Per-pixel MIP Mapping

MetaCreations Tech Memo
Michael Herf
7/21/99

Introduction

Accurate texture mapping avoids aliasing by sampling at different frequencies based on available gradient information. A simple texture sampling technique suitable for real-time is trilinear MIP mipping, based on Williams's MIP map. To implement this, I have created libraries to quickly make and sample Gaussian pyramids. To write a scanner or ray-tracer to use these, you'll have to know how to drive LOD computations in your code.

The first portion of this document focuses on optimization for scanners, texture mappers that evaluate a plane (and can incrementalize many of their computations). However, the techniques described in Part 4 and later are applicable to ray tracers and will help make your code fast.

The basic task we want to solve is the projection of a unit vector centered on the current pixel to the 2-D surface we are texturing. Failing analytic methods, ray tracers can "trace" two rays along each axis, one at (x + 0.5, y) and another at (x, y + 0.5), finally connecting the points to infer a derivative vector. See references on "generalized ray tracing."

Part 1: Conventions

In our discussion, we use a 3x3 matrix for all transformations from texture to screen space and vice-versa. A 3x3 matrix can capture all varieties of planar perspective projection, and is easily invertible. Because of convention and legacy code, we still represent all matrices as those suitable for multiplying row vectors. (This contrasts with newer APIs such as OpenGL, which use column vectors.)

To summarize standard texture mapping, a homogeneous screen coordinate $\begin{bmatrix} x & y & 1 \end{bmatrix}$ can be transformed using the following (matrix provided represents the inverse screen-to-texture projection):

$$\begin{bmatrix} u' & v' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} = \begin{bmatrix} ax+by+c & dx+ey+f & gx+hy+i \end{bmatrix}.$$

From this, the texture-space coordinates can be obtained with a homogeneous division per pixel:

$$\begin{bmatrix} u & v \end{bmatrix} = \begin{bmatrix} u'/w' & v'/w' \end{bmatrix} = \begin{bmatrix} \frac{ax+by+c}{gx+hy+i} & \frac{dx+ey+f}{gx+hy+i} \end{bmatrix}.$$

This is well known. What we're most concerned with is the computation of LOD for texture mapping. We answer here, "How do you efficiently compute LOD per pixel?"

Part 2: Analytic derivatives

We're concerned about four partial derivatives for an accurate texture mapper. These we can compute using standard calculus, so we simply list the correct equations here.

$$\frac{du}{dx} = \frac{w'a - u'g}{w'^2}$$

$$\frac{dv}{dx} = \frac{w'd - v'g}{w'^2}$$

$$\frac{du}{dy} = \frac{w'b - u'h}{w'^2}$$

$$\frac{dv}{dy} = \frac{w'e - v'h}{w'^2}$$

Part 3: Common terms, forward differencing

As we know from standard texture mapping, u', v', and w' are linear across a scanline. Each of these can be computed at a cost of one add per pixel, and the resulting texture coordinates require a divide by w'.

To talk further about derivatives, we'll name some variables:

j := w'a – u'g => dudx = j / w'^{2}.
k := w'd – v'g => dvdx = k / w'^{2}.
l := w'b – u'h => dudy = l / w'^{2}.
m := w'e – v'h => dvdy = m / w'^{2}.

'l' and 'm' are constant across a scanline, and the other two quantities (j, k) are also linear, so we can compute each with a single add per pixel. We use the divide from 1/w' and square to compute 1/w'^{2}.

Now, we want to find what the projection of a unit in screen coordinates ("dx") is in texture space. We'll recombine the partials (du/dx, du/dy) to make a single vector that tells us how fast we're stepping ("du").

In general, to find ||du||, we would need a square root. However, we can build that into later computations (and have the FPU do it automatically), so we will compute signed magnitude of the projected vectors. This is simply:

```
dusq = dudx*dudx + dudy*dudy;
dvsq = dvdx*dvdx + dvdy*dvdy;
```

We can save a multiply by refactoring and precomputing j*j and k*k, since they are constant across each scanline:

```
invw4 = invw2 * invw2;   // 1/w^4 and 1/w^2
dusq = (j2 + l*l) * invw4;
dvsq = (k2 + m*m) * invw4;
```

(The alternative is to multiply by 1/w'^{2} for each term in advance, which requires one additional multiply.)

Finally, we find the stepsize in texture space by taking max(dusq, dvsq). A final factoring gives us:

```
max(j2 + l*l, k2 + m*m) * invw4;
```

Part 4: Floating point hacks, efficient exponents

Now that we have a floating point stepsize in u, v-space, we can use the exponent of the floating point number to efficiently find the LOD.

We use the following routine:

```
// integer log2 of a float, clamped to zero
inline uint32 ilog2clamp(float x)
{
    uint32 ix = (uint32&)x;
    uint32 exp = (ix >> 23) - 127;
    uint32 clamp = (~exp >> 8) & exp & 0xFF;
    return clamp;
}
```

The LOD is simply ilog2clamp(max(du, dv)) / 2.

Note: the non-clamped version is as follows:

```
// integer log2 of a float
finline int32 ilog2(float x)
{
        uint32 ix = (uint32&)x;
        uint32 exp = (ix >> 23) & 0xFF;
        int32 log2 = int32(exp) - 127;

        return log2;
}
```

Next, we can still use the squared magnitude of du or dv to compute the trilinear interpolant. We take advantage of the FPU to read the mantissa (an interpolant between two sequential powers of two), and mask in the least-significant bit of the exponent to interpolate between two exponents.

This results in a double piecewise-linear approximation, since each of our squared LOD spans two exponents:

```
// trilinear interpolant for squared numbers
inline uint32 trivalsq(float x)
{
    uint32 ix = *(uint32*)&x;
    uint32 c = ix >> 16 & 0xFF;
    return c ^ 0x80;
}
```

If you can compute the derivative as a non-squared quantity, you can use the following formula instead:

```
// trilinear interpolant (percentage to next power of two)
inline uint32 trival(float x)
{
    uint32 ix = *(uint32*)&x;
     return ix >> 15 & 0xFF;
}
```

Using the MIPMap3d Interface

Now that we know how fast we're moving across a texture, and we have a trilinear interpolant, we can use the MIPMap3d interface to look up texture values.

The interface provides several levels of access. We'll discuss each of these from lowest-level to highest-level.

Image interface

> `mip->LODImage(uint32 lod)` returns a pointer to a 2^{lod} x 2^{lod} image in 32-bit ARGB format. You may treat this as a texture in any form you like, writing to it, reading it, etc.

Pixel interface

`mip->Texel(uint32 x, uint32 y, uint32 lod)` returns a pointer to the texel found at the specified lod at local coordinates (x, y). i.e. x and y must be smaller than 2^{lod}.

This interface is accessed heavily by the filtered interface below, and it is very fast.

Filtered interfaces

All of these interfaces respect a fixed-point coordinate system of 12.20 bits, allowing you to do external wrapping calculations with significant accuracy. Regardless of LOD, you can address u and v in the same fixed point coordinate system. 0 corresponds to the border of the texture, and 0xFFFFF corresponds to right/bottom minus epsilon.

`mip->Nearest(uint32 fixu, uint32 fixv, uint32 lod)` is the only non-boundschecked filter interface. Use for scanners that are confident they never specify coordinates larger than 0xFFFFF.

`mip->NearestWrap(...)` wraps input coordinates quickly.

`mip->NearestClamp(...)` clamps input coordinates to edge pixels.

`mip->NearestClip(...)` returns '0' outside texture. This is correct if you regard the texture as having premultiplied alpha, since edges will be transparent.

`mip->BilinearWrap(...)` wraps input coordinates using bilinear interpolation.

`mip->BilinearClamp(...)` clamps input coordinates.

`mip->BilinearClip(...)` clips, mixing with '0' for border pixels and pixels outside.

Other Utilities

`mip->Levels()` returns the maximum available LOD.

Converting ilog2clamp(step) to LOD

After obtaining the log of the step, you can compute the LOD for a pixel by subtracting from mip->Levels() and clamping at zero. For example:

```
int32 lod = mip->Levels() - ilog2clamp(step);
if (lod < 0) lod = 0;

destination[0] = mip->BilinearWrap(fixu, fixv, lod);
```

Per Poly MIP [draft]

Per-poly mipmapping tries to balance aliasing and blurring in a fair way. This typically means that a point near the center of the triangle will be correctly sampled, ones nearer may blur, and ones further away may alias.

To simplify computation, we "measure" each triangle by determining its area in texture space. For objects with (u, v) fixed per vertex (even if the vertices morph), this value can be precomputed. Given an input mipmap "mip" and a triangle with texture coordinates uv0, uv1, and uv2, we do the following:

```
const real sqmip = mip->Size() * mip->Size();
Vector p1 = uv1 – uv0;
Vector p2 = uv2 – uv0;
tri.invarea = 1.0 / (sqmip * Cross(p1, p2).Mag());       // 1 / (2x the area of the triangle)
```

Notice that we pre-invert the area, to avoid divides at triangle setup time.

After projection to the screen (during triangle setup), we have three 2-D vertices: s0, s1, s2. Since the normal is definitely perpendicular to the view plane, we simply compute the 'z' coordinate of the cross product, which is also the vector magnitude, and twice the area of the triangle:

```
Vector q1 = s1 – s0;
Vector q2 = s2 – s0;
real sarea = fabs(q1.x * q2.y – q1.y * q2.x);
```

And we compute the mip level like this:

```
real ratio = sarea * tri.invarea;
int32 logr = ilog2(ratio * bias);       // bias can be (0.5 to 2), but 1 is fine.
int32 lod = mip->MaxLOD() + logr;
Clamp(lod, 0, mip->MaxLOD());
```

We can use the resulting LOD to map the entire triangle.