CS 314: Magnification and Minification

Robert Bridson

November 4, 2008

1 Looking up Texture Values

Last time we looked at defining 2D textures in terms of setting up mapping from 3D points on the surface of an object to 2D texture coordinates, that can then be fed into a procedural texture function or used to look up a colour from a regular image. We'll now focus on this last possibility: once we know the texture coordinates, exactly what colour value should we get back from the image?

Actually, there is first a small technical issue of mapping between texture coodinates, which might be defined on any arbitrary rectangle in 2D, and actual texel coordinates (e.g. numbers between 0 and 512 for a 512×512 texture image). This is handled as usual by a windowing transform, which can be conveniently represented with a transformation matrix: as mentioned last time, OpenGL even provides a matrix stack to account for exactly this sort of thing. We'll assume that's easy enough for you to set up, so we can more precisely assume we are given texel coordinates into the texture image, and need to return an appropriate value.

As mentioned last time, the tricky cases to consider are **magnification** and **minification**, i.e. if a texel is bigger than or smaller than the corresponding pixel in the image. Magnification is easier to think about, so we'll tackle that first.

1.1 Magnification: Nearest-Neighbour Interpolation

The simplest thing of all to do is return the colour of the nearest texel, which is called **nearest neighbour interpolation**. If the texel coordinates are (p, q) we can round them to the nearest integer values (i, j), look up the colour stored in the array at location (i, j) and return that.

Unfortunately, this has serious visual artifacts under magnification, assuming as is almost always

the case that the texture image represents a mostly smooth function. In magnification, several pixels will get mapped to the same texel, and thus will all get the exact same colour; at some point in the output image the next group of pixels will be mapped to the next texel over, a different colour. We see an edge between the two groups of pixels—that shouldn't be there, and is just an artifact of the boundary between one texel and the next—and unfortunately the human visual system is especially sensitive to edges so it pops out at us. We see the textured geometry as if it had a grid of big squares (maybe distorted) painted on, each of constant colour, instead of a smoothly varying texture.

As an aside, it's worth delving a bit deeper into how the human visual system works, since ultimately computer graphics is mostly about fooling or pleasing human viewers. We've already talked about how light enters our eyes through the pupil (and lens and cornea) and hits the retina at the back of the eye, where there are light-sensitive cells which send neural signals when light strikes them. However, those raw neural signals don't immediately get passed to our higher consciousness: that would make seeing as difficult as reading a text print-out of the values in the pixels of an image and trying to figure out what the picture is of. Instead we have evolved a "low-level" system of processing those raw retinal signals in ways that highlight important features of the image, and its that ensemble of information that eventually makes it to our higher consciousness. Much of the lowest level of processing can be modeled mathematically as taking two input signals—say from neighbouring retinal cells—and taking either the sum or difference. The output from that might feed into another sum or difference. The summing essentially averages or blurs the light over a region, letting us instantly understand if one whole region of an image is darker or lighter for example. The differencing instead detects image variations from one point to a nearby point, which happens most strongly at an edge in the image. Part of what our visual system does is detect and flag edges in an image, and forces us to pay attention to them and try to figure out what they mean. If a computer graphics image has edges that don't mean anything, it looks bad. (We saw another case of this a long time ago, incidentally, noticing that when oblique triangles are rasterized they produce a jagged pattern of pixels: that jagged pattern is full of bad edges our visual system notices and objects to.)

Getting back to texture magnification, it's clear we need a way to get smoothly varying pixel values between texels, eliminating these bad edge artifacts. We've actually solved this problem several times, partially, with the idea of using barycentric coordinates to linearly interpolate colours or normals from the vertices in a triangle mesh. We can't directly use linear interpolation here since we have values on a grid of squares, not triangles, but the idea is still a good one.

1.2 Magnification: Bilinear Interpolation

We can instead use **bilinear interpolation**, which basically means linearly interpolating along one dimension and then linearly interpolation along another (in 3D, this is extended to **trilinear interpolation**, working over three axes).

Given texel coordinates (p, q), the first step is to find the surrounding four texels (i, j), $(i + 1, j)$, $(i, j + 1)$ and $(i + 1, j + 1)$ where $i = \texttt{floor}(p)$ and $j = \texttt{floor}(q)$. Let the fractional part of p be $\theta = p - i$ and the fractional part of q be $\phi = q - j$. These values, between 0 and 1, tell us the fraction of the way the point is between the texels along x and along y.

Our first step is to do 1D linear interpolation along the x-axis, twice. We'll estimate the texture values at points (p, j) and $(p, j + 1)$, as

$$\begin{aligned} f(p, j) &\approx (1 - \theta) f(i, j) + \theta f(i + 1, j) \\ f(p, j + 1) &\approx (1 - \theta) f(i, j + 1) + \theta f(i + 1, j + 1) \end{aligned}$$

These new values are lined up with the point (p, q) along the y-axis. We can then do 1D linear interpolation between them to get the final interpolated value:

$$f(p, q) \approx (1 - \phi) f(p, j) + \phi f(p, j + 1)$$

This is probably the best way to evaluate bilinear interpolation. However, if you're curious we can expand this out in terms of the original texel values,

$$f(p, q) \approx (1 - \phi)(1 - \theta) f(i, j) + (1 - \phi)\theta f(i + 1, j) + \phi(1 - \theta) f(i, j + 1) + \phi\theta f(i + 1, j + 1)$$

at which point it should be obvious it doesn't actually matter if we interpolate along x first and then y or the other way around: we get the same answer.

Using bilinear interpolation to deal with magnification is a huge improvement over nearest neighbour: the final image no longer has spurious discontinuous edges lined up with the texture image, and our visual system isn't nearly so distracted.

However, it's not perfect either. Linear interpolation, and hence bilinear interpolation, guarantees that the values interpolated between two grid points will be in between the values at the points. This implies local maxima or minima[1] only happen at grid points, or in the 2D bilinear case along the grid lines. Unfortunately our eyes are also pretty good at spotting the points or curves of maximum/minimum

[1] Remember from calculus that a local maximum (or minimum) is a point where the value is largest (or smallest) in a small neighbourhood. There might be larger (or smaller) values elsewhere, but not immediately nearby.

intensity in an image (though much less sensitive to this than to edges), and thus we still get distracted by these grid artifacts to some extent.

Going from piecewise constant (nearest neighbour) to piecewise linear continuous interpolation helped out a lot; taking this further to piecewise quadratics or cubics or higher degree polynomials, with higher degrees of smoothness (such as differentiability) can essentially eliminate the problem. This comes with a cost, however: they are much more expensive to evaluate (and thus aren't even available from real-time APIs like OpenGL) and trade grid artifacts for *blurring* (true sharp edges in the images become less distinct). The precise definition of these higher order schemes, generally termed **splines**, is left for fourth year courses.

1.3 Minification: the Naïve Way

Using the algorithm as above (with any degree of interpolation, but let's assume bilinear for the sake of concreteness) can fail spectacularly for minification situations, where each image pixel "covers" many texels. In a nutshell, each sample point in the image will interpolate its texture value from the four surrounding texture samples, meaning most of the texture samples (those that lie in between) will be completely ignored.

This is a big problem. For example, a classic trick underlying black and white displays as well as laser printing is to approximate shades of grey by using a pattern mixing black and white. Even from a small distance, your visual system averages over the black and white mixture and gives you the impression of looking at a shade of grey; from a further distance the averaging happens inside your eye at the retina, and you can't possible distinguish the difference between the black and white pattern and an actual uniform grey. However, if we rendered this black and white pattern as a minified texture in the naïve way, we might end up ignoring all the white parts and thus produce a pure black image (or vice versa).

This is the first guide to doing a better job of minification: we would like the average colour of the texture to (ideally) remain constant as we zoom out. In the limit where the entire texture occupies just a single pixel in the image, the colour of that pixel should be the average of the entire texture image, not just the value from a randomly chosen texel.

1.4 Minification with Averaging

One conceptual approach to the problem is to turn things around: instead of looking up texels based on image samples, find the image sample that each texel should contribute to. That is, for every texture sample, find the nearest image sample and include it in the average there; each image sample will be the average of all the closest texels. This way, every single texel will make a contribution to the final image, nothing is ignored, and the correct average is preserved.

This isn't bad at all, but has two issues. The first is that the nearest-neighbour averaging still causes some image artifacts—which we'll postpone discussing until the next lecture, where we take a more advanced signal-processing approach to the problem—but just like interpolation in the magnification case this can be ameliorated by using a weighted average with bilinear or higher order weight functions. The second issue is that this is fairly difficult to implement efficiently for texture mapping.

To make it practical, we take a slightly different tack. Instead of computing an average of texels on the fly for each image sample, we first resize the texture image to be lower resolution, making each new texel large enough that we no longer are in a minification situation: then simple interpolation should work fine. The values of the new texels are calculated as appropriate averages from the original texture, which is easy enough to do as a pre-computation.

Now the problem shifts to picking the right resolution to which reduce the texture image. Without getting into the gory mathematical details, a bit of calculus can derive a rough bound on the ratio of the space between image samples and the space between corresponding texture samples[2]. If this ratio is one or less, we're not minifying and we can proceed as before; otherwise we reduce the resolution of the texture image by that ratio. Each texel of the lower resolution texture image should be an average of the appropriate block of texels in the original.

1.5 Minification with Mipmaps

So far so good, except this still isn't practical. With perspective projection and/or texture coordinate distortion, even in a single image there may be many different ratios of pixel to texel spacing, and if we're interactively viewing a textured object each frame could certainly have different ratios. We can't afford to store (or compute on the fly) every lower resolution version needed. However, we don't need to be

[2]Basically we need to consider the mapping from a pixel to a point in the texture, which is the inverse of the composition of the following: map from texture to 3D geometry, then map from 3D geometry to camera space (multiply by the model-view matrix), then project to normalized device coordinates (multiply by the projection matrix, and apply homogenization), then scale to pixel coordinates. If we differentiate this with respect to pixel coordinates, we get an idea of the ratio of pixel size to texel size.

perfect: we can instead store and precompute "enough" of a range of different sizes that by picking the nearest we get a good enough result. This underlies the technique called **mipmapping**, where the original texture image is augmented with a version half the size, a quarter of the size, an eighth of the size, and so on down the powers of two to a single pixel version.

There's still one source of artificial discontinuity here: particularly with a perspective projection where a single texture may be magnified in the foreground but end up minified in the distance, there could be objectionable sharp edges where the renderer switches from one level of the mipmap hierarchy to the next. These can be smoothed over too by linearly interpolating the results between the nearest two mipmap levels (at a small extra cost).

Finally, it should be noted that because we're dealing 2D images, it's quite possible that along one axis the texture is being magnified while along the other it's minified. (Imagine looking closely at a piece of paper that's almost edge-on to the viewer.) Going with the magnification algorithm is bound to cause severe artifacts along the minified axis, but going with the minification will at worst cause unwanted blurring along the magnified axis; the latter is generally less distracting and is preferred.