Lecture 21: Shading

put your trust in my shadow. Judges 9:15

1. Polygonal Models

Polygonal models are one of the most common representations for geometry in Computer Graphics. Polygonal models are popular because the underlying surface of each polygon is a plane and planes are the easiest surfaces to analyze: the normal vector along a plane is constant, and the points on a plane satisfy a linear equation. Thus rendering algorithms are particularly simple for polygonal models.

But models consisting solely of polygons also have severe limitations because planar polygons cannot model smooth curved surfaces. To overcome this deficiency, many small polygons are needed to simulate curved shapes. But even with lots of small polygons, the human eye is adept at seeing the edges between adjacent polygons. Thus surfaces modeled by polygons do not look smooth. *The purpose of shading algorithms is to smooth the appearance of polygonal models by reducing the impact of sharp edges in order to give the eye the impression of a smooth curved surface.*

We shall study three different shading algorithms: uniform shading, Gouraud shading, and Phong shading. As in ray tracing, we will consider three illumination models: ambient reflection, diffuse reflection, and specular reflection. Since a computer model of a complicated curved shape may contain millions of polygons, the emphasis in shading algorithms is on speed. A good shading algorithm must be fast and make the model look smooth; shading algorithms are not necessarily intended to be physically accurate models of the behavior of light.

Uniform shading is fast, but makes no attempt to smooth the appearance of the model. Thus we introduce uniform shading only to compare the results with Gouraud and Phong shading. Gouraud shading smooths the appearance of the model by linearly interpolating intensities along scan lines; Phong shading simulates curved surfaces by linearly interpolating normal vectors along scan lines.

To speed up our computations, we shall make several simplifying assumptions. In addition to assuming that curved surfaces are approximated by planar polygons, we shall assume that all light sources are point light sources and that both the light sources and the eye are located far from the model (essentially at infinity). These assumptions mean that the vectors to each of the light sources as well as the vector to the eye are constant along each polygon. Together with the fact that the normal vector is constant along a polygon, these assumptions help to simplify the computation of diffuse and specular reflections.

1.1 Newell's Formula for the Normal to a Polygon. Intensity calculations require normal vectors. Polygons are typically represented by their vertices, so to calculate the intensity of light on a polygon, we need to compute the normal vector from the vertices. Newell's formula is a numerically robust way to compute the normal vector of a polygon from its vertices.

Naively, we could compute the normal vector N to a polygon from its vertices $P_0,\ldots,P_n$ by selecting two adjacent edge vectors $P_{i+1}-P_i, P_{i+2}-P_{i+1}$ and setting

$$N = (P_{i+1}-P_i)\times(P_{i+2}-P_{i+1}),$$

since this cross product is perpendicular to the plane of the polygon. The problem with this approach is that two of the vertices might be very close together so one of the edge vectors $P_{i+1}-P_i, P_{i+2}-P_{i+1}$ might be very short; worse yet the three points P_i,P_{i+1},P_{i+2} might be collinear or almost collinear. In each of these cases, the computation of the unit normal $N/|N|$ is numerically unstable.

Newell's formula overcomes this difficulty by using all of the vertices to calculate the normal to the polygon. (For a derivation of Newell's formula, see Lecture 11, Section 4.2.2.)

Newell's Formula

$$N = \sum_{k=0}^{n} P_k \times P_{k+1} \qquad \{P_{n+1} = P_0\} \tag{1.1}$$

Recall from Lecture 11 that the magnitude of the normal vector in Newell's formula is related to the area of the associated polygon. In fact,

$$|N| = 2 \times Area(Polygon).$$

Thus Newell's formula avoids degeneracies due to collinear vertices.

2. Uniform Shading

In *uniform shading* or *flat shading* no attempt is made to smooth the appearance of a polygonal model. Each individual polygon is displayed using the intensity formula

$$I_{uniform} = \underbrace{I_a k_a}_{ambient} + \underbrace{I_p k_d (L \bullet N)}_{diffuse} + \underbrace{I_p k_s (R \bullet V)^n}_{specular}, \tag{2.1}$$

where

- N is the unit vector normal to the polygon;
- L is the unit vector pointing from the polygon to the light source;
- V is the unit vector pointing from the polygon to the eye;
- $R = 2(L \bullet N)N - L$ is the direction of the light vector L after it bounces off the polygon.

By assumption, the vectors N,L,R,V are constant along each polygon, so the intensity $I_{uniform}$ is also constant along each polygon (see Figure 1). For multiple light sources, we add the contributions of the individual light sources. Thus uniform shading is fast, since we need to compute only one intensity for each polygon.

Unfortunately, with uniform shading individual polygons are highly visible because the vectors N,L,R,V and hence the intensity $I_{uniform}$ is different for each polygon. Therefore, discontinuities in intensity appear along the edges of the polygons, and these discontinuities are heightened by the physiology of the eye. Discontinuities in intensity along polygon edges are called *Mach bands*. The purpose of the shading algorithms that we shall study next is to eliminate these Mach bands.

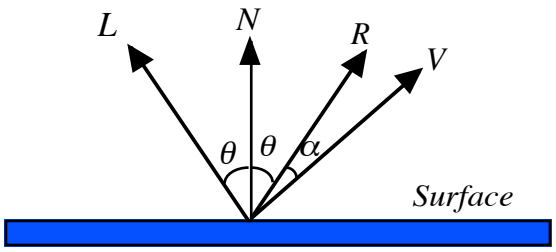

Figure 1: The vectors N,L,R,V are constant along each polygon

3. Gouraud Shading

The purpose of Gouraud shading is to eliminate Mach bands; the method employed by Gouraud shading is linear interpolation of intensities. The underlying strategy here for each polygon is to compute the intensities for the pixels in three steps:

i. First, calculate the intensities at the individual vertices of the polygon.
ii. Next, interpolate these vertex intensities along the edges of the polygon.
iii. Finally, interpolate these edge intensities along scan lines to the pixels in the interior of the polygon (see Figure 2).

We shall now elaborate on each of these steps in turn.

To compute the intensity at a vertex, we need to know the unit normal vector at the vertex. Since each vertex may belong to many polygons, we first use Newell's formula (Equation (1.1)) to calculate the unit normal for each polygon containing the vertex. We then calculate the unit normal vector at the vertex by averaging the unit normals of all the polygons containing the vertex:

$$N_{vertex} = \frac{\sum_{vertex \in Polygon} N_{polygon}}{\left| \sum_{vertex \in Polygon} N_{polygon} \right|}. \qquad (3.1)$$

Finally, we apply Equation (2.1) to compute the intensity at the vertex.

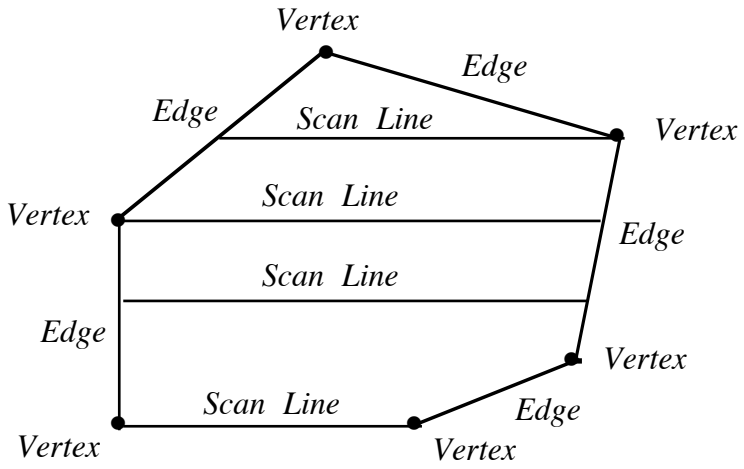

Figure 2: Gouraud Shading. First compute the intensity at each vertex. Then interpolate these vertex intensities along each edge. Finally interpolate these edge intensities to the pixels in the interior of the polygon by interpolating edge intensities along scan lines.

Once we have the intensities at the vertices, we can apply linear interpolation to compute intensities first along edges and then along scan lines in the following fashion. Suppose that we know the intensities I_1, I_2 at two points P_1, P_2. The parametric equation of the line through P_1, P_2 is

$$L(t) = (1-t)P_1 + tP_2 = P_1 + t(P_2 - P_1)$$

Linear interpolation means that we compute the intensity I at a parameter t by the analogous formula

$$I(t) = (1-t)I_1 + tI_2 = I_1 + t(I_2 - I_1).$$

To speed up the computation of intensity, we shall reduce this computation to one addition per pixel by computing $I(t)$ incrementally. For the line

$$L(t + \Delta t) = P_1 + (t + \Delta t)(P_2 - P_1) = P_1 + t(P_2 - P_1) + \Delta t(P_2 - P_1)$$

$$L(t) = P_1 + t(P_2 - P_1),$$

so

$$\Delta L = \Delta t(P_2 - P_1) \qquad (3.2)$$

$$L(t + \Delta t) = L(t) + \Delta L.$$

Thus

$$L_{next} = L_{current} + \Delta L. \qquad (3.3)$$

In terms of coordinates, Equation (3.2) for ΔL is really three equations, one for each coordinate:

$$\Delta x = \Delta t(x_2 - x_1) \qquad \Delta y = \Delta t(y_2 - y_1) \qquad \Delta z = \Delta t(z_2 - z_1). \qquad (3.4)$$

Similarly, for intensity

$$I(t + \Delta t) = I_1 + (t + \Delta t)(I_2 - I_1) = I_1 + t(I_2 - I_1) + \Delta t(I_2 - I_1)$$

$$I(t) = I_1 + t(I_2 - I_1),$$

so

$$\Delta I = \Delta t(I_2 - I_1)$$
$$I(t + \Delta t) = I(t) + \Delta I. \tag{3.5}$$

Thus

$$I_{next} = I_{current} + \Delta I. \tag{3.6}$$

Since we know I_1 and I_2, if we know Δt, then we can compute ΔI and we can update I_{next} by a single addition per pixel.

Now consider what happens in two cases: along a scan line and along an edge. Along a scan line, when we move to the next pixel $\Delta x = 1$ (see Figure 3). Since by Equation (3.4) $\Delta x = \Delta t(x_2 - x_1)$, it follows that along a scan line

$$\Delta t = 1/(x_2 - x_1)$$
$$\Delta I = (I_2 - I_1)\Delta t = (I_2 - I_1)/(x_2 - x_1).$$

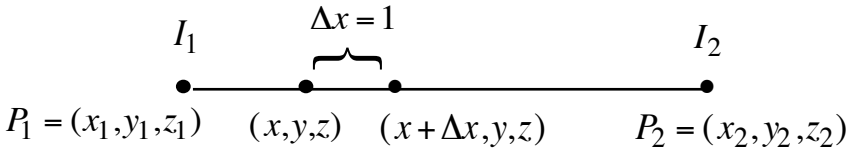

Figure 3: Moving to the next pixel along a scan line: $\Delta x = 1$.

To get to the next scan line, we move along an edge. When we move to the next scan line $\Delta y = 1$ (see Figure 4). Since by Equation (3.4) $\Delta y = \Delta t(y_2 - y_1)$, it follows that along an edge

$$\Delta t = 1/(y_2 - y_1)$$
$$\Delta I = (I_2 - I_1)\Delta t = (I_2 - I_1)/(y_2 - y_1).$$

Thus in each case, we can reduce the computation of intensity to one addition per pixel.

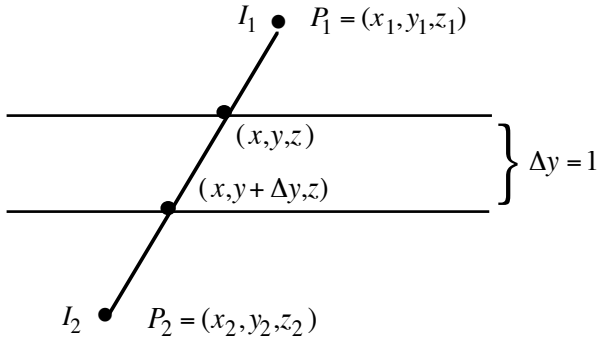

Figure 4: Moving to the next scan line: $\Delta y = 1$.

Notice that for adjacent polygons, intensities necessarily agree along common edges because the intensities agree at common vertices. Moreover, on both sides of an edge, the value of the intensity near the edge is close to the value of the intensity along the edge. Thus linear interpolation of intensities along scan lines blurs the difference in intensities between adjacent polygons. This blurring eliminates Mach bands and provides the appearance of smooth curved surfaces.

The Gouraud shading algorithm is an example of a scan line algorithm. A *scan line algorithm* is an algorithm that evaluates values at pixels scan line by scan line, taking advantage of the natural coordinate system induced by the orientation of the scan lines. One problem that arises with a scan line approach to shading is that the method is coordinate dependent -- that is, the intensities depend not only on the lighting, but also on the orientation of the polygons relative to the coordinate system. Consider, for example, the square in Figure 5. Oriented one way, half the square is white, whereas if the coordinate systems is rotated by 90° relative to the polygon, the same half of the square is gray.

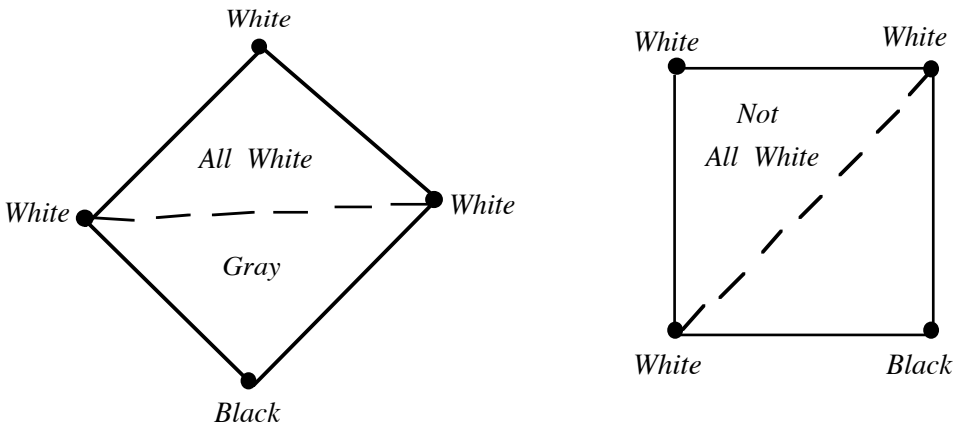

Figure 5: The same square oriented differently relative to the coordinate system has different shading.

One way to fix this problem is to subdivide all the polygons into triangles. Scan line algorithms for triangles are independent of the orientation of the triangles relative to the coordinate system. We can establish this independence in the following manner. Consider $\Delta P_1P_2P_3$ in Figure 6. Since P_1, P_2, P_3 are not collinear, the vectors $P_3 - P_1, P_3 - P_2$ span the plane. Therefore any point P inside the triangle can be written uniquely as

$$P = P_1 + \beta_2(P_2 - P_1) + \beta_3(P_3 - P_1) = \beta_1 P_1 + \beta_2 P_2 + \beta_3 P_3.$$

where $\beta_1 + \beta_2 + \beta_3 = 1$. The values $\beta_1, \beta_2, \beta_3$ are called the *barycentric coordinates* (see Lecture 4, Exercise 24) of the point P. We can compute the barycentric coordinates $\beta_1, \beta_2, \beta_3$ by performing linear interpolation twice: first along the edges of the triangle and then along a horizontal line (see Figure 6). Using this approach, we find that

$$P = \underbrace{\big((1-u)(1-t)+u(1-s)\big)}_{\beta_1} P_1 + \underbrace{\big((1-u)t\big)}_{\beta_2} P_2 + \underbrace{(us)}_{\beta_3} P_3 .$$

But linear interpolations of intensities is performed in exactly the same way. Thus

$$I = \underbrace{\big((1-u)(1-t)+u(1-s)\big)}_{\beta_1} I_1 + \underbrace{\big((1-u)t\big)}_{\beta_2} I_2 + \underbrace{(us)}_{\beta_3} I_3$$

Since the barycentric coordinates $\beta_1, \beta_2, \beta_3$ are unique, this result is independent of the orientation of the triangle.

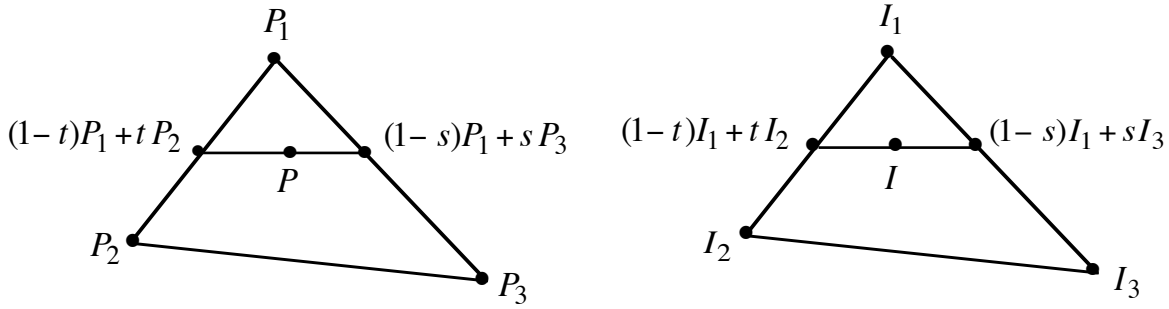

Figure 6: Linear interpolation for triangles: for points along edges (left) and for intensities (right).

Nevertheless, even though for triangles the pixel intensities are independent of the orientation of the triangles relative to the coordinate system, the pixel intensities for polygons still depend on exactly how we subdivide the polygons into triangles; different subdivisions will generate different intensities for the same pixels.

4. Phong Shading

Gouraud shading reduces the visibility of polygonal edges by blurring the intensity across the boundaries of adjacent polygons. But blurring these intensities also blurs specular reflections. To reproduce sharp specular reflections, we need a better way to simulate curved surfaces.

Phong shading simulates curved surfaces by interpolating normal vectors instead of intensities. Normal vectors vary along a curved surface, so Phong shading provides a better approximation to the intensity for curved surfaces modeled by planar polygons. We begin by presenting a naive version of Phong shading; more sophisticated enhancements will be presented in subsequent subsections.

4.1 Naive Phong Shading. The underlying strategy of Phong shading is similar to the underlying strategy for Gouraud shading, except that intensities are replaced by normal vectors. The normal vectors for each polygon are computed in three steps:

i. First, calculate the normals at the individual vertices of the polygon.
ii. Next, interpolate these normals along the edges of the polygon.
iii. Finally, interpolate these edge normals along scan lines to the pixels in the interior of the polygon.

Once the normal vectors are computed, the intensities are calculated in the usual manner using Equation (2.1).

As in Gouraud shading, we calculate the unit normal vector at a vertex by Equation (3.1), averaging the unit normals of all the polygons to which the vertex belongs. The unit normal for each polygon is computed by Newell's formula (Equation (1.1)).

Once we have the unit normals at the vertices, we can apply linear interpolation to compute normals first along edges and then along scan lines just as we calculated intensities for Gouraud shading. If N_1, N_2 are the unit normals at the end points P_1, P_2 of the line $P(t) = P_1 + t(P_2 - P_1)$, then using linear interpolation the normal $N(t)$ at the point $P(t)$ is calculated by

$$N(t) = (1-t)N_1 + tN_2 = N_1 + t(N_2 - N_1).$$

To speed up the computation of normal vectors, we reduce this computation to three additions (one for each coordinate) per pixel by computing normals incrementally, using the same approach we used to compute intensities incrementally in Gouraud shading. Replacing intensities by normals in Equation (3.5) leads to

$$\begin{aligned} \Delta N &= \Delta t(N_2 - N_1) \\ N(t + \Delta t) &= N(t) + \Delta N. \end{aligned} \tag{4.1}$$

Thus

$$N_{next} = N_{current} + \Delta N. \tag{4.2}$$

Since we know N_1 and N_2, if we know Δt, then we can compute ΔN, and we can update N_{next} by three additions (one for each coordinate) per pixel. However, to calculate intensities with Equation (2.1), we need not just a normal, but a unit normal. Thus after we calculate $N_{next} = N_{current} + \Delta N$, we need to renormalize -- that is, we must calculate

$$N_{next}^{unit} = \frac{N_{current} + \Delta N}{|N_{current} + \Delta N|}.$$

We compute the normals for Phong shading, just like we computed the intensities for Gouraud shading, in two stages: along scan lines and along polygonal edges. Recall that when we move to the next pixel along a scan line, $\Delta x = 1$ (see Figure 3). Since by Equation (3.4) $\Delta x = \Delta t(x_2 - x_1)$, it follows that along scan lines

$$\begin{aligned} \Delta t &= 1/(x_2 - x_1) \\ \Delta N &= (N_2 - N_1)\Delta t = (N_2 - N_1)/(x_2 - x_1). \end{aligned}$$

To get to the next scan line, we move along an edge. Now recall that when we move to the next scan line, $\Delta y = 1$ (see Figure 4). Since by Equation (3.4) $\Delta y = \Delta t(y_2 - y_1)$, it follows that along edges

$$\Delta t = 1/(y_2 - y_1)$$
$$\Delta N = (N_2 - N_1)\Delta t = (N_2 - N_1)/(y_2 - y_1).$$

Thus naive Phong shading is very similar to Gouraud shading: we simply replace intensities by normal vectors in our calculations along edges and along scan lines. There are, however, two differences: we must renormalize the normal vectors after the incremental computation, and then we must use these renormalized normals in Equation (2.1) to recalculate the intensity at each pixel. These additional computations make naive Phong shading considerably slower than Gouraud shading. In the next three subsections we shall develop more sophisticated techniques to increase the speed of Phong shading.

Phong shading, like Gouraud shading, is coordinate dependent -- that is, the normals depend not only on the polygons, but also on the orientations of the polygons relative to the coordinate system. Once again we can fix this problem by subdividing the polygons into triangles, since scan line algorithms for triangle are independent of the orientation of the triangles relative to the coordinate system. Nevertheless, as with Gouraud shading, the normal vectors and hence the pixel intensities for polygons will still depend on exactly how we subdivide the polygons into triangles; different subdivisions will still generate different intensities for the same pixels.

4.2 Fast Phong Shading and Diffuse Reflection. Linear interpolation for scalars is faster and cheaper than linear interpolation for vectors. For diffuse reflection, we can reduce at least part of the computation for Phong shading from a vector calculation to a scalar calculation.

Recall that the formula for diffuse reflection is

$$I_{diffuse} = I_p k_d (L \bullet N).$$

For Phong shading, the normal vector N is calculated by linear interpolation -- that is,

$$N(t) = (1-t)N_1 + t\,N_2.$$

Thus the unit normal is given by

$$N^{unit}(t) = \frac{(1-t)N_1 + tN_2}{\left|(1-t)N_1 + tN_2\right|}.$$

Substituting this formula for the unit normal into the formula for diffuse reflection yields

$$I_{diffuse} = I_p k_d \frac{(1-t)(L \bullet N_1) + t(L \bullet N_2)}{\left|(1-t)N_1 + t\,N_2\right|}.$$

Now we can consider the numerator and the denominator independently.

We begin with the numerator. Let $J(t) = L \bullet N(t)$. Then the numerator is

$$J(t) = (1-t)(L \bullet N_1) + t(L \bullet N_2) = (1-t)J_1 + tJ_2, \qquad (4.3)$$

where $J_1 = L \bullet N_1$ and $J_2 = L \bullet N_2$. Equation (4.3) for $J(t)$ is just linear interpolation of the scalar valued function $J(t) = L \bullet N(t)$. We can calculate this scalar valued function at each pixel in exactly the same way that we calculate the intensities for Gouraud shading: first calculate $J(t) = L \bullet N(t)$ at the vertices, then interpolate along edges, and finally interpolate along scan lines. This computation is line for line the exact same computation as Gouraud shading: simply apply the computation to the linear function $J(t)$ instead of the intensity $I(t)$.

Now consider the denominator. Let

$$N(t) = (1-t)N_1 + t\,N_2.$$

Then the denominator is given by

$$d(t) = \left|N(t)\right| = \sqrt{N(t) \bullet N(t)}.$$

From naive Phong shading (Equation (4.1)), we know that

$$\Delta N = \Delta t(N_2 - N_1).$$

Therefore by Equation (4.2)

$$N_{next} = N_{current} + \Delta N;$$

hence

$$d_{next} = \sqrt{N_{next} \bullet N_{next}}. \qquad (4.4)$$

We can calculate ΔN and N_{next} quickly using the scan line approach to linear interpolation presented in Section 4.1; what slows us down here is the square root in Equation (4.4).

To find d_{next} quickly, we can apply Newton's method (see Lecture 7, Section 3.1) to calculate this square root. Let

$$F(x) = x^2 - d_{next}^2 = x^2 - N_{next} \bullet N_{next}.$$

Newton's method starts with an initial guess x_0 for a root of $F(x)$ -- the value of d_{next} -- and computes subsequent guesses x_{n+1} from previous guesses x_n by the formula

$$x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)} = x_n - \frac{x_n^2 - d_{next}^2}{2x_n}.$$

Since N changes only by a small amount from pixel to pixel, ΔN is typically small, so we can start the iteration with the initial guess $x_0 = d_{current}$. Because our initial guess is already close to d_{next}, Newton's method will typically converge to a good approximation to the required square root after only a few iterations. Notice that in this implementation, we need to compute both N and d for each pixel. The scan line algorithm is used to update N and $d^2 = N \bullet N$; Newton's method is used only to compute $d = \sqrt{d^2}$.

4.3 Fast Phong Shading and Specular Reflection. Recall that the formula for specular reflection is

$$I_{specular} = I_p k_s (R \bullet V)^n ,$$

where

$$R = 2(L \bullet N)N - L .$$

Substituting this expression for R into the formula for specular reflection yields

$$I_{specular} = I_p k_s \big(2(L \bullet N)(N \bullet V) - (L \bullet V)\big)^n . \qquad (4.5)$$

Now recall again that for Phong shading the normal vector N is calculated by linear interpolation, so

$$N(t) = (1-t)N_1 + t\,N_2$$

and

$$N^{unit}(t) = \frac{(1-t)N_1 + tN_2}{\big|(1-t)N_1 + tN_2\big|}.$$

Substituting this formula for the unit normal into Equation (4.5) for specular reflection yields

$$I_{specular} = I_p k_s \left(\frac{2\big\{(1-t)(L \bullet N_1) + t(L \bullet N_2)\big\}\big\{(1-t)(V \bullet N_1) + t(V \bullet N_2)\big\}}{\big|(1-t)N_1 + t\,N_2\big|^2} - (L \bullet V) \right)^n . \quad (4.6)$$

To analyze this formula, let $J(t) = L \bullet N(t)$ and $K(t) = V \bullet N(t)$. Then the first term in the numerator on the right hand side of Equation (4.6) is the product of the expressions

$$J(t) = (1-t)(L \bullet N_1) + t(L \bullet N_2) = (1-t)J_1 + tJ_2$$

$$K(t) = (1-t)(V \bullet N_1) + t(V \bullet N_2) = (1-t)K_1 + tK_2,$$

where $J_1 = L \bullet N_1$, $J_2 = L \bullet N_2$, $K_1 = V \bullet N_1$ and $K_2 = V \bullet N_2$. These expressions are just linear interpolation for the scalar valued functions $J(t) = L \bullet N(t)$ and $K(t) = V \bullet N(t)$. Once again we can calculate these scalar valued functions at each pixel in exactly the same way that we calculate the intensities for Gouraud shading: first calculate $J(t)$ and $K(t)$ at the vertices, then interpolate along edges, and finally interpolate along scan lines. This computation is line for line the exact same computation as Gouraud shading, with $J(t)$ and $K(t)$ in place of the intensity $I(t)$. Moreover, the term $L \bullet V$ is a constant, so we need to compute this value only once. Thus, we have

$$I_{specular} = I_p k_s \left(\frac{2\underbrace{\big\{(1-t)(L \bullet N_1) + t(L \bullet N_2)\big\}}_{Similar\ to\ Gouraud}\ \underbrace{\big\{(1-t)(V \bullet N_1) + t(V \bullet N_2)\big\}}_{Similar\ to\ Gouraud}}{\big|(1-t)N_1 + t\,N_2\big|^2} - \underbrace{(L \bullet V)}_{Constant} \right)^n .$$

We are left to compute only the denominator of the first term on the right hand side. This denominator is given by

$$D(t) = \big|N(t)\big|^2 = N(t) \bullet N(t).$$

Notice, in particular, that unlike the calculation of Phong shading for diffuse reflection, there is no square root in this denominator. Now from naive Phong shading (Equation (4.1)), we know that

$$\Delta N = \Delta t(N_2 - N_1).$$

Therefore by Equation (4.2)

$$N_{next} = N_{current} + \Delta N,$$

and

$$D_{next} = N_{next} \bullet N_{next}.$$

We can calculate ΔN and N_{next} quickly using the scan line approach to Phong shading presented in Section 4.1, and there is no square root here to slow us down as in the calculation of diffuse reflection: to compute D_{next} from N_{next} only one additional dot product is required.

4.4 Phong Shading and Spherical Linear Interpolation. In Phong shading we perform linear interpolation on unit normals. But linear interpolation does not preserve length. Therefore for every pixel, we must renormalize the length of these normal vectors. This renormalization takes time and involves a square root, substantially slowing down the shading computation.

Linear interpolation is unnatural for unit vectors because linear interpolation does not preserve length. But there is a natural way to preserve length while transitioning between two unit vectors: we can simply rotate one vector into the other in the plane of the two vectors. Rotation preserves length, so no renormalization is required. If we rotate at a uniform speed, this approach is exactly *spherical linear interpolation* or *slerp*. (For a derivation of the formula for spherical linear interpolation, see Lecture 11, Section 6.)

Spherical Linear Interpolation

$$slerp(N_1, N_2, t) = \frac{\sin((1-t)\phi)}{\sin(\phi)} N_1 + \frac{\sin(t\phi)}{\sin(\phi)} N_2 \qquad (4.7)$$

$$where \quad \phi = \cos^{-1}(N_1 \bullet N_2)$$

Spherical linear interpolation eliminates the need to compute square roots (renormalization), but at the cost of computing the trigonometric functions $\sin((1-t)\phi)$ and $\sin(t\phi)$. This tradeoff hardly seems worthwhile. Fortunately, we can compute these trigonometric functions incrementally, and thus avoid computing sines for every pixel.

To see how to compute the unit normal incrementally, suppose we have already computed

$$N(t) = \frac{\sin((1-t)\phi)}{\sin(\phi)} N_1 + \frac{\sin(t\phi)}{\sin(\phi)} N_2 \qquad (4.8)$$

and we want to compute

$$N(t+\Delta t) = \frac{\sin\big((1-t-\Delta t)\phi\big)}{\sin(\phi)} N_1 + \frac{\sin\big((t+\Delta t)\phi\big)}{\sin(\phi)} N_2 .$$

First notice that the denominator, $\sin(\phi)$, is a constant. Moreover, since N_1, N_2 are unit vectors,

$$\cos(\phi) = N_1 \bullet N_2,$$

so

$$\sin(\phi) = \sqrt{1-\cos^2(\phi)} = \sqrt{1-(N_1 \bullet N_2)^2} .$$

The square root here is not a problem, since this square root is computed only once. We do not need to repeat this computation for each pixel; rather we compute $\sin(\phi)$ once and store the result.

Next observe that by the trigonometric identities for the sine of the sum or difference of two angles:

$$\sin\big((t+\Delta t)\phi\big) = \sin(t\phi)\cos(\Delta t\,\phi) + \cos(t\,\phi)\sin(\Delta t\,\phi) \qquad (4.9)$$

$$\sin\big((1-t-\Delta t)\phi\big) = \sin((1-t)\phi)\cos(\Delta t\,\phi) - \cos((1-t)\,\phi)\sin(\Delta t\,\phi) . \qquad (4.10)$$

Since $\Delta t, \phi$ are constants, the values $\cos(\Delta t\,\phi)$, $\sin(\Delta t\,\phi)$ are also constants, so we can also compute these values once and store them for subsequent use. Furthermore, we can assume that we have already calculated $\sin(t\,\phi)$, $\sin((1-t)\phi)$ for the current pixel, so we do not need to recalculate these values for the next pixel. But we still need $\cos(t\,\phi)$, $\cos((1-t)\,\phi)$. Fortunately, these values can also be computed incrementally from the trigonometric identities for the cosine of the sum or difference of two angles:

$$\cos\big((t+\Delta t)\phi\big) = \cos(t\,\phi)\cos(\Delta t\,\phi) - \sin(t\,\phi)\sin(\Delta t\,\phi) \qquad (4.11)$$

$$\cos\big((1-t-\Delta t)\phi\big) = \cos((1-t)\phi)\cos(\Delta t\,\phi) + \sin((1-t)\,\phi)\sin(\Delta t\,\phi). \qquad (4.12)$$

Using equations (4.9)-(4.12), after computing the constants $\cos(\Delta t\,\phi)$, $\sin(\Delta t\,\phi)$, we can compute all the necessary sines and cosines incrementally with just four additions/subtractions and eight multiplications. No trigonometric evaluations are necessary, nor are any square roots required.

Now we can proceed in the usual fashion for Phong shading:

i. First, calculate the unit normals at the individual vertices of each polygon.
ii. Next, apply spherical linear interpolation, using equations (4.9)-(4.12) with $\Delta t = 1/(y_2 - y_1)$ to calculate the unit normals incrementally along the edges of each polygon.
iii. Next, apply spherical linear interpolation to the unit normals along the edges, using equations (4.9)-(4.12) with $\Delta t = 1/(x_2 - x_1)$ to calculate the unit normals incrementally along scan lines for the pixels in the interior of each polygon.
iv. Finally, use these unit normals to calculate the intensity at each pixel using Equation (2.1).

The preceding algorithm corresponds to naive Phong shading with spherical linear interpolation replacing standard linear interpolation. This procedure eliminates square roots, but if we were to follow this approach, in step *iv* we would need to compute two dot products -- $L \bullet N$ and $V \bullet N$ -- for every pixel. To eliminate these dot products, we can proceed as in fast Phong shading by setting $J(t) = L \bullet N(t)$ and $K(t) = V \bullet N(t)$. Substituting the right hand side of Equation (4.8) for $N(t)$ into these expression for $J(t)$ and $K(t)$ yields the scalar valued functions:

$$J(t) = \frac{\sin\big((1-t)\phi\big)}{\sin(\phi)} L \bullet N_1 + \frac{\sin\big(t\,\phi\big)}{\sin(\phi)} L \bullet N_2$$

$$K(t) = \frac{\sin\big((1-t)\phi\big)}{\sin(\phi)} V \bullet N_1 + \frac{\sin\big(t\,\phi\big)}{\sin(\phi)} V \bullet N_2.$$

Now we can proceed as in the previous algorithm, but instead of updating the unit normal vector $N(t)$ incrementally along scan lines using Equations (4.9)-(4.12), we use these equations to incrementally update the scalar valued functions $J(t)$ and $K(t)$ along scan lines. This approach eliminates the need to compute two dot products at every pixel, but notice that we still need to update the normal vector $N(t)$ incrementally along each edge, since we need the angle ϕ between normal vectors at opposite edges along each scan line in order to calculate the functions $J(t)$ and $K(t)$ along the scan lines.

5. Summary

The purpose of shading algorithms is to smooth the appearance of polygonal models by reducing the impact of sharp edges in order to give the eye the impression of a smooth curved surface. Gouraud shading blurs these sharp edges by applying linear interpolation to the intensity for each polygon along scan lines. But Gouraud shading also blurs specular reflections. Phong shading provides a better model for curved surfaces by performing linear interpolation of normal vectors along scan lines. Phong shading is slower than Gouraud shading, but Phong shading is considerably better for specular reflections.

We investigated several ways to speed up Phong shading, including calculating diffuse and specular reflections independently by linear interpolation and using Newton's method to compute the square root needed to find the magnitude of the normal vector for diffuse reflection. We also examined spherical linear interpolation of unit normals in order to avoid the square root altogether.

To speed up these algorithms, the interpolation steps in both Gouraud and Phong shading are performed incrementally. Below we summarize these incremental formulas for both Gouraud and Phong shading.

Gouraud Shading

$I_{next} = I_{current} + \Delta I$ {intensity}

-- $\Delta I = (I_2 - I_1) / (x_2 - x_1)$ {along scan lines}

-- $\Delta I = (I_2 - I_1) / (y_2 - y_1)$ {along edges}

Phong Shading

a. Naive Phong Shading

$$N_{next} = \frac{N_{current} + \Delta N}{\left| N_{current} + \Delta N \right|}$$ (unit normal}

-- $\Delta N = (N_2 - N_1) / (x_2 - x_1)$ {along scan lines}

-- $\Delta N = (N_2 - N_1) / (y_2 - y_1)$ {along edges}

b. Fast Phong Shading

i. Diffuse Reflection

$J(t) = L \bullet N(t)$ *{numerator}*

$J_{next} = J_{current} + \Delta J$

-- $\Delta J = (J_2 - J_1) / (x_2 - x_1)$ {along scan lines}

-- $\Delta J = (J_2 - J_1) / (y_2 - y_1)$ {along edges}

$d_{next} = \sqrt{N_{next} \bullet N_{next}}$ {denominator}

$N_{next} = N_{current} + \Delta N$

-- $\Delta N = (N_2 - N_1) / (x_2 - x_1)$ {along scan lines}

-- $\Delta N = (N_2 - N_1) / (y_2 - y_1)$ {along edges}

ii. Specular Reflection

$J(t) = L \bullet N(t)$

$K(t) = V \bullet N(t)$ *{numerator}*

$J_{next} = J_{current} + \Delta J$

-- $\Delta J = (J_2 - J_1) / (x_2 - x_1)$ {along scan lines}

-- $\Delta J = (J_2 - J_1) / (y_2 - y_1)$ {along edges}

$K_{next} = K_{current} + \Delta K$

-- $\Delta K = (K_2 - K_1) / (x_2 - x_1)$ {along scan lines}

-- $\Delta K = (K_2 - K_1) / (y_2 - y_1)$ {along edges}

$D_{next} = N_{next} \bullet N_{next}$ {denominator}

$N_{next} = N_{current} + \Delta N$

-- $\Delta N = (N_2 - N_1) / (x_2 - x_1)$ {along scan lines}

-- $\Delta N = (N_2 - N_1) / (y_2 - y_1)$ {along edges}

c. *Spherical Linear Interpolation*

$$N_{next} = N(t+\Delta t) = \frac{\sin\big((1-t-\Delta t)\phi\big)}{\sin(\phi)} N_1 + \frac{\sin\big((t+\Delta t)\phi\big)}{\sin(\phi)} N_2 \qquad \{\text{along edges only}\}$$

$$J_{next} = J(t+\Delta t) = \frac{\sin\big((1-t-\Delta t)\phi\big)}{\sin(\phi)} L \bullet N_1 + \frac{\sin\big((t+\Delta t)\phi\big)}{\sin(\phi)} L \bullet N_2$$

$$K_{next} = K(t+\Delta t) = \frac{\sin\big((1-t-\Delta t)\phi\big)}{\sin(\phi)} V \bullet N_1 + \frac{\sin\big((t+\Delta t)\phi\big)}{\sin(\phi)} V \bullet N_2$$

-- $\sin\big((t+\Delta t)\phi\big) = \sin(t\,\phi)\cos(\Delta t\,\phi) + \cos(t\,\phi)\sin(\Delta t\,\phi)$

-- $\sin\big((1-t-\Delta t)\phi\big) = \sin((1-t)\phi)\cos(\Delta t\,\phi) - \cos((1-t)\,\phi)\sin(\Delta t\,\phi)$

-- $\cos\big((t+\Delta t)\phi\big) = \cos(t\,\phi)\cos(\Delta t\,\phi) - \sin(t\,\phi)\sin(\Delta t\,\phi)$

-- $\cos\big((1-t-\Delta t)\phi\big) = \cos((1-t)\phi)\cos(\Delta t\,\phi) + \sin((1-t)\,\phi)\sin(\Delta t\,\phi)$

-- $\Delta t = 1/(x_2 - x_1)$ {along scan lines}

-- $\Delta t = 1/(y_2 - y_1)$ {along edges}

Exercises:

1. A table of sines states that $\sin(24^o) = 0.40674$ and $\sin(25^o) = 0.42262$. Use linear interpolation to estimate $\sin(24.3^o)$.

2. Show how to compute e^t incrementally using only one multiplication per pixel.

Programming Project:

1. Implement both Gouraud and Phong shading in your favorite programming language using your favorite API.
   a. Use your implementation to shade several large polygonal models.
   b. Compare the relative speeds of the following algorithms for each of your models:
      i. Gouraud Shading
      ii. Naive Phong Shading
      iii. Fast Phong Shading
      iv. Phong Shading using Spherical Linear Interpolation