Representing geometric details with textures

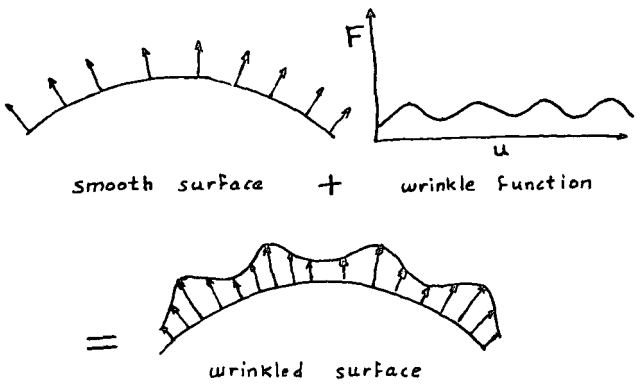
Steve Marschner
Cornell University
CS 569 Spring 2008, 14 February

To have visually appealing images, we like to have a lot of detail in the models we render. But detail is expensive—both in terms of the effort it takes to create it or measure it, and in terms of the cost of transmitting a large triangle mesh to the GPU and processing all the vertices. For this reason, people have developed a lot of schemes over the years for using less geometry in exchange for some information carried around in textures.

The basic idea of most of these schemes boils down to storing coarse geometry as a triangle mesh, and separating out the details, one way or another, in a texture map. I'm going to tell the story a bit out of order, to put the conceptually simplest idea ahead of the historically earliest one.

Displacement mapping

The basic idea behind many ways of adding surface detail is that of *displacing* the surface along its normal by a distance stored in a texture. The straightforward application of this idea is called displacement mapping:



[Blinn 78]

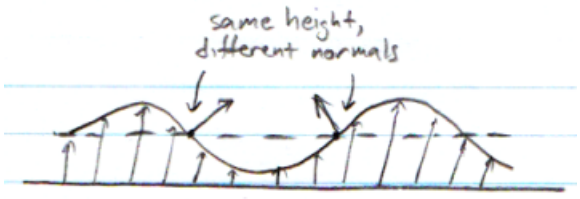
The simplest way to render the displaced surface is to go ahead and displace the vertices of a densely tessellated smooth surface. This can be done using a texture lookup in a vertex shader. Let's think about the details for a minute:

Attributes: position, normal, texture coordinates

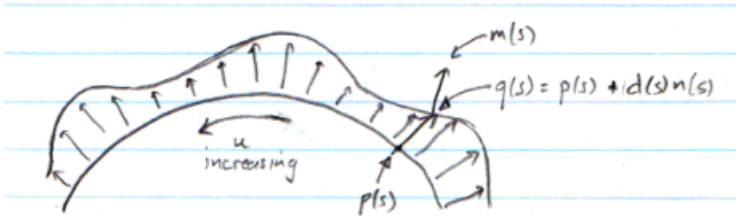
Varying: normal, texture coordinates

It's very clear what needs to be done with the position: you do a texture lookup to get a displacement, and you set the position to position + normal * displacement. (There are some more details to think about, such as whether you'd like to allow displacements to be signed, and what the meaning of a displacement of 1.0 is if you scale the geometry up and down.)

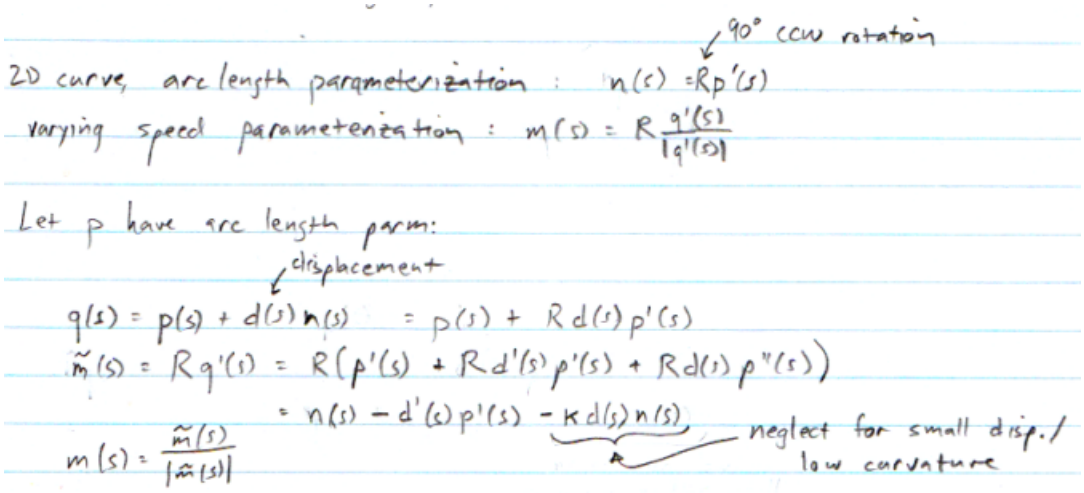
But what needs to be done with the normal? The displaced vertex needs to get a new normal, or it will be shaded much the same as the original vertex would have been. The normal depends on what displacements are being applied to neighboring parts of the surface; think of this 2D example in which the same height appears in two places with very different normals, even though the base surface is flat:



The direction of the normal depends on the slope (the derivative) of the displacement function, and on the underlying normal:



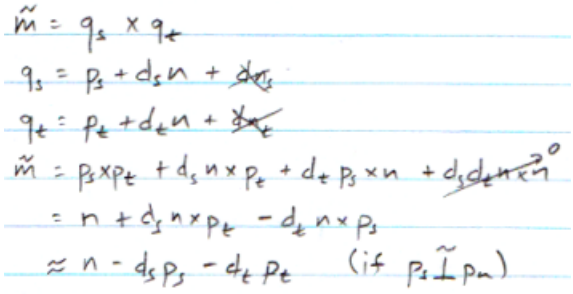
In 2D, it's relatively simple to work out the formula for the normal to the displaced surface [here labeled $q(s)$] in terms of the displacement function and an arc length parameterization of the base surface, $p(s)$:



The conclusion of this derivation is that, if we are willing to assume that the displacement is much smaller than the radius of curvature of the surface (that is, the product $d\kappa$ of displacement and curvature is small relative to 1), then the normal to the displaced surface is just the base normal altered by adding a correction in the direction of the base tangent with the magnitude of the derivative of the displacement. When the displacement is constant, the normal is the same as the

base normal; when the displacement is changing fast it pushes the normal towards the downhill side.

In 3D a similar result still holds, but it's a lot more complicated to derive. Blinn [1978] derived this result, and Lee et al. [2000] give a more mathematical view. The end result, after we discard that second-order term, is:



(Note that *m*-tilde needs to be normalized still.) This result says much the same thing as the 2D result; it's easiest to read if we think of the *s* and *t* derivatives of *p* as being about perpendicular to one another (the last line of the preceding derivation). Then it says that displacing the surface makes its normal tilt toward the downhill direction. This isn't a big surprise if you think about it for a minute. When the texture coordinates are rather distorted, so that the derivatives of *p* are far from perpendicular, you have to recognize that an *s* derivative causes the normal to tilt in the plane *perpendicular to the* t *derivative*, not in the plane of the *s* derivative. Weird.

In any case, this is all simple enough to write down in a vertex shader, save two things. First, we need to compute the two texture coordinate derivatives and get them to the vertex shader via additional attributes. Second, we don't have the derivatives of the displacement map stored anywhere; just the map itself. The canonical way to compute these derivatives is using central differences: for example, $p_S = (p(s + \delta, t) - p(s - \delta, t)) / (2\delta)$, where δ is the size of a texel in texture coordinate space.

The implication of all this is that to do displacement mapping you need to have the two tangent vectors—the derivatives of surface position with respect to the *s* and *t* texture coordinates—available. For rigid objects, one can compute these ahead of time and store them (as you did in the first assignment). Note that these are not exactly the same as the tangent and binormal you'd use for something like anisotropic shading, because they are not always orthogonal to one another.

So we end up with a vertex shader like this:

1. Attributes: position, normal, *s* tangent, *t* tangent, texture coordinates.

2. Vertex shader: look up height; displace position; look up height derivatives (4 more non-dependent texture reads); compute displaced normal.

3. Varying: normal, texture coordinates.

4. Fragment shader: compute shading using any standard model

Now that I've derived all this, I'll tell you that displacement mapping is still used more often in offline rendering than in real time, because you still have to feed in a highly tessellated base surface. Things are starting to change so that you can dice up coarse geometry on the GPU, then process the triangles without ever having them see the CPU, and in that case displacement mapping is a big win in terms of CPU-GPU bandwidth.

But in the grand scheme of things, displacement mapping is more often used as a modeling tool: a way to conveniently add detail to a model using a paint program to control the surface relief (see toy locomotive example due to Paweł Filip).

Lee, Moreton, & Hoppe [2000] demonstrate how effective displacement is with a nice, smooth subdivision surface as the base surface.

One use of displacement mapping (as long as the base surface is well-behaved) is as a way to animate a complex surface: make it a displacement from a smooth surface, and animate the smooth surface. We'll talk more about animating detailed meshes later on in the course.

Bump mapping

If you look at a displacement mapped surface that's facing mainly toward the camera, the displacements hardly change the surface position in the image—most of the visible effects are through the surface normal. The idea of bump mapping, which predates displacement mapping because it is much cheaper, is to ditch the displacement itself but keep the effect of the displacement on the surface normal.

The earliest work on simulating surface details was by Jim Blinn in 1978. At the time, using enough triangles to get a texture of small-scale bumps—for instance, to render the peel of an orange, the iconic image from this paper—was out of the question. Generating a one-channel texture that stored the bumps as a displacement field was feasible, though, and he observed that what really matters for details in shading is the fast-changing *normals*, not the changes in the actual surface position itself. So the idea was to *think* in terms of a displaced surface to compute normals that would then be used to shade the flat polygons without actually *constructing* the displaced surface.

Now that we already derived displacement mapping, the bump mapping is easy: Forego the change in the surface position, but keep the change to the surface normal, and move the computation to the fragment shader.

1. Attributes: position, normal, *s* tangent, *t* tangent, texture coordinates.

2. Vertex shader: pass everything straight through.

3. Varying: normal, *s* tangent, *t* tangent, texture coordinates.

4. Fragment shader: look up height derivatives (4 more non-dependent texture reads); compute displaced normal; compute shading using any standard model.

Normal mapping

Bump mapping has been a staple in the toolbox for quite some time, but a method that trades heavier use of texture memory for a decrease in complexity has become quite popular as texture memory has grown. The idea is to just go ahead and store the normal—all three components— in the texture map, to save the trouble of computing the derivatives of the bump map at render time. The normal map could be derived from a displacement map or bump map using the equations above (this is probably easiest if you wanted to paint the map); or it could be computed on its own (more on this later).

There is a choice of what space to represent the normals in. The simplest thing is to store them in object space—that is, store all of them with reference to the same 3D coordinate system you're using to store the vertex positions. Then you only need to transform the normal into world space, which is a fixed (uniform) transformation. But this is not a good choice for deformable surfaces because you'd need a way to deform the normals with the surface—they don't all want to transform by the same matrix.

A second option is to store the normals in a local frame on the surface (in tangent space). This comes with the cost of transforming the normals into world space for lighting calculations (this is more trouble than transforming the from world to object space because it involves a transformation that changes per fragment), but has a number of advantages that make it usually the preferred approach. You can use a tangent-space normal map on any geometry, or move it around on the surface, without worrying about recomputing the normals. The underlying surface can also deform, and the normals will be carried with it.

So for tangent-space normal mapping, the operations are:

1. Attributes: position, normal, tangent, binormal, texture coordinates.

2. Vertex shader: pass everything straight through.

3. Varying: normal, tangent, binormal, texture coordinates.

4. Fragment shader: look up tangent-space shading normal; construct tangent-space matrix; transform normal to world space; perform shading.

Note that you could compute the binormal at either the vertex or the fragment stage if you need to conserve CPU–GPU bandwidth or vertex attributes. If you can get away with object space normals (your geometry is rigid and rigidly transformed, and your texture coordinates are one-to-one) then you can use a much simpler implementation that passes nothing but texture coordinates into the vertex shader and transforms the looked-up normal by a uniform matrix to get it into world space.

In the fixed-function pipeline, normal mapping was done using a special texture mode that would compute the dot product between the looked-up normal and an interpolated vector. If the second vector is the light direction, you get diffuse shading, and if it's the half vector, you can compute specular shading. This mode is called "DOT3" so you will sometimes hear normal mapping called "DOT3 bump mapping."

Deriving maps from geometry

Very often, displacement, bump, and normal maps are used as modeling primitives: they are painted, derived from the color texture, pasted in from a library, or grabbed from any other convenient source and used to add detail that is not present in the geometric model.

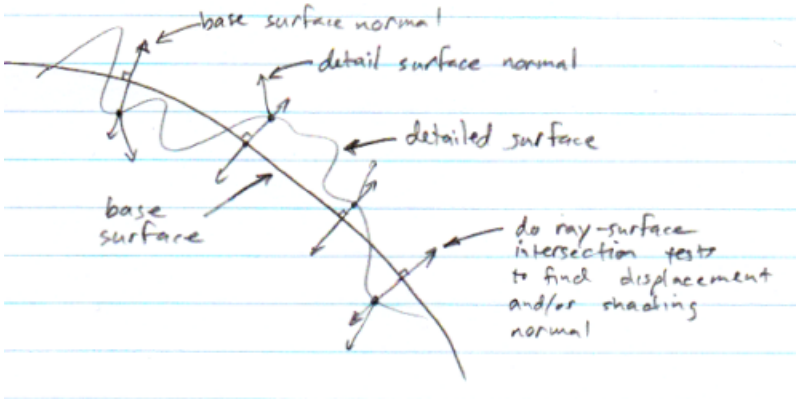
Sometimes, though, you do actually have a detailed model that you'd like to display, but you only have the polygon budget to display a simplified mesh. In this case you already have the detail information; you just want to turn it into the appropriate type of texture map so that you can fake the more complex geometry cheaply.

There are various approaches for this, but the basic steps are:

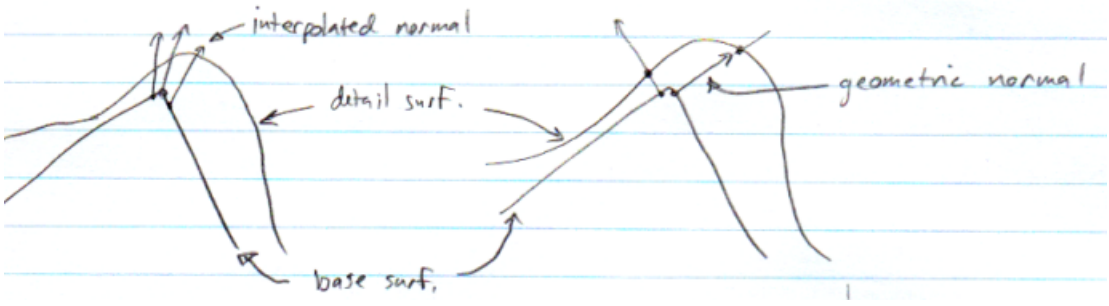
Establish a correspondence between the base surface and the detail surface.

For each texel, store information about the corresponding point on the detail surface.

For deriving a displacement map, the obvious thing to do is to find a point on the detail surface that you can reach by displacing from the base surface along the base surface normal. This is basically a ray tracing operation:

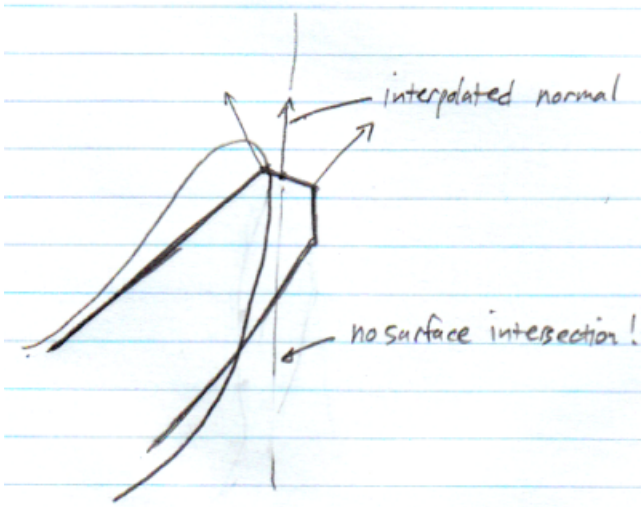


The normal you use in this computation should be the normal you'll use for displacement—which, for a coarse triangle mesh, could be quite different than the geometric normal:

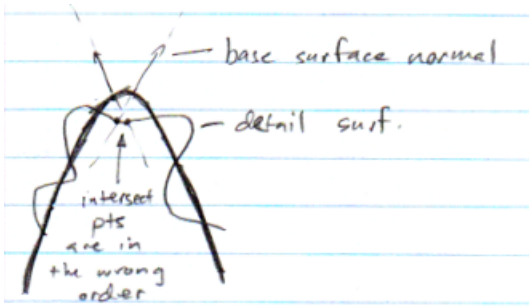


Note the large jump in displacement values on the right—this will be bad news if you use the resulting map for displacement mapping (with interpolated normals) or bump mapping (at all). The normal vectors will also jump suddenly, if you store them for a normal map. On the left everything is nice and continuous.

Note that this intersection might fail, in cases where the base surface is not such a great approximation:



You have to be ready for this case somehow; leaving a "void" value in the map and interpolating across the hole afterwards might be a reasonable approach. Something more subtle that can also happen is the surface folding back on itself:



This is harder to detect, but will cause bad artifacts in a displaced surface. For a bump or normal map it might not be so obvious this is happening.

If there is some other correspondence between the two surfaces available, that correspondence can be used for normal mapping. The usual source of such correspondence information is some kind of mesh simplification (more on these methods later in the course). Many mesh simplification methods are therefore set up to take in a detailed mesh and spit out a simplified mesh with a normal map designed to make it look like the detailed mesh. This can be quite effective.

Height fields

A common use of displacement is for terrain, where the base surface can be a plane and the displacement is an elevation map. These elevation maps are a standard kind of geographic data (if you're interested in real-world environments, as in a flight simulator) or can be synthesized by simple fractal approaches that produce quite convincing hills and mountains. It's also fairly easy to synthesize very convincing height fields for water surfaces ranging from swimming pools to high seas, and with the appropriate shader and environment map they look great.

Geometry images

If you take a displacement map and, instead of storing a displacement relative to the base surface, just store the position you want in 3D, the geometry of the base surface no longer matters; all the geometric information is in the texture. This is a geometry image.