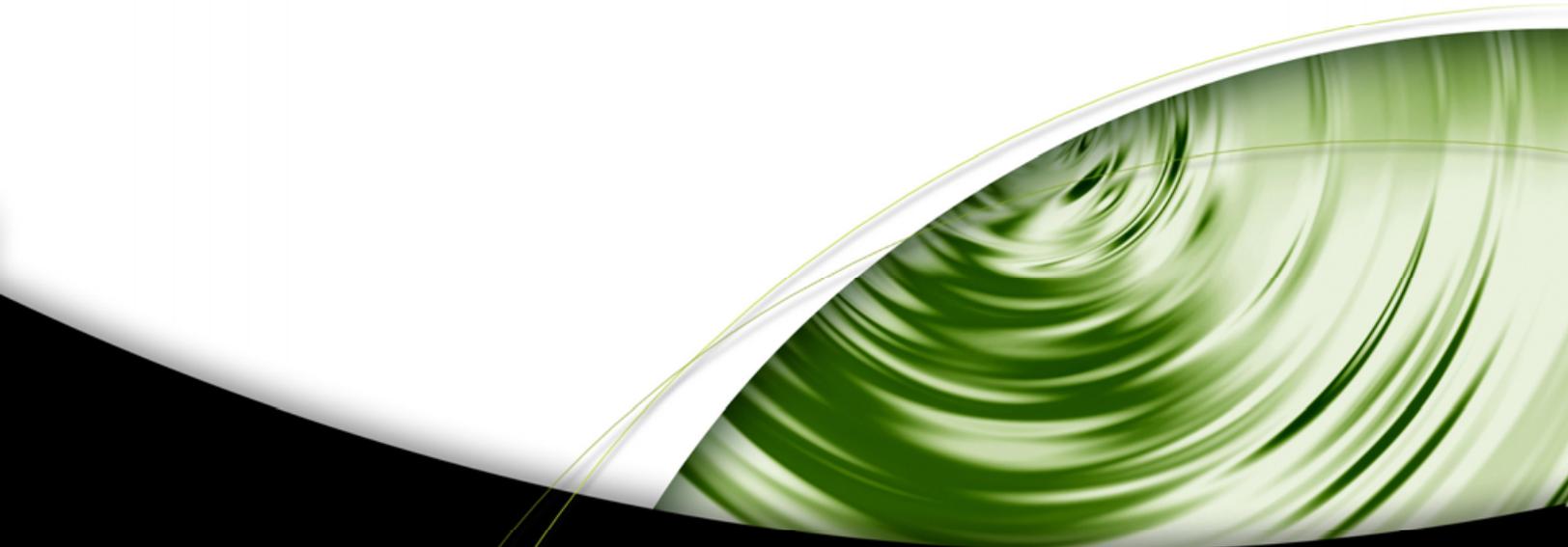




Technical Report

Non-Power-of-Two Mipmap Creation



DEVELOPMENT

A decorative graphic at the bottom of the page consists of a white, curved shape that appears to be peeling away from a black background, revealing a green, swirling, liquid-like texture underneath. The word "DEVELOPMENT" is printed in white, uppercase, sans-serif font on the black background at the bottom.

Abstract

Mipmapping is commonly used to avoid sampling artifacts during texture minification. The construction of mipmaps for textures whose size is a power-of-two is usually implemented with a box filter that computes the average of a 2x2 block of texels (texture pixels).

The GeForce 6 Series GPU family generalizes mipmapping to accommodate textures whose sizes are not powers-of-two. Here the mipmap construction is more complex. This document describes and gives code for a mipmap construction algorithm for non-power-of-two textures that yields good quality results. The box filter used for power-of-two textures is generalized into a so-called polyphase filter that changes its filter weights for each texel. Additionally, the support of the filter increases from a 2x2 block of texels to up to 3x3. When a texture happens to have power-of-two dimensions, the algorithm described reduces to the standard algorithm.

We recommend the use of the algorithm here for non-power-of-two mipmap construction because use of other algorithms, such as naïve bilinear interpolation, for example, can lead to poor image quality when texture mapping.

Stefan Guthe
sguthe@nvidia.com

Paul Heckbert
pheckbert@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

October 5, 2003

Mipmap Creation

Before generalizing mipmap creation to the non-power-of-two setting, let's first take a look at the commonly used algorithm. It involves averaging of 2 texels in the 1D case, 4 texels in the 2D case and 8 texels in the 3D case.

Power-of-Two Mipmaps

Creating a mipmap for a power-of-two 2D texture is usually done with this simple box filter algorithm.

```
// Initialize mipmap[0] with texture
...
// Compute mipmap levels
for (l=1; l<=maxlevel; l++)
    for (y=0; y<max(1,height>>1); y++) {
        float y0=y<<1;
        float y1=min(y0+1,max(1,height>>(l-1))-1);
        for (x=0; x<max(1,width>>1); x++) {
            float x0=x<<1;
            float x1=min(x0+1,max(1,width>>(l-1))-1);
            mipmap[l][y][x]=0.25*(mipmap[l-1][y0][x0]+mipmap[l-1][y0][x1]+
                                mipmap[l-1][y1][x0]+mipmap[l-1][y1][x1]);
        }
    }
```

It constructs an array of 2D images `mipmap[l]`, where level number `l` ranges from 0 at the finest level to $\log_2(\max(\text{width}, \text{height}))$ at the coarsest. The size of level `l` is $\max(1, \text{height} \gg l)$ by $\max(1, \text{width} \gg l)$.

See [Lance Williams, Pyramidal Parametrics, SIGGRAPH '83 Proceedings, July 1983].

Non-Power-of-Two Mipmaps

We describe the scheme in 1D first. With non-power-of-two textures, the widths of the mipmap levels are no longer powers of two, but can be any number. Let `wid[i]` denote the width of level `i`. Two rules for the sequence of level widths are:

- Rounding Down: $\text{wid}[i] = \text{floor}(\text{width} \gg i)$, $\text{maxlevel} = \text{floor}(\log_2(\text{width}))$

- Rounding Up: $wid[l] = \text{ceil}(\text{width} \gg l)$, $\text{maxlevel} = \text{ceil}(\log_2(\text{width}))$

Examples:

- base width=127, rounding down: 127, 63, 31, 15, 7, 3, 1
- base width=127, rounding up: 127, 64, 32, 16, 8, 4, 2, 1
- base width=128, rounding down: 128, 64, 32, 16, 8, 4, 2, 1
- base width=128, rounding up: 128, 64, 32, 16, 8, 4, 2, 1
- base width=129, rounding down: 129, 64, 32, 16, 8, 4, 2, 1
- base width=129, rounding up: 129, 65, 33, 17, 9, 5, 3, 2, 1

If the base width happens to be a power of two, rounding up and down yield identical results. In general, the scale factor between successive levels is no longer 2. Rounding up is generally preferable because less information is lost between levels. But current Direct3D and OpenGL APIs support rounding down only. See http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_non_power_of_two.txt.

For non-power-of-two textures, mipmap construction gets more complicated. First of all, we have to define some properties we would like the filter to have. The filter has to be

- symmetric, i.e., mirroring the texture will mirror all mipmap levels.
- energy conserving, i.e., the average of all mipmap levels has to be the same.
- a box filter of width 2 for each reduction by exactly a factor of 2 in any axis from a previous level.

Taking a look at the requirements, the simplest filter is a box filter with the width n'/n for filtering from texture width n' to n . For power-of-two textures or when n' is even, $n'/n=2$, but when rounding down with n' odd, the ratio is greater than 2, and when rounding up with n' odd, the ratio is less than 2. In order to get the correct weights for each texel in the previous mipmap level, we have to convolve this filter with a reconstruction filter, i.e. a box filter of width 1. The output texel is at the origin in Figure 1, whereas the texel of the previous mipmap level each have a distance of 1 to each other. The resulting “trapezoid filter” has a support of $\text{width}+1$.

You can also derive this filter by thinking of each pixel as being one pixel wide, and computing the fractional pixel areas covered by a sliding box of width ‘width’.

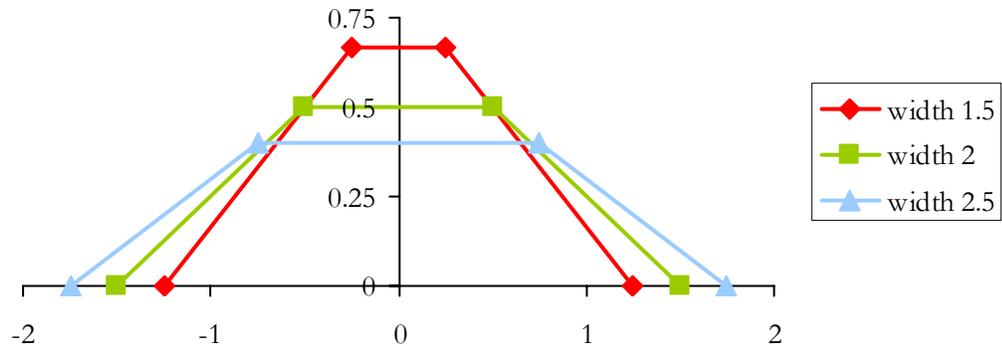


Figure 1. Trapezoid filters constructed by convolving different box filter widths with a box filter of width 1.

Simplifying the Polyphase Box Filter

Since the position of the filter is incremented by its width, the filter weights for the texel of the previous level change with x and y if the filter width is not an integer. Thus we have a polyphase filter. This is in contrast to the monophasic filter used for power-of-two mipmap construction, whose coefficients are the same at every pixel (in 1D, $\frac{1}{2}$ and $\frac{1}{2}$). To simplify our filter a bit we first take a look at the possible settings we will encounter going from n' to n .

- Rounding down: i.e. $n' = 2*n + 1$, filter width $(2*n + 1)/n$
- No rounding: i.e. $n' = 2*n$, filter width 2
- Rounding up: i.e. $n' = 2*n - 1$, filter width $(2*n - 1)/n$

For textures other than 1D textures, a combination of cases 1 and 2 or 2 and 3 will be encountered for creating each of the mipmap levels.

Rounding Down

For the rounding down setting, we observe the following behavior.

- For position x in the current mipmap level, we always sample $2*x$, $2*x + 1$ and $2*x + 2$ in the previous level.
- The first sample always ends up on the upward slope.
- The second sample is always in the flat center area.
- The third sample always ends up on the downward slope.
- The sample points move with a speed of $-1/n$ along the filter kernel.

Thus we can simplify the polyphase filter for this setting to the following algorithm in the 1D setting.

```

...
for (x=0; x<n; x++) {
    w0=(float)(n-x)/(2*n+1);
    w1=(float) n / (2*n+1);
    w2=(float)(1+x)/(2*n+1);
    mipmap[l][x]=w0*mipmap[l-1][2*x ]+w1*mipmap[l-1][2*x+1]+
                w2*mipmap[l-1][2*x+2];
}
...

```

No Rounding

The no rounding setting is even simpler, since sampling a width 2 trapezoid at -0.5 and 0.5 yields a monophasic filter equivalent to a width 2 box filter. It can therefore be written as.

```

...
for (x=0; x<n; x++) {
    mipmap[l][x]=0.5*(mipmap[l-1][2*x ]+mipmap[l-1][2*x+1]);
}
...

```

Rounding Up

The behavior for the rounding up setting is very similar to the rounding down setting.

- For position x in the current mipmap level, we always sample $2*x-1$, $2*x$ and $2*x+1$ in the previous level.
- The first sample always ends up on the upward slope.
- The second sample is always in the flat center area.
- The third sample always ends up on the downward slope.
- The sample points move with a speed of $1/n$ along the filter kernel.

Thus we can simplify the polyphase filter for this setting to the following algorithm in the 1D setting.

```

...
for (x=0; x<n; x++) {
    w0=(float)(n-x-1)/(2*n-1);
    w1=(float) n / (2*n-1);
    w2=(float) x / (2*n-1);
    mipmap[l][x]=w0*mipmap[l-1][2*x-1]+w1*mipmap[l-1][2*x ]+
                w2*mipmap[l-1][2*x+1];
}
...

```

When generalizing to 2D, the formulas require slight modification to allow for non-square textures.

- Rounding Down: $\text{wid}[l] = \max(1, \text{floor}(\text{width} >> l))$
 $\text{hei}[l] = \max(1, \text{floor}(\text{height} >> l))$
 $\text{maxlevel} = \text{floor}(\log_2(\max(\text{width}, \text{height})))$
- Rounding Up: similar, replacing 'floor' with 'ceil'

With rounding down, for example, the sequence of level sizes for a 31x10 texture is: 31x10, 15x5, 7x2, 3x1, 1x1.

Using the 2D variant of this simplified polyphase box filter, we get a mipmap as seen in Figure 2.

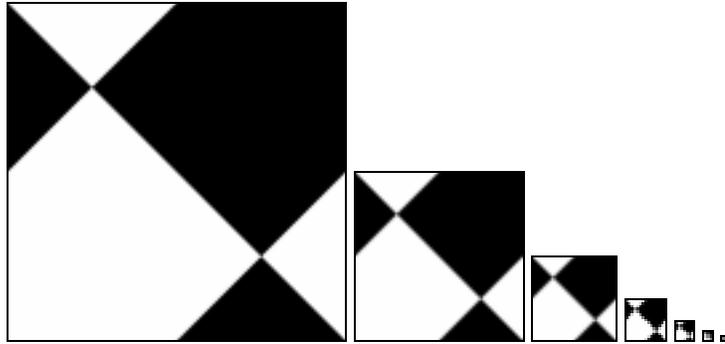


Figure 2. Mipmap for a 127^2 checkerboard texture and rounding down setting.

For a better evaluation of the quality of the mipmap creation algorithm we compare an image (Figure 3) rendered with a power-of-two mipmap (128^2) against an image (Figure 4) rendered with a non-power-of-two mipmap (127^2). This size, a power of two minus one, was chosen because it is the most difficult test case for rounding down.

While there is little difference in the images rendered with anisotropic filtering, the isotropic filtering is blurrier near the horizon for the non-power-of-two case. This is because the hardware does level of detail (LOD) selection using $\log_2(\text{footprint size})$, which assumes a factor of 2 between successive levels, but the mipmap deviates from this rule. The hardware's LOD formula assumes that the region in level 0 corresponding to a texel at level l has width 2^l , but in a non-power-of-two texture, that width is actually $\text{wid}[0]/\text{wid}[l]$. With rounding down, that width is greater than 2^l , so the image is blurred more than desired. This blurring is worst when the texture size is a power of two minus one. There is no new blurring when the size is a power of two, because our algorithm reduces to the standard power-of-two mipmap construction algorithm in that case.

Although this blurring is undesirable, it is the best one can do when rounding down. A naïve alternative algorithm would use standard bilinear interpolation to construct successive mipmap levels, but that would result in dropped details because of gaps between the triangle filters, and that would be far more objectionable than blurring.

If API's supported rounding up, superior results would be achievable, nearly as good as those for power-of-two textures, if overlapping filters of fixed width were used from level to level. A complication of this approach is that the texture wrap mode (repeat, mirror, clamp, etc) would need to be known during mipmap construction. We urge Direct3D and OpenGL to adopt rounding up as their standards so that non-power-of-two mipmaps will no longer be handicapped by blurring. For now, we must live with it, however.

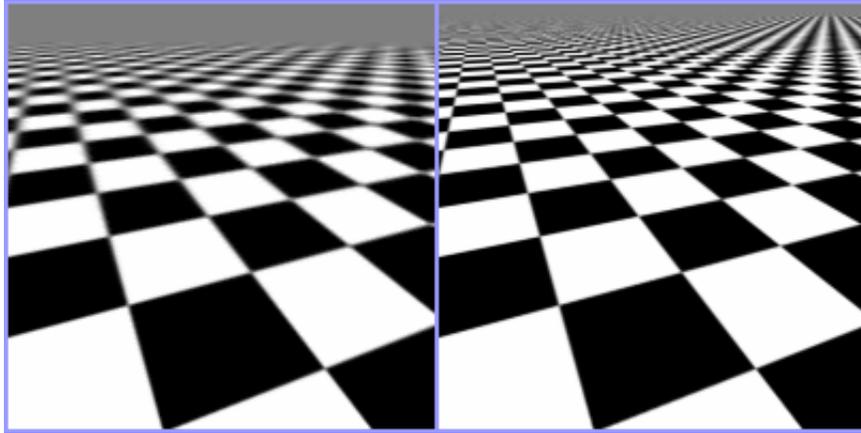


Figure 3. Image rendered using a power-of-two mipmap (128^2) with isotropic (left) and anisotropic filtering (right).

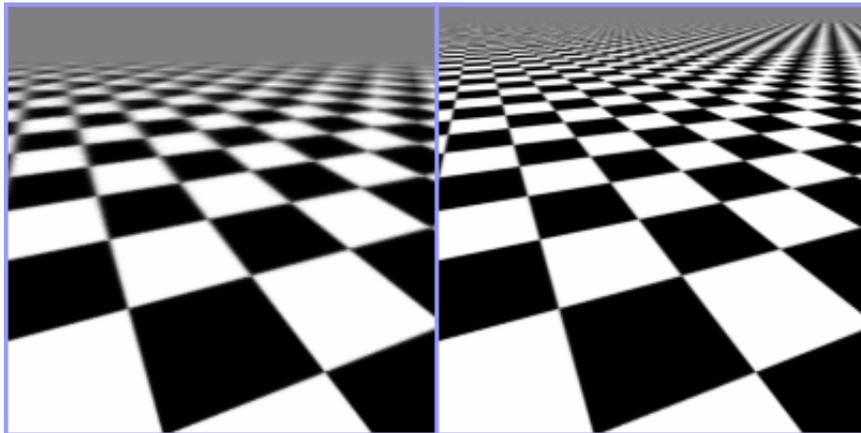


Figure 4. Image rendered using a non-power-of-two mipmap (127^2) with isotropic (left) and anisotropic filtering (right), and rounding down.

GPU Implementation

We now describe two implementations of the mipmap construction algorithm. The first, higher precision approach employs a fragment program and operates in one pass, while the second, lower precision approach employs no fragment program and operates in two passes. The latter is faster.

To implement automatic generation of mipmaps and mipmapping for dynamic textures at high speed, the filtering can be carried out using the GPU.

The simplified polyphase box filter is very easy to implement using a fragment program in OpenGL or Direct3D that does 9 nearest neighbor samples and applies the filter weights. In order to do so, we set up the rendering in a way that the center of texel 0 of the current mipmap level maps to 0 in camera space and the center of texel n-1 maps to 1 in camera space. We then set up the fragment program and supply it with the distance between two texel in the previous mipmap level in texture space. After that we draw a quad that covers the whole current mipmap level and supply the filter weights as texture coordinates of that quad. The complete filtering to decimate an image of size lastwidth*lastheight to size width*height, looks like this.

```
...
float hx = (2.0f*width -1.0f)/lastwidth;
float hy = (2.0f*height-1.0f)/lastheight;
float w0xs, w0xe, w2xs, w2xe, w0ys, w0ye, w2ys, w2ye;

if (lastwidth&1) {
    w0xs = w2xe = (float) width/(float) lastwidth;
    w0xe = w2xs = 1.0f/(float) lastwidth;
} else {
    w0xs = w0xe = 0.5f;
    w2xs = w2xe = 0.0f;
}
if (lastheight&1) {
    w0ys = w2ye = (float) height/(float) lastheight;
    w0ye = w2ys = 1.0f/(float) lastheight;
} else {
    w0ys = w0ye = 0.5f;
    w2ys = w2ye = 0.0f;
}
float wlx = w0xs;
float wly = w0ys;

glBegin(GL_QUADS);
```

```

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
glMultiTexCoord4fARB(GL_TEXTURE1_ARB, w0xs, w2xs, w0ys, w2ys);
glVertex2f(0.0f, 0.0f);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, hy);
glMultiTexCoord4fARB(GL_TEXTURE1_ARB, w0xs, w2xs, w0ye, w2ye);
glVertex2f(0.0f, 1.0f);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, hx, hy);
glMultiTexCoord4fARB(GL_TEXTURE1_ARB, w0xe, w2xe, w0ye, w2ye);
glVertex2f(1.0f, 1.0f);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB, hx, 0.0f);
glMultiTexCoord4fARB(GL_TEXTURE1_ARB, w0xe, w2xe, w0ys, w2ys);
glVertex2f(1.0f, 0.0f);
glEnd();
...

```

Together with the following fragment program.

```

!!FP2.0
DECLARE Const0 = {1.0f/lastwidth, 1.0f/lastheight, wlx, wly};
TEX      R0, f[TEX0], TEX0, 2D;
MULR     R0, R0, f[TEX1].x;
MADR     R3.xy, Const0, {1,0}, f[TEX0];
TEX      R1, R3, TEX0, 2D;
MADR     R0, R1, Const0.z, R0;
MADR     R3.xy, Const0, {2,0}, f[TEX0];
TEX      R1, R3, TEX0, 2D;
MADR     R0, R1, f[TEX1].y, R0;
MULR     R0, R0, f[TEX1].z;
MADR     R3.xy, Const0, {0,1}, f[TEX0];
TEX      R1, R3, TEX0, 2D;
MULR     R1, R1, f[TEX1].x;
MADR     R3.xy, Const0, {1,1,0,0}, f[TEX0];
TEX      R2, R3, TEX0, 2D;
MADR     R1, R2, Const0.z, R1;
MADR     R3.xy, Const0, {2,1,0,0}, f[TEX0];
TEX      R2, R3, TEX0, 2D;
MADR     R1, R2, f[TEX1].y, R1;
MADR     R0, R1, Const0.w, R0;
MADR     R3.xy, Const0, {0,2,0,0}, f[TEX0];
TEX      R1, R3, TEX0, 2D;
MULR     R1, R1, f[TEX1].x;
MADR     R3.xy, Const0, {1,2,0,0}, f[TEX0];
TEX      R2, R3, TEX0, 2D;
MADR     R1, R2, Const0.z, R1;
MADR     R3.xy, Const0, {2,2,0,0}, f[TEX0];
TEX      R2, R3, TEX0, 2D;
MADR     R1, R2, f[TEX1].y, R1;

```

```
MADR    o[COLR], R1, f[TEX1].w, R0;
END
```

Two-Pass Approach

If we take a closer look at what the filter does, we can rewrite the filter weights w_0 , w_1 and w_2 as

- $w_0 = (1 - a) / 2$
- $w_1 = (1 - b) / 2 + a / 2$
- $w_2 = 1 - w_0 - w_1 = b / 2$

with

- $a = 1 - 2 * w_0 = 1 - 2 * (n - x) / (2 * n + 1) = (1 + 2 * x) / (2 * n + 1)$
- $b = 2 * w_2 = (2 + 2 * x) / (2 * n + 1)$.

We can then change the calculation of the current mipmap from

```
mipmap[l][x] = w0 * mipmap[l-1][2*x ] + w1 * mipmap[l-1][2*x+1] +
              w2 * mipmap[l-1][2*x+2];
```

into

```
mipmap[l][x] = 0.5f * ((1 - a) * mipmap[l-1][2*x ] +
                    a * mipmap[l-1][2*x+1]) +
              0.5f * ((1 - b) * mipmap[l-1][2*x+1] +
                    b * mipmap[l-1][2*x+2]);
```

Now the filtering is implemented as two linear interpolations, one by a , and the other by b , followed by the average of two textures. Additionally, if we compare $b(x)$ to $a(x)$, we get:

- $a(x) = (1 + 2 * x) / (2 * n + 1)$
- $b(x) = (2 + 2 * x) / (2 * n + 1)$
- $a(x+1) = (3 + 2 * x) / (2 * n + 1)$
- $b(x+1) = (4 + 2 * x) / (2 * n + 1)$

So the first two linear interpolations can be combined into a single one and the average can be calculated as a width 2 box filter. Therefore we can do the filtering in two passes of standard texture mapping with bilinear filtering. We first set up an intermediate texture of size $2 * \text{width} \times 2 * \text{height}$ and render the first quad.

```
...
glViewport(0, 0, 2*width, 2*height);
glOrtho(0, 2*width, 0, 2*height, 0, -1);

float lx = (float) (lastwidth - 2*width) / (2*lastwidth * lastwidth);
float ly = (float) (lastheight - 2*height) / (2*lastheight * lastheight);
float hx = 1.0 - lx;
float hy = 1.0 - ly;
```

```

glBegin(GL_QUADS);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, lx, ly);
glVertex2f(0.0f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, lx, hy);
glVertex2f(0.0f, 2.0f*height);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, hx, hy);
glVertex2f(2.0f*width, 2.0f*height);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, hx, ly);
glVertex2f(2.0f*width, 0.0f);
glEnd();
...

```

After that we render the second quad using the intermediate one as the current texture.

```

...
glViewport(0, 0, width, height);
glOrtho(0, width, 0, height, 0, -1);

glBegin(GL_QUADS);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
glVertex2f(0.0f, 0.0f);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 1.0f);
glVertex2f(0.0f, height);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 1.0f);
glVertex2f(width, height);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0f, 0.0f);
glVertex2f(width, 0.0f);
glEnd();
...

```

Although this implementation is a lot simpler, the result is nearly always the same as for the single pass algorithm. It would be exactly the same if not for quantization of fractional texture coordinates during bilinear interpolation, and quantization of pixel values during the first pass. We have found that mipmaps constructed with the two pass algorithm differ in pixel value from those of the one pass algorithm by at most 1 in each level with non-integer scaling, however. The average of each mipmap level stays constant with this approach; so the 1x1 mipmap level has an error of 1 or less. Since the two pass algorithm is about twice as fast as the one pass algorithm, we recommend it.



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, the NVIDIA logo, are trademarks of NVIDIA Corporation.

Microsoft, Windows, the Windows logo, and DirectX are registered trademarks of Microsoft Corporation.

OpenGL is a trademark of SGI. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© Copyright 2005 NVIDIA Corporation

developer.nvidia.com

The Source for GPU Programming



nVIDIA.

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