

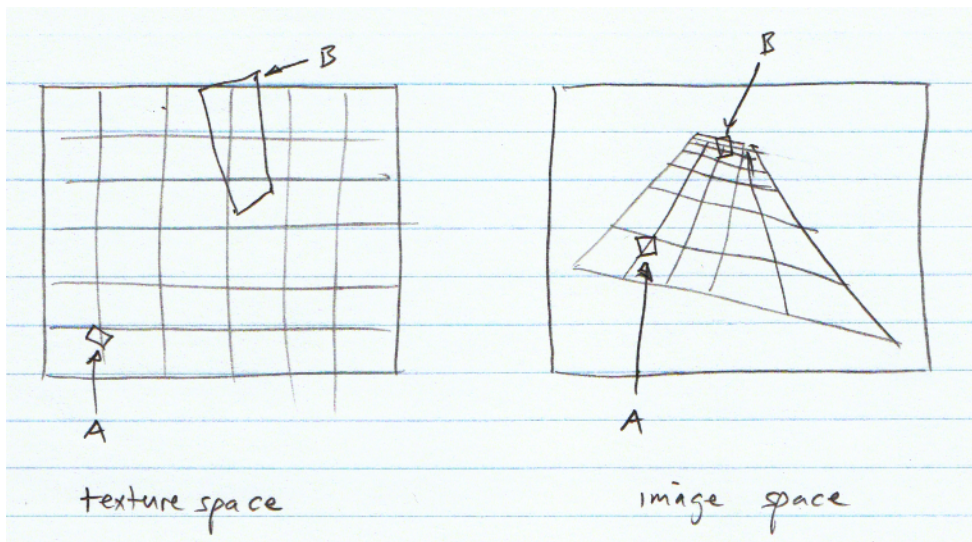
# Texture filtering

Steve Marschner  
Cornell University  
CS 4620 Fall 2008, 5 November

Texture maps viewed in perspective create a difficult antialiasing problem. A single polygon spanning a large range of depths can result in both strong magnification of texture in the foreground and extreme shrinkage, or “minification,” of texture in the background. This means that, unlike in the image resampling discussed in Chapter 4, the filter used for antialiasing needs to be very different in different areas of the image. This breaks the regular, efficient separable filtering procedure that works so well for applying the same filter everywhere when reducing and enlarging images. For textures, we need a filtering scheme that can efficiently compute averages of varying-size areas of the texture.

The goal of texture filtering, like any other antialiasing procedure, is to compute the average value of the image over an area around each pixel. We could compute this average by supersampling in image space: take many sample points spread over the pixel’s rectangle, transform them all into texture space, sample the texture, and average. This would produce quite accurate results. However, in cases of extreme texture minification, a great many samples would be needed, which would be too expensive, and many samples would be wasted for areas without a lot of fine detail.

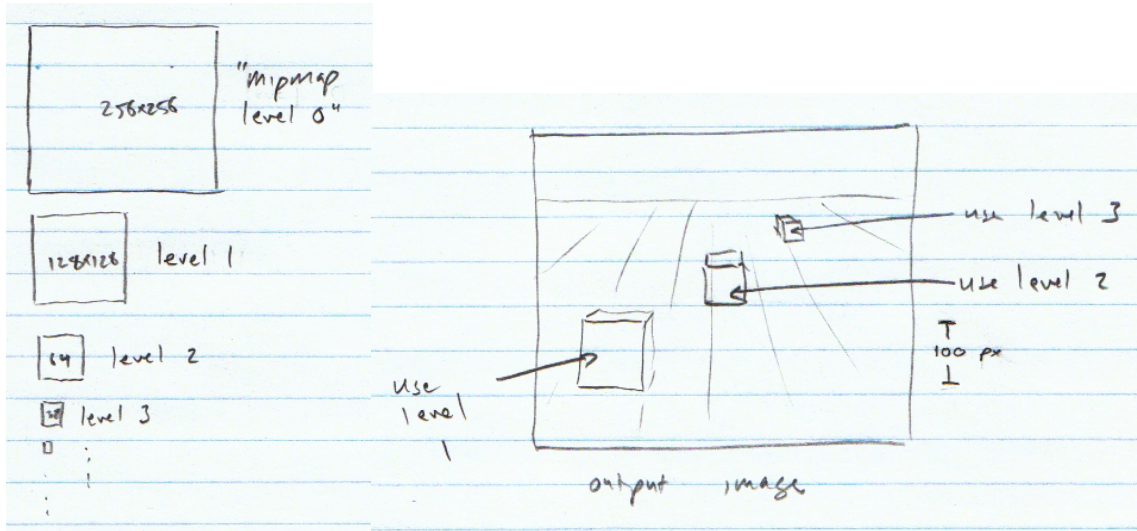
Another way to think of texture filtering is in terms of averages in texture space, rather than image space. For the commonly used box filtering, computing the average value of a pixel is (nearly) the same as computing the average value of the texture over the area that projects into a one-pixel-wide square. The texture content that projects into that square area comes from a quadrilateral-shaped region in the texture map, known as the *footprint* of the pixel:



This is the way most texture filtering methods think about the problem: we have to compute the average value of a texture over some region that will be different for every pixel. A range of different methods is available, each making different quality-speed tradeoffs: the filtered texture val-

ues can be computed faster if we are willing to compromise on the area being averaged. The most widely used approach is called the *mip map*, which tries to get the size of the footprint right but does not worry about the shape.

The idea of the mip map (mip stands for “multim in parvo,” reputed to mean “many things in a small space”) is to pre-compute an *image pyramid*, which is a set of copies of the texture image, each scaled down by a factor of two compared to the previous one. Then each of these scaled-down textures can be used directly whenever the effective size of the texture in the output image is about the same as the scaled-down mip map texture:



In this example, the original texture map, which is the *base level* of the mip map, or “level 0,” is 256 texels (a *texel* is a texture pixel) square, and the first scaled-down texture, called “mip map level 1” is 128 texels square, and so forth. Any time a shading procedure is supposed to sample a texture at texture coordinates  $(u, v)$ , it can just as easily sample level  $d$  of the mip map instead (at the same texture coordinates), thereby automatically getting a version of the texture that is averaged over a square area of size  $2^d$  by  $2^d$ . The front face of the front cube is about 128 pixels high in the image, so it should be textured by sampling at level 1; the next cube is farther away and appears about 64 pixels high, so it should use level 2, and so forth. Since the mip map is precomputed, there is no additional cost compared to just point-sampling the texture (corresponding to always using level 0). You can think of the mip map as being three-dimensional: you sample at the texture coordinates  $(u, v, d)$ , where  $u$  and  $v$  tell where in the texture you’re sampling, and  $d$  tells what level of detail you want.

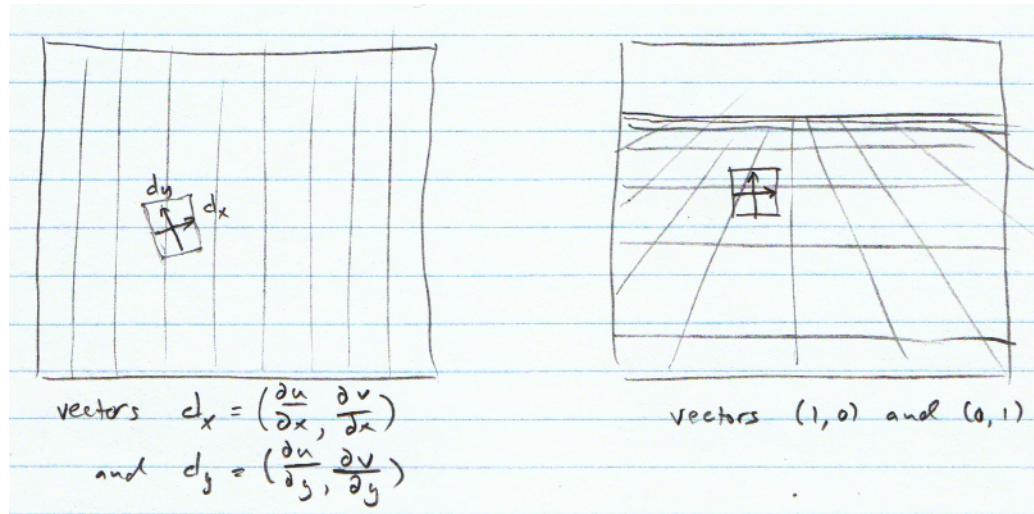
Usually one computes  $d$  as a real number. Rounding it to select a mip map level can produce noticeable jumps between levels, so it’s common to linearly interpolate: sample both levels  $\text{floor}(d)$  and  $\text{ceil}(d)$ , then interpolate linearly between the results. When combined with bilinear interpolation within each mip map level, this is *trilinear interpolation*, and it requires fetching 8 pixel values from the mip map data.

The one important piece that is still missing is: how do we know what value to use for  $d$ ? If we are displaying front-facing rectangles like the ones in this example, it’s easy enough to see what  $d$  should be, but what about the general case of a texture on a polygon viewed in perspective, or a curved surface? One approach would be to match the area of the mip map lookup to the area of

the quadrilateral footprint. We know a pixel at level  $d$  covers an area  $2^{2d}$  (measured in terms of the pixels of the original (level 0) texture map). If we find that the area of the footprint is  $A$ , then we can match the area by using

$$d = \log_2 (\text{sqrt}(A)). \quad (1)$$

Unfortunately, finding the quadrilateral explicitly and computing its area is a somewhat awkward computation. Since so much has already been approximated, we can instead linearize: find the derivatives of the texture coordinates  $u$  and  $v$  with respect to the screen coordinates  $x$  and  $y$ , and approximate the footprint with a parallelogram whose axes are defined by those derivatives:



This amounts to using the first-order Taylor expansion of the functions  $u(x, y)$  and  $v(x, y)$  around the point  $u_0, v_0$  that is the image of the point  $x_0, y_0$  under the texture coordinate function:

$$u(x, y) \approx u_0 + (\partial u / \partial x)(x - x_0) + (\partial u / \partial y)(y - y_0)$$

$$v(x, y) \approx v_0 + (\partial v / \partial x)(x - x_0) + (\partial v / \partial y)(y - y_0)$$

and mapping the pixel's 1-by-1 square through that approximation. The area of this parallelogram is

$$A = |(\partial u / \partial x)(\partial v / \partial y) - (\partial u / \partial y)(\partial v / \partial x)|$$

(Another way to see this: it is the determinant of the Jacobian of the 2D to 2D texture coordinate function.) With Eq. (1) this approach gives us a mip map level. It's straightforward to work out the formulas for the four partial derivatives, given the projective matrix that transforms  $(x, y)$  to  $(u, v)$ .

An advantage of working from the derivative is that the same procedure then generalizes to situations where the mapping is not projective—e.g. texture maps on curved surfaces, or texture maps used as reflection maps.

It's important to realize that the computation of  $d$  from the derivatives is a heuristic: no matter what level we choose, the filter footprint will be dramatically wrong when the texture is stretched

anisotropically so that the correct footprint is very elongated. Another common heuristic is to use length of the longer of the vectors  $d_x$  and  $d_y$  in the figure, which is more conservative: it tries to make the mip map area large enough to entirely contain the elongated pixel footprint.

In my examples I have used a base texture with a power-of-two size. This is convenient but not necessary: we can just round off the sizes each time we halve the image dimensions (OpenGL rounds down), and proceed as usual, using a decent resampling filter to compute the almost-exactly-half-size image. As long as we keep track of the size of each level, sampling each level is just like sampling any other texture:  $u$  and  $v$  are between 0 and 1, and we find the texel coordinates based on the actual size of the mip map at the level we're looking at.

Many real systems based on rasterization take advantage of the fact that the rasterizer has already computed the increments in  $u$ ,  $v$ , and  $w$  for single-pixel increments in  $x$  and  $y$ . These can be used to compute approximations of the partial derivatives used to compute  $d$ , and they are often used instead of analytically computing the derivatives.