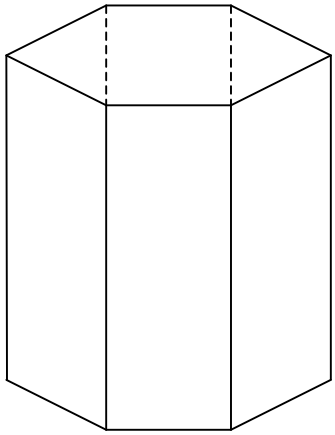
Shading Techniques

1. Summary of Shading Techniques

Throughout these notes we'll assume that we want to display a polygonal object. There are two basic approaches, which might be mixed not only within a scene, but also within an object, flat shading and smooth shading. E.g., say that we want to display a can of pop, which is represented as a series of polygons; in the figure below we have eight polygons, the two on the top and bottom and the six round the outside. If we display this object using flat shading we'll see the object as it appears here with dramatic jaggies and not much resemblance to a cylinder. If, however, we display it with smooth shading, then the outside will appear cylindrical, with no obvious vertical edges, and the top and bottom will appear flat.



There are three major shading techniques, one for flat shading and two for smooth shading. The flat shading method, which is called Lambert shading, is by far the fastest, and so it is used not only for polygonal objects which should have flat surfaces, but also for smooth objects when we want to render them faster e.g., in some animations, when the object is far away or out of the main view, or when we are debugging. The two smooth methods are much slower than Lambert shading. Phong shading produces better output than Gouraud shading, but it takes much longer to render the images.

Bui-Tuong Phong has been very productive, and so his name is attached to both the Phong illumination model and Phong shading, and so you have to be careful to avoid confusion. These are completely different systems and it is very possible, for example, that you will be using the Phong illumination model with any of the three shading methods, Lambert, Gouraud, or Phong.

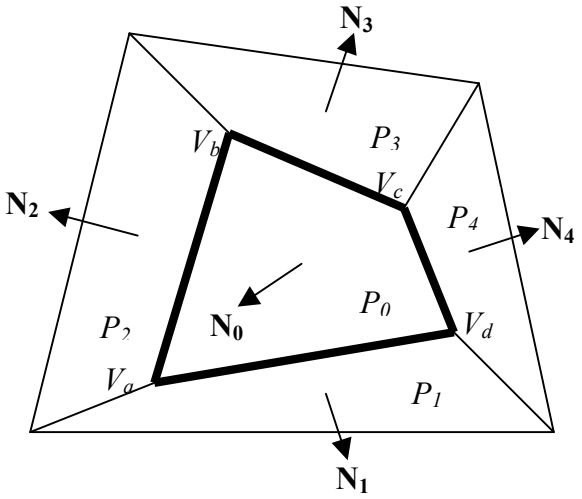
2. Lambert (Flat) Shading

With Lambert shading each polygon is uniformly colored. Computationally it is very simple and very fast. For each polygon in the scene we select one point, use the Phong illumination model to compute its color, and then flood fill that color through the polygon. In theory, it is best to use the center of the polygon for the point that is selected for coloring, but in practice many systems will just select one of the vertices since this isn't a very high quality rendering system and so computing the center of gravity of the vertices might not be worth the tiny amount of extra effort.

3. Smooth Shading and Vertex Normals

As we have discussed, there are two major smooth shading techniques, Gouraud and Phong. While they differ in their final details, they are both based on the concept of vertex normals, which we'll describe in this section, and on linear interpolation across the polygon. The difference, as we will see, is that Gouraud only runs the illumination model to calculate intensities (i.e., colors) at the vertices, and then linearly interpolates these intensities across the polygon. By comparison, Phong interpolates the normals across the polygon and then does an illumination calculation at every point in the polygon that will project to a pixel on the screen.

The first thing that we need to define is a vertex normal. (For the mathematically inclined this can be annoying, since the vertices are the only place on a polygonal structure where the normals aren't well defined.) Say that we have the polygon P_0, in bold, with four vertices V_a, V_b, V_c, V_d, shown below, which has the four adjacent polygons P_1, P_2, P_3, and P_4. Also assume that the unit polygon normal for each polygon P_i is $\mathbf{N_i}$, as shown.



Think of this figure with P_0 higher than the other polygons, which slope down the hill.

If we use Lambert shading, then the polygonal structure will be displayed. Say, however, we want this to look like the smooth top of the hill, which we've represented with polygons. This is where we will use Gouraud or Phong shading. In both cases we first define the unit vertex normals as being the average of the surrounding polygon normals that we want to appear to be smoothly connected. In this case we will compute

$$\mathbf{N_a} = (\mathbf{N_0} + \mathbf{N_1} + \mathbf{N_2}) / | \mathbf{N_0} + \mathbf{N_1} + \mathbf{N_2} |,$$
$$\mathbf{N_b} = (\mathbf{N_0} + \mathbf{N_2} + \mathbf{N_3}) / | \mathbf{N_0} + \mathbf{N_2} + \mathbf{N_3} |,$$
$$\mathbf{N_c} = (\mathbf{N_0} + \mathbf{N_3} + \mathbf{N_4}) / | \mathbf{N_0} + \mathbf{N_3} + \mathbf{N_4} |, \text{ and}$$
$$\mathbf{N_d} = (\mathbf{N_0} + \mathbf{N_1} + \mathbf{N_4}) / | \mathbf{N_0} + \mathbf{N_1} + \mathbf{N_4} |,$$

where $\mathbf{N_a}$ is the defined vertex normal at V_a, etc.

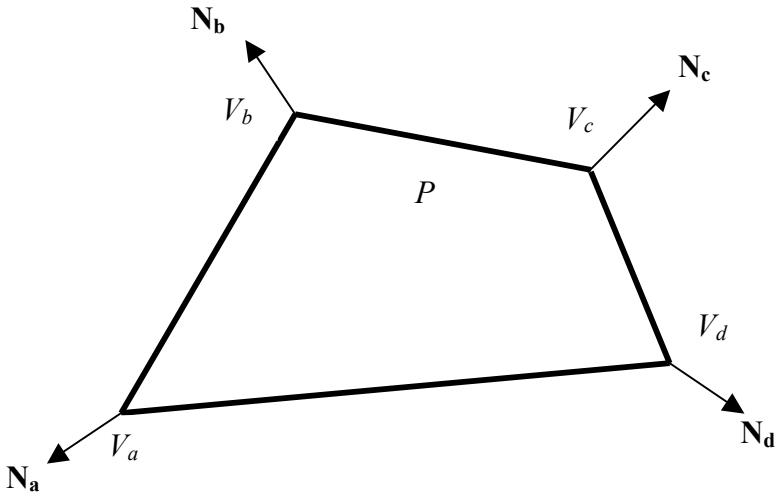
Alternatively, say that we'd wanted a smooth saddle across polygons P_4, P_0, and P_2, with polygons P_1 and P_3 dropping off with an abrupt edge. Now we'll change the vertex normals for P_0 so that they no longer average in P_1 and P_3. I.e., we'll use the equations:

$$\mathbf{N_a} = (\mathbf{N_0} + \mathbf{N_2}) / | \mathbf{N_0} + \mathbf{N_2} |,$$
$$\mathbf{N_b} = (\mathbf{N_0} + \mathbf{N_2}) / | \mathbf{N_0} + \mathbf{N_2} |,$$
$$\mathbf{N_c} = (\mathbf{N_0} + \mathbf{N_4}) / | \mathbf{N_0} + \mathbf{N_4} |, \text{ and}$$
$$\mathbf{N_d} = (\mathbf{N_0} + \mathbf{N_4}) / | \mathbf{N_0} + \mathbf{N_4} |.$$

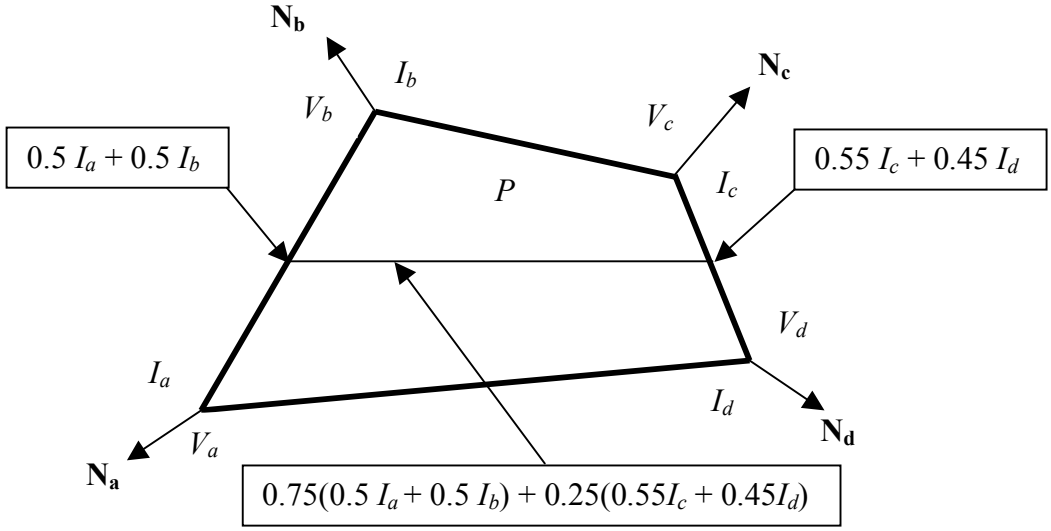
The difference between Gouraud and Phong shading, as we'll see in more detail in the sections below, is that Gouraud uses these normals to compute color intensities at the vertices and then interpolates the intensities across the polygon, while Phong interpolates the normals across the polygons and then uses the interpolated normals to compute colors at each pixel.

4. Gouraud Shading

Consider the polygon, *P*, shown below, where we have computed vertex normals as shown by averaging against the smoothly surrounding polygons.



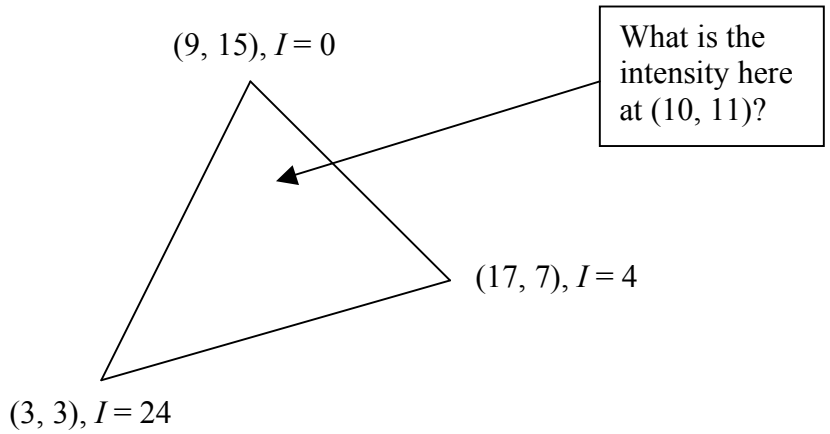
We will now use these vertices to compute intensities (colors) at the vertices, usually by using the Phong illumination model with the vertex normal as the normal at the point. This gives intensities I_a, I_b, I_c, and I_d at the vertices as shown below. Using the Gouraud method we now linearly interpolate these intensities down all of the edges of the polygons and then across rows giving, for example, the values shown in the figure in boxes.



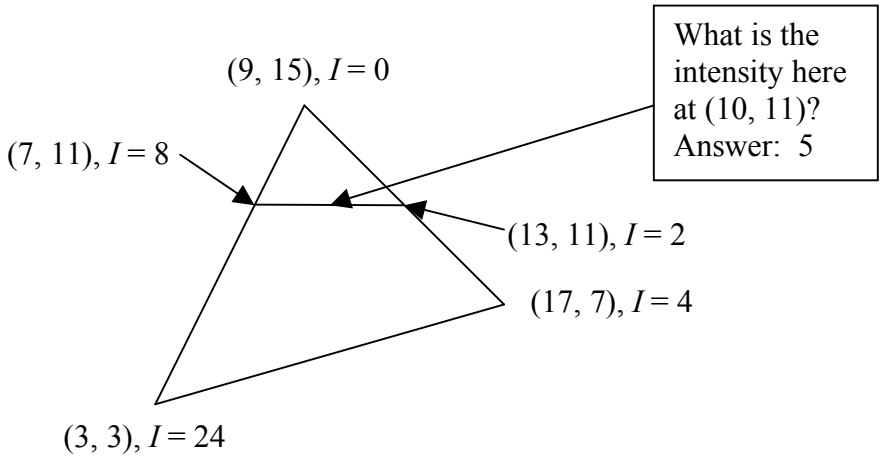
The approach, which we'll look at in more detail in a later section on efficiently computing linear interpolations, is that if, for example, we are $\frac{2}{3}$ down an edge, then the intensity will be $\frac{2}{3}$ of the intensity of the closest edge vertex and $\frac{1}{3}$ of the intensity of the farthest edge vertex. Then to get the intensities of interior points we linearly interpolate across scan lines using the edge intensities as the values on each end.

Gouraud Example:

Consider the triangle below, where the vertex coordinates and intensities are shown:

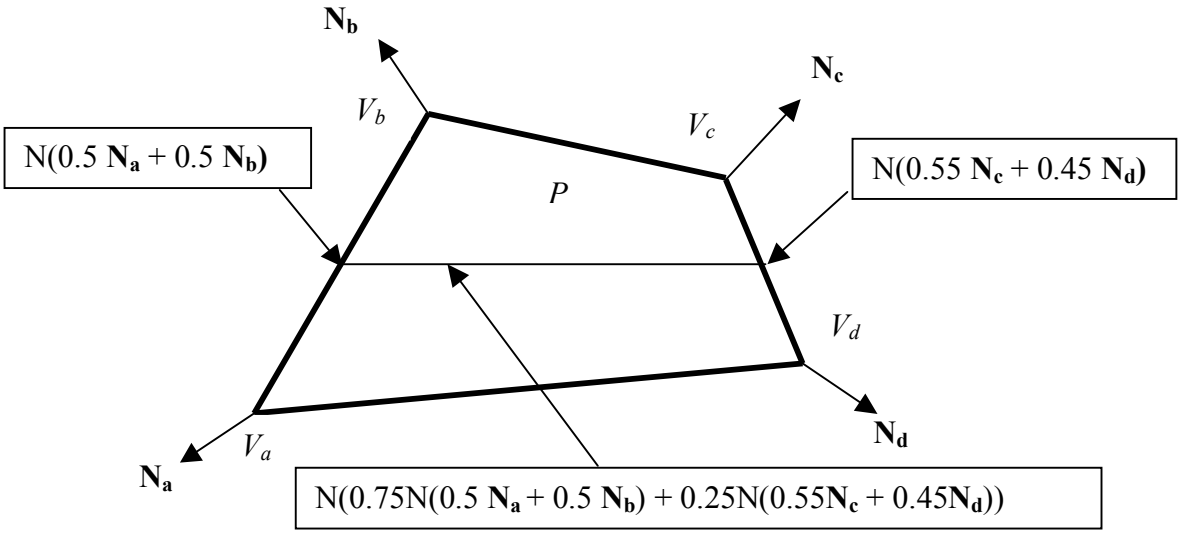


Simple calculations say that the endpoints of the scanline through (10, 11) are (7, 11) and (13, 11). (7, 11) is $\frac{1}{3}$ down the left edge, so has intensity 8 ($\frac{2}{3}0+\frac{1}{3}24$) and (13, 11) is $\frac{1}{2}$ down the right edge, so has intensity 2 ($\frac{1}{2}0+\frac{1}{2}4$). The midpoint of this scanline is (10, 11), so its intensity is 5 ($\frac{1}{2}8+\frac{1}{2}2$).


What is the intensity here at (10, 11)? Answer: 5
(9, 15), $I = 0$
(7, 11), $I = 8$
(13, 11), $I = 2$
(17, 7), $I = 4$
(3, 3), $I = 24$

5. Phong Shading

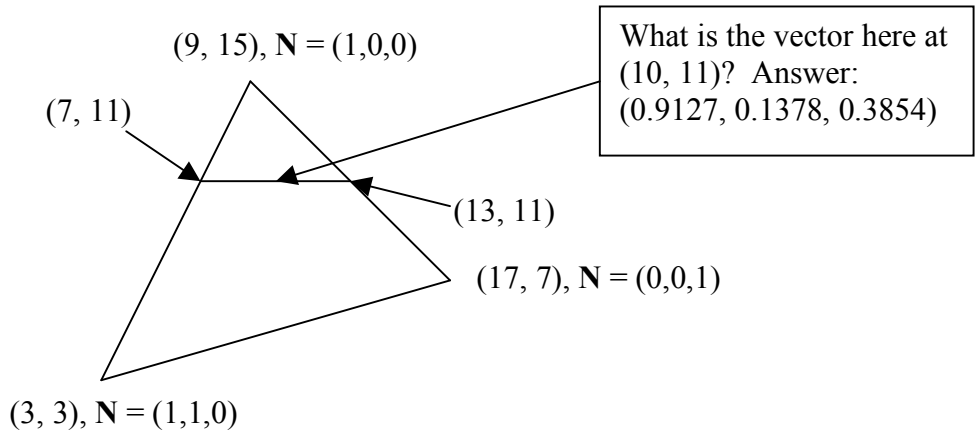
With Phong shading we interpolate the normals, instead of the intensities, down the edges of the polygon. E.g., in the figure that we used for Gouraud, the normal half way down the left edge will have the value 0.5 $\mathbf{N_a}$ + 0.5 $\mathbf{N_b}$, which now needs to be normalized by dividing by its length. To stop the figure becoming too messy, I've added a function N(**vector**) which returns the normalized vector. I.e., N(**vector**) = **vector** / | **vector** |.



So apart from the fact that we are averaging vectors instead of intensities, and that we need to ensure that all of our vectors are unit by normalizing them as soon as they are created, the interpolation procedure is similar to the procedure that we used for Gouraud.

Phong Example:

Things rapidly get messy as we interpolate vectors, so I have used very simple vertex normals on the same figure that I used for the Gouraud example in the figure below:

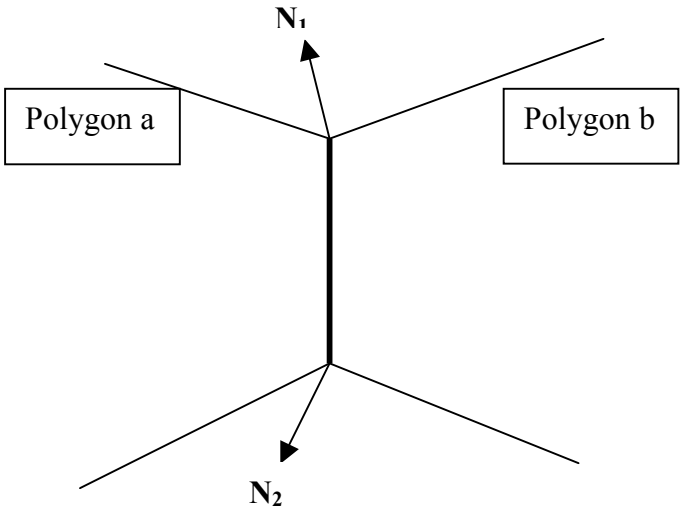


Before we do anything else, remember that the vertex normals need to be unit vectors. Two of them are, but we need to normalize the vector at (3, 3) to get (0.7071, 0.7071, 0) before we begin the interpolation.

The normal at (7, 11) is N($\frac{2}{3}$(1, 0, 0) + $\frac{1}{3}$(0.7071, 0.7071, 0)), which is N(0.9024, 0.2357, 0), or (0.9675, 0.2527, 0). The normal at (13, 11) is N($\frac{1}{2}$(1, 0, 0) + $\frac{1}{2}$(0, 0, 1)), which is N(($\frac{1}{2}$, 0, $\frac{1}{2}$)), or (0.7071, 0, 0.7071). The midpoint of the line between them has vector N(0.8373, 0.1264, 0.3536), which gives our solution vector (0.9127, 0.1378, 0.3854).

6. Why do Gouraud and Phong Make the Object Appear Smooth?

To see why the two smooth shading techniques work well, look at the shared edge between two polygons, after vertex normals have been computed.

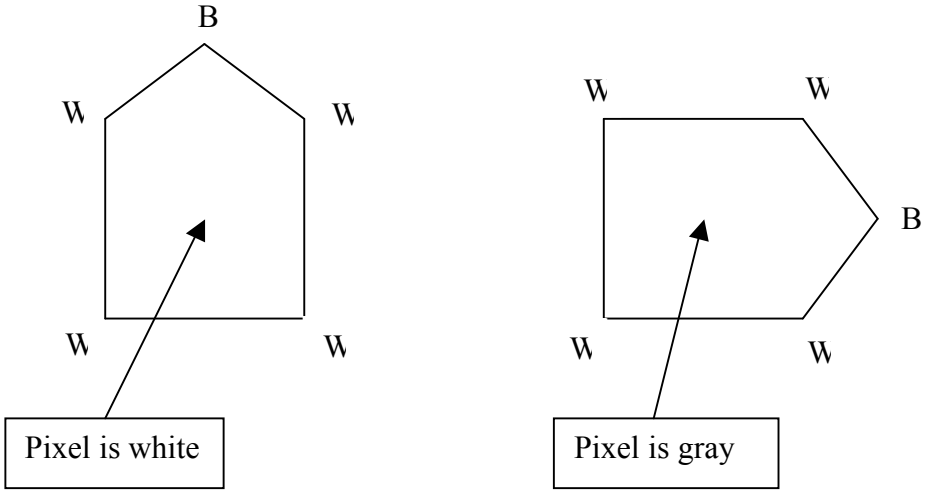


The vertex normals, $\mathbf{N_1}$ and $\mathbf{N_2}$, will be the same for both polygons, and so the intensities (directly for Gouraud or indirectly computed from normals for Phong) will be the same at any point on the edge for both polygons. Given any scan line that passes through the edge, the two polygons linearly interpolate the intensities on either side of the point where the scanline intersects the edge, and so values on either side of the edge are almost identical. Since there are no color discontinuities at the edge, it doesn't show up. In fact the only change at the edge is that the rate of change of the colors can change on opposite sides of the edge.

7. Comparative Strengths and Weaknesses of Gouraud and Phong Shading

Both methods share a problem, which is that although they do a pretty good job of smoothly shading the interior of the object, the silhouette of the object will still have jaggies. E.g., consider the cylinder that we had at the beginning of these notes. Even though it might appear to be smooth in its interior, the bottom will still have the three straight edges shown in the polygonal diagram. One solution to this is to use much smaller polygons near the outside edge of an object than we use in the interior, but increasing the number of polygons will, of course, increase the computational cost.

Another problem with both methods is that they aren't invariant under rotation. I.e., as you rotate a polygon the internal colors are not always the same. Fortunately this usually isn't obvious, and it takes a relatively pathological case to demonstrate it. E.g., consider the two cases below, where the second figure is a 90° rotation of the first. Assume that B is black and W is white.

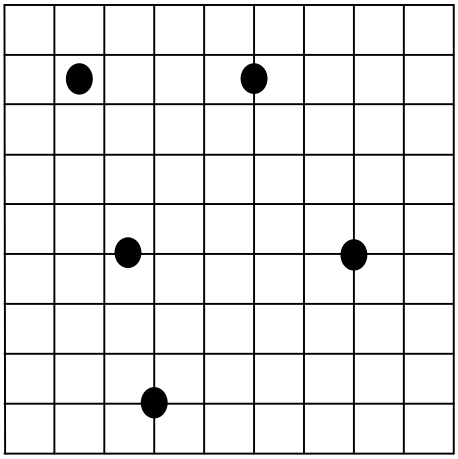


To see why this has happened, remember that we scan down the edges and then across the scan lines. In the left figure the pixels on both ends of the scan line are white, and so everything on the scanline is white. In the right figure the scan line is interpolating from white to black, and so all pixels on the scanline are grays, darkening from white to black across the scanline.

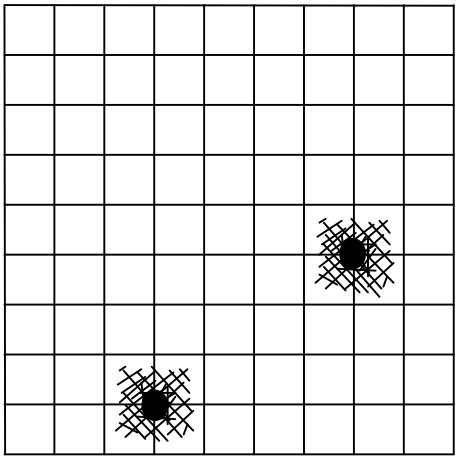
Gouraud's only advantage over Phong and it is a very significant advantage, is that it is much faster since (a) it only has to compute the illumination

model once for each vertex, and Phong has to do it for every displayed pixel, and (b) interpolating and normalizing the vectors is much more work than interpolating the intensities.

One disadvantage of Gouraud compared to Phong is that it doesn't deal well with hot spots, in particular if the size of the hot spots are small relative to the size of the polygon mesh which is defining the object. E.g., say that we have the polygonal mesh shown below which has 81 polygons, bending around in 3D, where the black circles are places where there are supposed to be relatively small hot spots.



With Phong, the picture will look just as it is supposed to. However with Gouraud the picture will look more like:



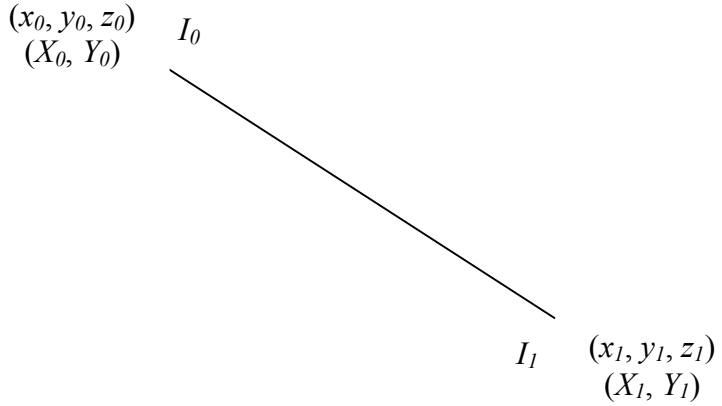
I.e., some hot spots will vanish completely, and others will lose their definition and take on some of the shape of the surrounding polygons. This

is because Gouraud is based on interpolating from the intensities at the vertices. If the hotspot doesn't cover at least one vertex, it can't appear in the averages over the vertex colors. Also, when it is on a vertex the intensity will usually be very bright as compared to the other vertices, so its effects, under interpolation, will be felt through most of the containing polygons. To avoid these problems you need polygons that are small relative to hot spot size, but that can be computationally expensive as it requires more polygons.

Gouraud can also have a problem with mach banding, which doesn't show up with Phong. This is an appearance of light parallel stripes in the image, which can partially be blamed on the human visual system, which does a very good job of enhancing very subtle changes in intensity across an edge.

8. Computing Interpolations Efficiently

There are two kinds of interpolations that go on, in either Gouraud or Phong; first we interpolate along all of the polygon edges, and then we interpolate across scan lines. Since the scanlines are on the screen this means that we are only interested in edge values that are also pixel locations on the screen. So we will have some polygon with 3D vertices (x_0, y_0, z_0), (x_1, y_1, z_1), …, which we will have to first project these vertices into screen pixel locations (X_0, Y_0), (X_1, Y_1), …, and it is these values which will be used in the interpolation. Throughout this section I will assume that we are using Gouraud and interpolating intensities. This is just to simplify the notation. Phong will follow exactly the same interpolation procedures. The figure below shows a typical edge for the polygon:



We have an edge going from the 3D point (x_0, y_0, z_0) to (x_1, y_1, z_1), which when projected onto the screen have the coordinates (X_0, Y_0) and (X_1, Y_1). Since we are wanting endpoints for scanlines, we need to interpolate at each integer Y value between Y_0 and Y_1, finding the appropriate X values and intensities. We can do this with a simple loop like:

sort (X_0, Y_0) and (X_1, Y_1) so that $Y_0 > Y_1$;
$X = X_0$; $Y = Y_0$; $I = I_0$;
$N = Y_0 - Y_1$;
$\delta X = (X_1 - X_0) / N$; $\delta I = (I_1 - I_0) / N$;
StorePixel(X, Y, I);
while ($Y > Y1$) {
    Y--;
    X += δX;
    I += δI;
    StorePixel(round(X), Y, I);

}

The basic idea is that once we've computed the δX and δI by dividing the changes in X's and I's by the difference in the Y's, we can just add in these X and intensity changes for each scanline (new Y). Unfortunately we still need to compute round(X) to get the nearest integer X. (This can be improved on by using Bresenham line drawing techniques to get the new X values without any float operations.)

The function StorePixel will store scanline endpoints with corresponding intensities for subsequent interpolation across the line. This interpolation is much easier since the Y values are constant. Now N becomes the difference in the X's, δI is computed as before, and a simple while loop on the X's adds in δI each time to the latest intensity value.