CS 314: Dealing with Normals

Robert Bridson

October 28, 2008

1 Computing Normals

While discussed shading, we've often used surface normals—but apart from the case of a plane where we actually required the user to specify the normal, we haven't properly talked about how to compute them.

There are effectively two ways to do this. If the surface is specified *implicitly*, i.e. as the set of 3D points $\vec{x}$ satisfying an equation $F(\vec{x}) = 0$, then the gradient of the defining function F must be orthogonal to the surface. We can normalize this gradient (i.e. scale it to be unit-length) and get the surface normal:

$$\hat{n} = \frac{\nabla F}{\|\nabla F\|}$$

For the case of a sphere of radius r centred at $\vec{p}$, we used $F(\vec{x}) = \|\vec{x} - \vec{p}\|^2 - r^2$: you can verify that the normal is just $(\vec{x}-\vec{p})/\|\vec{x}-\vec{p}\|$. For a plane going through point $\vec{p}$ with normal $\hat{n}$, we used $F(\vec{x}) = (\vec{x}-\vec{p})\cdot\hat{n}$, and you can verify that the gradient of this F is indeed just $\hat{n}$.

For an explicit surface, where we have a formula which generates the points on the surface, we need a different approach. The main example we've seen is triangles: the points on a triangle can be generated by non-negative weighted averages of the vertices (i.e. using barycentric coordinates). Here we find two linearly independent tangent vectors $\vec{u}$ and $\vec{v}$ at the desired point on the surface: for a triangle two of the edges will do, such as

$$\begin{aligned} \vec{u} &= \vec{x}_1 - \vec{x}_0 \\ \vec{v} &= \vec{x}_2 - \vec{x}_0 \end{aligned}$$

The cross-product of these two tangent vectors must be orthogonal to both of them, and hence the surface itself. Normalizing the cross-product gives the unit length normal:

$$\hat{n} = \frac{\vec{u} \times \vec{v}}{\|\vec{u} \times \vec{v}\|}$$

2 Faking Smooth Surfaces with Fake Normals

Triangle meshes play a dominant role in computer graphics thanks to their simplicity yet flexibility in approximating just about any shape. We didn't establish their approximation ability in a rigourous mathematical sense, but there are theorems that basically say with enough triangles we can always approximate a smooth surface to within any given error, where error is defined as the maximum distance between the triangle mesh and the smooth surface.

However, this isn't the only measure of error we care about. In particular, no matter how many triangles we use, a triangle mesh always consists of flat faces with sharp edges and corners: the normal jumps discontinuously from one constant value to another as we look over the mesh. Almost all the shading formulas are computed using the normal, and thus the colours we see in the final image will jump discontinuously from one triangle to the next—instead of varying smoothly like they should for a smooth surface[1] This error is very obvious to the human eye since we're good at noticing edges, where there's a high contrast.

On the other hand, just because we approximate the geometry with a triangle mesh doesn't mean we have to approximate the shading with the same triangles. At least conceptually you could imagine using the true smooth normals when evaluating shading formulas, while still using triangles for geometric calculations (rasterizing or ray intersection). For example, if we use a triangle mesh approximation for a sphere centred at point $\vec{p}$, we can still evaluate the outward normal used in shading at surface point $\vec{x}$ as

$$\hat{n}(\vec{x}) = \frac{\vec{x} - \vec{p}}{\|\vec{x} - \vec{p}\|}$$

This gives much superior results, modulo one issue we'll touch on below, but obviously isn't always a possibility: for some artist-designed meshes in particular we might have no idea what the "exact" normals are.

We hit this discontinuity issue before, in 2D rasterization of triangles long before we discussed shading or other 3D concepts. We had pointed out that using a constant colour per triangle in the rasterizer can't produce smoothly-varying images, but that with barycentric coordinates we can linearly interpolate colours from the vertices of the triangles—and the result will be continuous across the image. For shading this leads to **Phong normal interpolation**: we can specify normal vectors at the vertices of a mesh, and linearly interpolate the normal at any point in between using barycentric coordinates. Note that linearly interpolating between two or more unit-length vectors doesn't usually give a unit-length

[1] Ignoring, of course, discontinuities due to shadows or silhouettes.

result, so the interpolated normal must be renormalized:

$$\hat{n} = \frac{\alpha\hat{n}_0 + \beta\hat{n}_1 + \gamma\hat{n}_2}{\|\alpha\hat{n}_0 + \beta\hat{n}_1 + \gamma\hat{n}_2\|}$$

These vertex normals might be specified by an artist or a formula, or can themselves be estimated from the triangle mesh by averaging the normals of the incident triangles for each vertex.

This scheme works extremely well for how we've handled diffuse and glossy shading. However, for mirror reflections (or more advance path-tracing treatment of diffuse and glossy materials) we sometimes do hit a problem stemming from the fact that the shading normal is inconsistent with the rendered geometry. In particular, it's possible for an incoming ray that hits a triangle to be reflected—using the shading normal which is *not* exactly orthogonal to the triangle—to the wrong side of the triangle: instead of bouncing off the surface, the reflection erroneously goes inside the surface. This shows up usually as anomolous black pixels in the rendered image, typically near the silhouette edges where the incoming ray is at a very oblique angle to the geometry.

Speaking of silhouette edges, the other noticeable glitch in this normal fakery is that although the interior of an object will look properly smooth, the geometry of the silhouette is still determined by the triangle mesh and thus will not be smooth. If the mesh isn't tesselated finely enough, this might be visible and disturbing to the viewer. In fact, if the mesh is really coarse (has a small number of big triangles), the viewer will probably spot something wrong in the interior: the difference between linearly interpolated normals and truly smooth normals will be obvious.

For real-time rasterization, the extra work implied by linearly interpolating the normal at each pixel and then re-evaluating the full shading model can be too much. A further simplification is often used—in particular, in OpenGL—called **Gouraud shading**, where the shading formula is only evalated at vertices and the resulting colours are then linearly interpolated across the triangles. For diffuse-shaded objects, the difference between this and Phong normal interpolation is pretty subtle: linearly interpolating the colours versus the normals amounts is equivalent to failing to renormalize normals after interpolation. The visible difference is that the brightest and darkest (or strongest and weakest) colours can only appear at the triangle vertices with Gouraud shading, whereas they can appear naturally anywhere on the model (where the interpolated surface normal best aligns with the lighting direction) for Phong normal interpolation.

For very glossy objects, Gouraud shading is significantly worse, however: the characteristic highlight around the mirror reflection direction will either appear (but perhaps be too big) if it happens to lie on a triangle vertex, or vanish altogether if it falls between vertices. With Phong normal interpolation, the highlight is always present and reasonably well approximated.

3 Transforming Normals

Having introduced vertex normals as extra data on the mesh, we do run into an immediate problem: if we transform an object (translating it, rotating it, scaling it, etc.) we will obviously have to do something with the normals as well. In some sense, normal vectors are directions, so we might expect to be able to use the same rule we did for direction vectors when transforming rays, but it's actually a bit more complicated than that.

We can start by looking at a few special cases to see how things might differ:

- For a translation, just like directions, we don't want normal vectors to be altered.
- For a rotation, just like directions, we want normal vectors to be rotated exactly the same as points.

This is our first clue we want to use a homogeneous coordinate of 0, like directions, which is equivalent to just using the upper-left 3×3 submatrix of the transformation on the 3D representation of the vectors. However, as we saw in diagrams in class:

- For a positive x-shear, unlike directions, we want to instead perform a negative y-shear on the normal.
- For a positive y-shear, unlike directions, we want to instead perform a negative x-shear on the normal.
- For a scaling along one axis, unlike directions, we want to scale the normal along other axes.

What's going on?

Normals aren't the same as directions: they don't extend from one point to another point which gets transformed with the rest of the model. On the other hand, *tangent* vectors (say vectors that go from one vertex of a triangle to another vertex) are exactly direction vectors, and would be transformed as such. Normal vectors instead are constrained to always be orthogonal to the tangent vectors. This is true even for an implicitly defined surface where we didn't use tangent vectors in computing the normals!

Say A is the upper-left 3×3 submatrix of the transformation matrix we are using. If $\vec{u}$ and $\vec{v}$ are tangent vectors, then they will be transformed to $A\vec{u}$ and $A\vec{v}$. The original normal vector $\hat{n}$ is orthogonal to both $\vec{u}$ and $\vec{v}$, i.e. the dot-products are zero, which we can express in more matrix-friendly terms using the transpose:

$$\vec{u}^T \hat{n} = 0, \qquad \vec{v}^T \hat{n} = 0$$

(Remember we generally think of vectors as being arranged in columns.) We want this to be true after the transformation: if we assume the transformed normal is $B\hat{n}$ for some 3×3 matrix B, we get

$$(A\vec{u})^T(B\hat{n}) = 0, \qquad (A\vec{v}^T)(B\hat{n}) = 0$$

If we expand this out using the rules for multiplication and transposes, we get:

$$\vec{u}^T A^T B\hat{n} = 0, \qquad \vec{v}^T A^T B\hat{n} = 0$$

Using the fact $\vec{u}^T\hat{n} = 0$ and $\vec{v}^T\hat{n} = 0$, this will be true if $A^T B$ is just the identity matrix. This would mean:

$$B = (A^T)^{-1}$$

which we often abbreviate as $B = A^{-T}$. This is how we must transform normals then: multiply by the inverse transpose of the upper-left 3×3 submatrix. In general, we also have to finish by renormalizing the result (dividing by its length) since this inverse transpose operation only guarantees orthogonality at the end of the day, not whether the output stays unit-length.

(Also please note: just like in the case of directions we are not tackling perspective projections where there is an additional homogenization step.)

As a good exercise you can check that this matches what we saw before for examples of transformations. For a translation, A is just the identity, and thus A^{-T} is also the identity. For a rotation, A is an orthogonal matrix whose transpose is its inverse; $A^{-T} = A$, giving the expected result. For the shears, taking the inverse transpose does flip the axis of shearing and the sign of shearing, as expected. For scaling, the inverse transpose uses the reciprocals of the scale factors.