Generating Vertex Normals

Author: Max Wagner, mwagner@digipen.edu

Pre-Reqs

Familiarity with terms and concepts from Linear Algebra, including the ideas behind vectors and vector spaces.

Notation

Vectors AND points in bold, i.e. $\boldsymbol{u}$ or $\boldsymbol{n}$. I leave it to the reader to discern from the context whether the object is a point or a vector. Scalars will be weighted normally, that is un-bolded. Often I will refer to a vector $\boldsymbol{u}$, and then shortly afterward refer to its components as u_x, u_y, u_z, etc.

Vertex Normals vs. Surface Normals

We all know what a surface normal is (if not, it's the normal to the plane that contains the surface). So how can a vertex (i.e., a point), have a normal? Strictly speaking, it can't. What vertex normals provide is a means of simulating smoother surfaces during lighting calculations when using procedures such as Phong or Gouraud shading (sometimes just called Smooth shading). Imagine a polygonal mesh of a human: technically, this mesh is just a bunch of flat polygons. But really, this mesh is simulating the smooth surface of a human body. If all pixels within a polygon were lit identically, the "flatness" of each polygon would be starkly obvious; but by using vertex normals, we can light each vertex on a triangle differently, thus causing a smoother appearance. The trick is to generate these vertex normals so that they actually enhance this smooth appearance.

The Intuition behind Vertex Normals

Let's take a close look at the lighting process involved in traditional polygonal mesh, scan-line rasterization techniques (such as those employed on graphics cards and in software renderers for real-time applications). Lighting will be applied per-vertex, where the dot product of the surface normal and the light's direction will be used to modulate the intensity of the light's color (this aspect of the light model is called the Diffuse lighting term). This modulated light color will then be added to a pre-computed, constant vertex color. Other terms are also possible (specular, ambient) as well. Thus, if we are using triangles, all three vertices of a triangle will have an identical dot product between the light direction and surface normal vectors[1]. Using the same normal for all three vertices assumes that the actual surface we are simulating is flat, but we know this is not the case. It's as though we are using a single sampling point to compute the normal for all three vertices. But imagine we could actually sample the "true" surface at the vertices

[1] Alternatively, if point light sources are being used, there will be a small amount of variation in the dot product, as the light's direction will be computed per vertex as the difference between the vertex's position and the light's position. Nonetheless, given small triangles, the diffuse term will remain nearly identical for all three vertices.

themselves; then we would surely get more variation amongst the vertex normals, in turn creating a (smooth) variation in the diffuse lighting terms. But how can we sample the "true" surface?

Unfortunately, we can't. However, we can come close, using the observation that a vertex lies at the intersection of multiple triangles. Hence, instead of using the surface normal of the one triangle of which the vertex is a part, we can detect the adjacent triangles (i.e., the other triangles that share the same vertex), and perform a kind of "averaging" of all adjacent surface normals. This provides the intuition behind the notion of a vertex normal, as there is no strict definition (there are multiple techniques for computing vertex normals, and no one is "correct").

The Algorithm

Now that we have an idea of what the vertex normal, let's devise an algorithm to compute them. After devising a standard algorithm, I will discuss a few variations that can be incorporated for a nicer appearance.

The idea is that we want to average the normals of all polygons that are adjacent to a given vertex. Recall that in standard rasterization techniques, we have access to a list of triangles; this list will necessarily contain duplicate vertices, as a given vertex is included for each triangle of which it is a part. This duplication is often alleviated using indexed triangle lists, but to keep the argument simple, let us assume we are using an un-indexed list, where the i^{th} triangle in our vertex list is defined using the three vertices' positions *v[i*3].p* , *v[i*3+1].p* , and *v[i*3+2].p*. Then our task is to find, for each vertex, all the triangles that share that vertex. Each time we find a triangle that shares the current vertex, we compute the triangle's surface normal, and add it into a running tally. When we are done, we normalize the vertex normal, and store it with the vertex.

In pseudocode:

```
struct Vertex { Position p,  Normal n }
VertexList v

for each vertex i in VertexList v
      n ← Zero Vector
      for each triangle j that shares i^th vertex
            n ← n + Normalize(Normal(v, j))
      end for
      v[i].n ← Normalize(n)
end for
```

The routine `Normal(v, j)` is assumed to return the outwardly facing normal to the triangle using the index scheme described above.

This pseudocode is pretty simple, though not particularly fast for a large number of vertices ($O(n^2)$).

One problem with the existing approach is that, for surfaces that were actually *not* meant to be smooth (i.e., a mesh of a cube), the result will appear strange; at the corners of a sharp crease, the lighting will appear smooth, which is clearly not what we want.

One approach to combat this is to use a threshold when deciding whether to include a triangle's normal in the averaging. Specifically, when examining the i^{th} vertex, we cache the surface normal of the triangle to which the vertex technically belongs (as opposed to the triangles which share a copy of the vertex elsewhere in the triangle list). Then, when upon finding another triangle that shares the vertex, we compute the dot product between the cached normal and the current triangle's normal; if this dot product is less than some threshold (call it ε, usually a little above 0), then we do not include the current triangle's normal in the average. How does this work? Recall that the dot product corresponds to the cosine of the angle between two vectors. What we are effectively saying is, if the angle between two surfaces is greater than ε, don't average them.

The new pseudocode:

```
struct Vertex { Position p,  Normal n }
VertexList v
epsilon e

for each vertex i in VertexList v
      n ← Zero Vector
      m ← Normalize(Normal(v, i%3))
      for each triangle j that shares i^th vertex
            q ← Normalize(Normal(v, j))
            if DotProduct(q, m) > e
                  n ← n + q
            end if
      end for
      v[i].n ← Normalize(n)
end for
```

Another possible improvement to our algorithm comes from examining the fact that, while a vertex lies at the intersection of multiple triangles, these triangles are not necessarily of equal size. Some people suggest using the area of the triangle as a weighting factor when including a surface normal in the averaging process. The pseudocode is hardly changed to incorporate this; we just require a new routine to compute the area of a triangle, given the three vertices.

The final version looks like:

```
struct Vertex { Position p,  Normal n }
VertexList v
epsilon e
```

```
for each vertex i in VertexList v
      n ← Zero Vector
      m ← Normalize(Normal(v, i%3))
      for each triangle j that shares i^th vertex
            q ← Normalize(Normal(v, j))
            w ← Area(v, j)
            if DotProduct(q, m) > e
                  n ← n + w*q
            end if
      end for
      v[i].n ← Normalize(n)
end for
```

Applications to Real-Time Generation

So what about a deformable mesh? For a rigid body of course, where the vertices *do not* move with respect to each other, we can place a vertex normal in the world using the inverse transpose of the world transformation matrix. But this won't work for vertices whose positions change with respect to the other vertices on the mesh. This is because the planes governing the generation of the normals change with respect to each other.

We clearly can't just use the above algorithm *as is*, because it's too slow for a mesh with many polygons. We can however do a pre-process which will allow us to more quickly update the normals. Consider that generating a vertex normal for a given vertex requires knowledge of the adjacent polygons; finding these adjacent polygons in an arbitrary triangle list requires looping through the entire list; this is slooow ($O(n^2)$). But if we do this part once, as a pre-process, and associate with each vertex a list of adjacent polygons, then computing all the vertex normals can be performed in linear time. You simply walk your vertex list, then iterate through the (pre-computed) list of adjacent polygons, summing their normals as above, before finally normalizing the normal.

However, in newer architectures utilizing vertex shaders, often we want to initialize our vertex buffer once, and perform all vertex modification through shaders (i.e., on the GPU, not the CPU). Re-computing the vertex normals as described above using an adjacency list requires relative addressing (array indexing) and access to a large amount of data (all the vertices at once), two features that aren't readily supported by most vertex shader languages (though even this is changing as shaders become more flexible). Hence, this computation would happen on the CPU in your normal C/C++ code, and then you would feed these dynamically updated normals to your graphics card by locking down your vertex buffer and copying the new data in (potentially causing a stall in the pipeline).

Surface Approximation

So how can we get around this? The answer is highly dependent on the needs and context of the application. However, if you're willing to sacrifice some accuracy in exchange for an extremely rapid and easy-to-implement alternative, then I propose a very simple approximation that is well suited to simple objects and vertex shader implementations.

If we can find an analytic function that sufficiently approximates our surface, then taking the partial derivative of this function will give us the normal of the surface at a given point, i.e.:

$$\nabla f(x,y,z) = \left(\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}\right) = \text{surface normal at the point on the surface } (x,y,z)$$

where it is assumed that $f(x,y,z)=0$ for a point on the surface.

An analytic function is easy to express in a vertex shader; notice that the only dependence of the function is the point on the surface, i.e. the vertex position; but this is the natural input of a vertex shader. Of course, the reason we use polygonal meshes is because there is rarely an analytic function that can describe the sorts of shapes we have in games or other applications. Again, I stress the word *approximation*.

At the extreme, we can say that our object is represented by a sphere. Taking the partial derivative of the equation for a sphere centered at an arbitrary position yields:

$$f = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r^2 = 0$$

$$\nabla f = \left(\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}\right) = (2(x - C_x), 2(y - C_y), 2(z - C_z))$$

Since we are going to normalize the normal anyway, we can scale the above result by 0.5; notice that this yields just the point on the surface minus the center of the sphere! Of course, this is what we would expect intuitively, that the surface normal of a sphere would be in the direction along the line from the point on the sphere to the center. In our vertex shader, now all we need as extra input is the "center" of our object; calculating the normal then involves a vector subtraction and normalization.

An extension of this method could account for concave objects by performing a pre-process of the vertex list and associating with each vertex a sign of negative or positive one. This sign represents whether the vertex normal should point toward or away from the center. Computing this sign would depend on some heuristic, such as performing ray casts from the point to the center of the object and determining how many times the ray enters or exits the surface.

Surfaces using Regular Grids of Vertices

Other times, we may have surfaces defined using a regular grid of vertices, such as terrain or a piece of cloth. In this case, we can use the original "correct" algorithm without an adjacency list. This is because we can generate in constant time the neighbors of a given face, simply using the (i,j)-th index of a given vertex. To minimize redundant polygon normal computations, first we would run through the entire grid and generate each polygon normal; then we would run through all the vertices of the grid, and access the adjacent polygons, summing their normals, then normalizing. This technique is not

suitable for shaders, but it is fast and very accurate; in cases where the accuracy is necessary (such as a piece of cloth, or a deformable terrain), a surface approximation will likely not provide an adequate visualization; thus we resort to this grid based approach.