps	18.3 fps	20.1 fps
1	0	1	1	29	5	97.6 fps	110.5 fps	47.8 fps	52.6 fps	19.8 fps	21.9 fps
1	1	0	0	30	5	90.5 fps	102.2 fps	44.2 fps	47.7 fps	18.4 fps	20.3 fps
1	1	0	1	29	5	97.7 fps	107.7 fps	47.6 fps	51.5 fps	19.6 fps	21.7 fps
1	1	1	0	28	5	96.5 fps	108.4 fps	47 fps	51.1 fps	20.3 fps	22.3 fps
1	1	1	1	27	5	103.8 fps	116.5 fps	51.3 fps	55.6 fps	21.1 fps	23.3 fps

Table 5 contains the complete performance comparison for all configurations with signed RGB cubemaps, half (16-bit) registers, with and without mipmaps. In the mipmap case, the maximum mipmap level is set to 3.

Table 5. Signed RGB Cube Maps 16-bit Half Registers

Active Cubemaps				# inst	# R regs	FX 5950 Performance		FX 5700 Performance		FX 5200 Performance	
						no mipmaps	mipmaps*	no mipmaps	mipmaps*	no mipmaps	mipmaps*
N	H	L	V								
0	0	0	0	34	4	132.5 fps	132.5 fps	61.4 fps	61.4 fps	17.7 fps	17.7 fps
0	0	0	1	33	3	132.3 fps	132.3 fps	61.4 fps	61.4 fps	18.6 fps	18.6 fps
0	0	1	0	32	5	137.3 fps	137.3 fps	63.4 fps	63.4 fps	18.2 fps	18.2 fps
0	0	1	1	31	4	146.8 fps	146.8 fps	68.2 fps	68.2 fps	21.6 fps	21.6 fps
0	1	0	0	32	5	138.6 fps	138.6 fps	68.0 fps	68.0 fps	19.3 fps	19.3 fps
0	1	0	1	31	4	145.2 fps	145.2 fps	71.1 fps	71.1 fps	21.5 fps	21.5 fps
0	1	1	0	30	5	142.0 fps	142 fps	66.1 fps	66.1 fps	19.9 fps	19.9 fps
0	1	1	1	29	5	155.9 fps	155.9 fps	75.9 fps	75.9 fps	23.4 fps	23.4 fps
1	0	0	0	32	5	119.3 fps	139.3 fps	57.9 fps	63.5 fps	18.2 fps	19.9 fps
1	0	0	1	31	4	121.4 fps	142.3 fps	60.3 fps	66.4 fps	18.9 fps	20.8 fps
1	0	1	0	30	6	118.3 fps	139 fps	57.6 fps	63.8 fps	18 fps	19.7 fps
1	0	1	1	29	5	130 fps	152.2 fps	63.9 fps	70.9 fps	20.6 fps	22.8 fps
1	1	0	0	30	5	129.3 fps	105.9 fps	60.5 fps	66.9 fps	17.9 fps	19.6 fps
1	1	0	1	29	5	133.3 fps	150.3 fps	62.9 fps	69.2 fps	19.3 fps	21 fps
1	1	1	0	28	5	134.2 fps	152.5 fps	63.3 fps	70.3 fps	19.5 fps	21.4 fps
1	1	1	1	27	5	148.3 fps	172.9 fps	68.8 fps	76.8 fps	22.1 fps	24.5 fps

Table 6 contains the complete performance comparison for all configurations with signed HILO cube maps, float (32-bit) registers, with and without mipmaps. In the mipmap case, the maximum mipmap level is set to 3.

Table 6. Signed HILO Cube Maps, 32-bit float registers

Active Cubemaps				# inst	# R regs	FX 5950 Performance		FX 5700 Performance		FX 5200 Performance	
						no mipmaps	mipmaps*	no mipmaps	mipmaps*	no mipmaps	mipmaps*
N	H	L	V								
0	0	0	0	34	4	94.2 fps	94.2 fps	44.1 fps	44.1 fps	17.9 fps	17.9 fps
0	0	0	1	35	5	97.7 fps	97.7 fps	47.4 fps	47.4 fps	18 fps	18 fps
0	0	1	0	34	5	92.8 fps	92.8 fps	45.7 fps	45.7 fps	18.3 fps	18.3 fps
0	0	1	1	35	5	92.9 fps	92.9 fps	43.7 fps	43.7 fps	18.4 fps	18.4 fps
0	1	0	0	34	5	98.6 fps	98.6 fps	42.7 fps	42.7 fps	18.3 fps	18.3 fps
0	1	0	1	35	5	96.3 fps	96.3 fps	43.5 fps	43.5 fps	18.5 fps	18.5 fps
0	1	1	0	34	5	88.5 fps	88.5 fps	40.2 fps	40.2 fps	18.6 fps	18.6 fps
0	1	1	1	35	5	89.2 fps	89.2 fps	44.1 fps	44.1 fps	18.7 fps	18.7 fps
1	0	0	0	34	5	71.2 fps	85.8 fps	36.9 fps	40.3 fps	15.3 fps	17.4 fps
1	0	0	1	36	5	75.1 fps	88.8 fps	37.8 fps	40.9 fps	15.4 fps	17.4 fps
1	0	1	0	34	5	68.9 fps	81.2 fps	38.2 fps	41.5 fps	15.5 fps	17.6 fps
1	0	1	1	36	5	74.1 fps	87.2 fps	37.4 fps	40.7 fps	16.0 fps	18.2 fps
1	1	0	0	34	5	70.5 fps	84.1 fps	37.8 fps	40.9 fps	15.7 fps	17.8 fps
1	1	0	1	37	5	65 fps	76.5 fps	39.4 fps	43.3 fps	15.9 fps	18.1 fps
1	1	1	0	34	5	69.8 fps	80.4 fps	34.9 fps	41.0 fps	15.8 fps	18 fps
1	1	1	1	37	5	72.3 fps	83.2 fps	41.3 fps	45.5 fps	16.3 fps	18.5 fps

Table 7 contains the complete performance comparison for all configurations with signed HILO cube maps, half (16-bit) registers, with and without mipmaps. In the mipmap case, the maximum mipmap level is set to 3.

Table 7. Signed HILO Cube Maps, 16-bit half registers

Active Cubemaps				# inst	# H regs	FX 5950 Performance		FX 5700 Performance		FX 5200 Performance	
						no mipmaps	mipmaps*	no mipmaps	mipmaps*	no mipmaps	mipmaps*
N	H	L	V								
0	0	0	0	34	4	132.5 fps	132.5 fps	61.4 fps	61.4 fps	17.7 fps	17.7 fps
0	0	0	1	35	5	126.6 fps	126.6 fps	56.7 fps	56.7 fps	17.7 fps	17.7 fps
0	0	1	0	34	5	130 fps	130 fps	59.7 fps	59.7 fps	17.6 fps	17.6 fps
0	0	1	1	35	5	129.3 fps	129.3 fps	61.1 fps	61.1 fps	19.3 fps	19.3 fps
0	1	0	0	34	5	133 fps	133 fps	59.6 fps	59.6 fps	19.2 fps	19.2 fps
0	1	0	1	35	5	128.8 fps	128.8 fps	58 fps	58 fps	17.3 fps	17.3 fps
0	1	1	0	34	5	135.7 fps	135.7 fps	58 fps	58 fps	17.7 fps	17.7 fps
0	1	1	1	35	5	128.2 fps	128.2 fps	59.3 fps	59.3 fps	19.7 fps	19.7 fps
1	0	0	0	34	5	90.5 fps	114.4 fps	52.1 fps	58.7 fps	15.3 fps	17.3 fps
1	0	0	1	36	5	100.1 fps	124.6 fps	48.1 fps	54 fps	15 fps	16.9 fps
1	0	1	0	34	5	101.7 fps	127.9 fps	49.4 fps	55.2 fps	15 fps	17 fps
1	0	1	1	36	5	86.9 fps	107.5 fps	49.1 fps	55 fps	16.7 fps	19.1 fps
1	1	0	0	34	5	94.4 fps	120 fps	50.8 fps	57 fps	16.6 fps	19 fps
1	1	0	1	37	5	88.5 fps	110.3 fps	51 fps	57.2 fps	15.9 fps	18 fps
1	1	1	0	34	5	91.3 fps	114.8 fps	44.4 fps	52.6 fps	16.5 fps	18.8 fps
1	1	1	1	37	5	95.2 fps	120.3 fps	48.8 fps	53.9 fps	16.2 fps	18.4 fps

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com