CS 314: Texture Mapping

Robert Bridson

October 28, 2008

1 Texture Mapping

So far we've more or less assumed that a 3D object we render either has a constant material type (diffuse, mirror, etc.) or at most has a variation per vertex in a mesh. We can interpolate colours and more between the vertices of a triangle, but we haven't thought about finer-scale variations.

However, there are many examples where this limitation is extremely inefficient: geometrically simple objects with complex variations in appearance over the surface. Consider a wooden table, with only a few flat surfaces but tiny details in the colour of the wood grain over those surfaces. The printed sheet of paper or computer screen on which you are reading this sentence is another fine example: it's basically a rectangle, but with high resolution variations in colour. We could build triangle meshes tessellated finely enough to capture these variations, but it would be terrible waste of memory and processing.

A much better alternative is **texture mapping**. The idea is to keep a simple geometric description, but use a "texture function" to efficiently encode the variation in colour or other material properties over the surface, potentially at much higher resolution.

Texture mapping can be done in several ways. We'll start off with conceptually the simplest (3D textures), and then move on to the much more common and useful but somewhat more sophisticated version (2D textures).

2 3D Textures

Imagine modeling an object carved from a piece of wood. It might be geometrically simple like a sphere, but we have to account for the complexity of the wood grain in its appearance. We do this with a "texture

function" $f(x, y, z)$ in 3D space which is supposed to give the colour[1] of the wood at the point (x, y, z): at least conceptually, this could be done by sticking a log in some sort of 3D scanner. The colour at a surface point (x, y, z) of an object carved from the wood should just be $f(x, y, z)$.

That's essentially all there is to 3D textures: define a 3D texture function $f(x, y, z)$ alongside the regular geometry, and then at any point on the geometric surface look up the colour there by evaluating f. This cleanly separates the geometric representation from the details of appearance.

One big detail that's important to get right is exactly what coordinates to send to f. Especially if the object is moving in the scene, then really we want to evaluate the texture in object space coordinates: as a wooden ball rolls, the wood grain appearance should stay fixed to the ball, not "swim" through it; we need to "undo" the rotation and translation of the ball to get from world space coordinates of a surface point to the object space coordinates at which we can evaluate the texture function. Some APIs, like OpenGL, even give you an entire separate matrix stack, similar in design to the model-view stack, just for transforming texture coordinates.

Another big deal is how to define the texture function. One approach—used especially in high quality film rendering—is to define it **procedurally**, that is actually implement f as a piece of code which gets called whenever a colour is needed. Many tricks have been developed in computer graphics research to help in constructing such functions to mimic all sorts of natural phenomena: if you're interested, you might check out the book "Texturing & Modeling: A Procedural Approach" by Ebert, Musgrave, Peachey, Perlin and Worley, the best starting reference for the subject.

The main alternative is to instead sample values of the texture function on a 3D grid, and then use interpolation to get values in between the grid points (more on this later). Each element of the 3D array of texture values is called a **voxel**, short for volume element, similar to the name "pixel" for picture element. Voxel arrays are very flexible—the artist can get the data from anywhere and manipulate it however they want—but also possibly very memory intensive.

3 2D Textures

While 3D textures work well in some scenarios like objects carved from wood, at least conceptually even if memory issues pose practical problems, they are not always the right choice. For example, a piece of paper with text printed on both sides is so thin it would be silly to represent it as more than an infinitely thin surface—maybe just a rectangle if the paper is flat—and then there's no way to sensibly define a 3D

[1] More generally, we can imagine texture functions which give more than just colour, but other material properties too, like glossiness or mirror reflectivity versus diffuse shading etc.

texture which can display different print on each side of the paper. Of course, in reality, the paper does have a nonzero thickness and the ink or toner itself has a thickness: everything is 3D in reality, but that's not always the most effective approach to a problem.

The answer to texturing the printed piece of paper is obvious: use a 2D texture function instead, one for each side of the paper. This means attaching a 2D image to the surface.

To make this work, we have to introduce a **mapping** from points on the surface of the object in 3D (say in object space) to 2D texture coordinates. For a flat rectangle modeling the piece of paper, this should be pretty straightforward: we can arrange the rectangle to have object space coordinates $0 \leq x \leq w$, $0 \leq y \leq h$ and $z = 0$, for example, and then map object space (x, y, z) to texture coordinates (x, y), with the texture image occupying the $[0, w] \times [0, h]$ rectangle.

With this map in place, rendering is almost the same as for 3D textures: for any point on a surface that is being rendered (whether rasterized or raytraced or something else) get its object space coordinates, then apply the mapping to find its 2D texture coordinates, and then find the texture value at those coordinates.

As before, the texture could be **procedural** (implemented in code as a function) or it could be based on a stored 2D grid of values—a regular image—called the **texture map**; the individual values in the texture map are often called **texels** (texture elements) to distinguish them from pixels in the final rendered image. Using texture maps (images) is very attractive, particularly for real-time rendering systems, since looking up a value in one is an $O(1)$ operation and they are also fairly memory efficient: unlike with a 3D texture, there isn't a lot of space "wasted" on voxels that will never show up in the rendering. In fact, if you build a mapping from the surface of an object to a 2D texture space, you can re-encode a 3D texture as a more efficient 2D texture—storing just the values that actually show up on the surface.

Let's look at a more interesting example of a mapping from a surface to a 2D texture space, supposing we want to convert our wooden sphere example from a 3D texture to a more memory efficient 2D texture. That is, we want a mapping from the 3D points on the surface of a sphere in object space—for simplicity, let's assume this is a unit sphere centred on the origin—to a 2D rectangle. Anybody who has tried to wrap a ball for a birthday present knows this isn't so easy.

The most common mapping of the sphere's surface to a rectangle, **latitude-longitude**, is derived from cartography (drawing actual maps of the Earth); it's also known as **spherical coordinates** in math and physics. Longitude θ is the angle of how far around the Earth a point is, in the east-west sense; latitude ϕ is an angle of a point in the north-south sense. Lining up north with the positive y-axis, and

putting longitude zero on the positive x-axis, maps latitude and longitude to 3D object space as:

$$(x, y, z) = (\cos(\phi)\cos(\theta), \sin(\theta), \sin(\phi)\cos(\theta))$$

Here we take $\theta \in [-\pi, \pi]$ and $\phi \in [-\pi/2, \pi/2]$. This is then a mapping from the 2D rectangle $[-\pi, \pi] \times [-\pi/2, \pi/2]$ to the surface of the sphere. However, we're more interested in going the other way, the mapping from a point (x, y, z) on the sphere to a 2D point in this rectangle. This mapping involves inverse trigonometic functions:

$$\begin{aligned} \theta &= \arcsin(y) \\ \phi &= \arctan 2(z, x) \end{aligned}$$

If you haven't seen it before, the last function $\arctan 2$ is available in the C standard library as `atan2`, a convenient generalization of $\arctan(z/x)$ which gets the sign of the resulting angle correct (and the $\arcsin$ is available as `asin`).

Latitude-longitude mapping does have some oddities. First of all, there is an artificial discontinuity at the negative x-axis in object space: the surface points with θ near $-\pi$ are actually right next to the surface points with θ near π, even though they are far apart in the texture coordinates: one way of thinking about this is that to unwrap the surface of the sphere (a closed surface) onto the rectangle (an open surface) we have to cut it somewhere, and we chose to cut along the $\theta = \pm\pi$ meridian. Another weird thing is that at the north and south pole ($\phi = \pm\pi/2$) we have a singularity: every value of θ maps to the same point on the sphere there. The entire top line of the texture rectangle corresponds to a single point on the sphere, as does the entire bottom line. This implies the texture is increasingly distorted near the north and south poles, meaning a normal-looking pattern drawn directly on the sphere near either pole will look nonlinearly stretched in the texture map—you're probably familiar with this form looking at latitude-longitude maps of the world and noticing how strange the Arctic and Antarctic appear.

The discontinuity and the distortion in the latitude-longitude map make it harder to design a good texture image for the sphere: it's not as simple as just finding a regular image. Unfortunately, more advanced math can prove there is no perfect mapping between the sphere and a rectangle (or even more general shapes on the plane): there will always be a discontuity somewhere, and always some degree of distortion. However, we can trade off less distortion for more discontinuous "cuts", or vice versa: one particularly attractive alternative that is often used in computer graphics is the so-called **cube map** which you can look up if interested.[2]

[2]Essentially it puts a cube around the sphere, projects the sphere outwards onto the cube, and then unwraps the cube into six connected flat squares.

For more general objects, say with all sorts of curved parts and interesting topology (like holes), finding a good mapping is much harder, and still a subject of computer graphics research. By good, we mean:

- avoids cuts or discontinuities as much as possible
- avoids distortion as much as possible
- fills a rectangle in 2D texture space as efficiently as possible

(On the subject of the last point, though we talked about mapping the sphere to the entire rectangle in 2D texture space, we can map it to just a subset, like an ellipse, that might let us avoid some distortion at the cost of wasting the memory spent on the unused parts of the texture image.) Without getting into that can of worms, we can still look at how a mapping from object space to texture coordinates can be represented for general triangle mesh models. The big idea is that across a triangle, we probably want texture coordinates to vary continuously, just like we've looked at for plain colours and Phong-interpolated normals. Aha! We just need to store the 2D texture coordinates of each vertex of the model, and we can then use barycentric coordinates as before to linearly interpolate the texture mapping across each triangle. No explicit mapping function for the full surface is needed, just the mapping at each vertex—which could be generated automatically by clever algorithms, designed by hand, or even "painted" on the model by an artist.

(As an aside, this scheme also works with 3D textures, or even 1D textures which we haven't gotten into: just specify some number of texture coordinates at each vertex, and use barycentric coordinates to linearly interpolate between them. OpenGL provides for all of this.)

Unfortunately, there is one last wrinkle in texture mapping that needs to be addressed. The exact details of interpolating from texels in the texture image to a point on the surface rendered in a pixel are very important. In particular, we have to be very careful in both the **magnification** case (where a single texel may "occupy" several pixels in the final image, i.e. it looks locally as if we have zoomed in or magnified the texture image) and the **minification** case (where a sinle pixel overlaps many texels, i.e. it looks locally as if we zoomed out or shrunk the texture image). We'll tackle these next lecture.