Shading Models

Prerequisites

An understanding of the concept of color, of polygons, and of intepolation across a polygon.

Introduction

Shading is the process of computing the effect of light on an object in a scene. This process is based on the physics of light, and the most detailed kinds of shading computation can involve deep subtleties in that physics, including the way light scatters from various kinds of materials with various details of surface treatments. Considerable Rresearch has been done in those areas and any genuinely realistic rendering must take surface details into account.

Most graphics APIs do not have the capability to do these detailed kinds of computation. The usual beginning API such as OpenGL supports two shading models for polygons: flat shading and smooth shading. You may choose either, but smooth shading is usually more pleasing and can be somewhat more realistic. Unless there is a sound reason to use flat shading in order to represent data or other communication concepts more accurately, you will probably want to use smooth shading for your images. We will briefly discuss just a bit more sophisticated kinds of shading, even though the beginning API cannot directly support them.

Definitions

Flat shading of a polygon assumes that each polygon is strictly planar and all the points on the polygon have exactly the same kind of lighting treatment. This shading technique thus displays all the points in a polygon with a single color. This is the effect you will get if you set a color for the polygon, or define a polygon's color with glMaterial calls, and then display the polygon with a single normal vector for each polygon inside the `glBegin(...) - glEnd()` pair that models the polygon. This allows you only a single lighting computation for the entire polygon, so the polygon is presented with only one color.

Smooth shading of a polygon displays the points in a polygon with smoothly-changing colors across the surface of the polygon. This requires you to define a separate color for each vertex of your polygon, because the smooth color change is computed by interpolating the vertex colors across the interior of the triangle with the standard kind of interpolation we saw in the graphics pipeline discussion. Computing the color for each vertex is done with the usual computation of a standard lighting model, but in order to compute the color for each vertex separately you must define a separate normal vector for each vertex of the polygon. This allows the color of the vertex to be determined by the lighting model that includes this unique normal.

The default shading behavior of OpenGL is smooth, but you will not get the visual effect of smooth shading unless you specify the appropriate normals for your model, as described below. OpenGL allows you to select the shading model with the `glShadeModel(...)` function and the only values of the single parameter are the symbolic parameters `GL_SMOOTH` and `GL_FLAT`. You may use the glShadeModel function to switch back and forth between smooth and flat shading any time you wish.

Some examples

We have seen many examples of polygons, and we have not been careful to distinguish between flat and smooth shading in them. Figure 8.1 shows see two different images of the same function surface, one with flat shading (left) and one with smooth shading (right), to illustrate the difference. Clearly the smooth-shaded image is much cleaner, but there are still some areas where the triangles change direction very quickly and the boundaries between the triangles still show in the smoothly-shaded image. Smooth shading is very nice, but it isn't perfect.

The difference between the programming for these two shading models is that the flat-shaded model uses only one normal per polygon and the smooth-shaded model uses a separate normal per vertex. It can take a bit more work to compute the normal at each vertex instead of only once per polygon, but that is the price for smoothing. In the example shown above, the vertex normals for smooth shading are calculated by using the analytic partial derivatives at each vertex, while the normals for flat shading are simply calculated based on a cross product of two triangle edges. This is shown in the code sample below and in the full code example, `flatSmooth.c`.

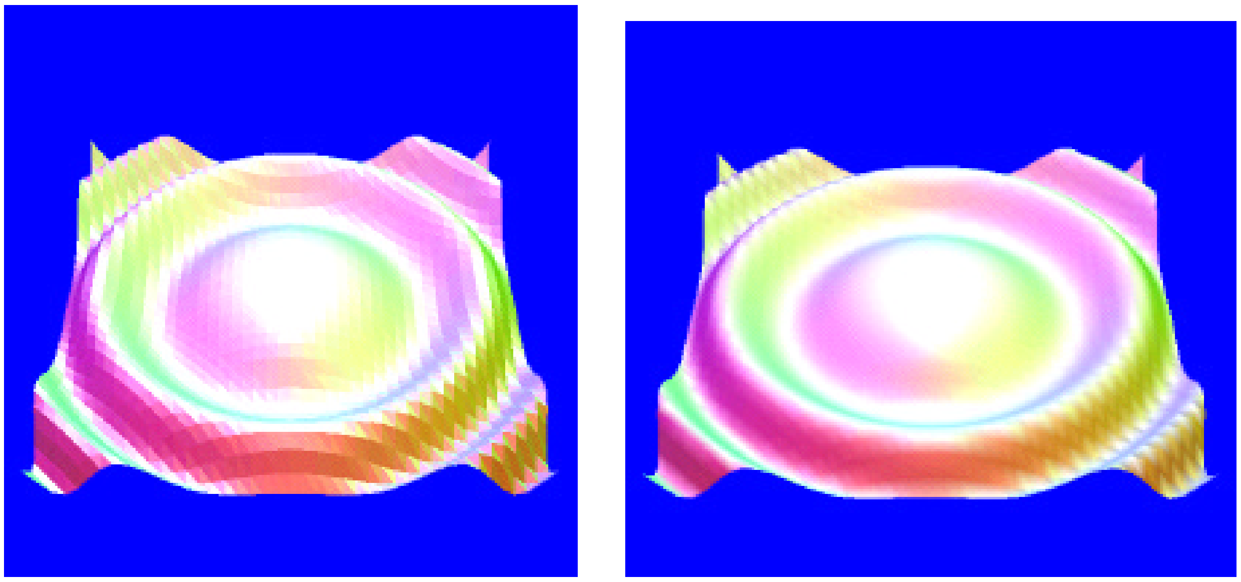

Figure 8.1: a surface with flat shading (left) and the same figure with smooth shading (right)

The computation for smooth shading uses the polygon interpolation described in the chapter on the graphics pipeline. Because each vertex has its own normal, the lighting model computes a different color for each vertex. The interpolation then calculates colors for each pixel in the polygon that vary smoothly across the polygon interior, providing a smooth color graduation across the polygon. This interpolation is called *Gouraud shading* and is one of the standard techniques for creating images. It is quick to compute but because it only depends on colors at the polygon vertices, it can miss lighting effects within polygons. Visually, it is susceptible to showing the color of a vertex more strongly along an edge of a polygon than a genuinely smooth shading would suggest, as you can see in the right-hand image in Figure 8.1. Other kinds of interpolation are possible that do not show some of these problems, though they are not often provided by a graphics API, and these are discussed below.

An interesting experiment to help you understand the properties of shaded surfaces is to consider the relationship between smooth shading and the resolution of the display grid. In principle, you should be able to use fairly fine grid with flat shading or a much coarser grid with smooth shading to achieve similar results. You should define a particular grid size and flat shading, and try to find the smaller grid that would give a similar image with smooth shading. Figure 8.2 is an example of this experiment; it still shows a small amount of the faceting of flat shading but avoids much of the problem with quickly-varying surface directions and is probably superior in many ways to the smooth-shaded polygon of Figure 8.1. It may be either faster or slower than the original smooth shading, depending on the efficiency of the polygon interpolation in the graphics pipeline. If you have two different ways to approximate an effect, it can be very useful to try both and see which works better — both for effect and for speed — in a particular application!

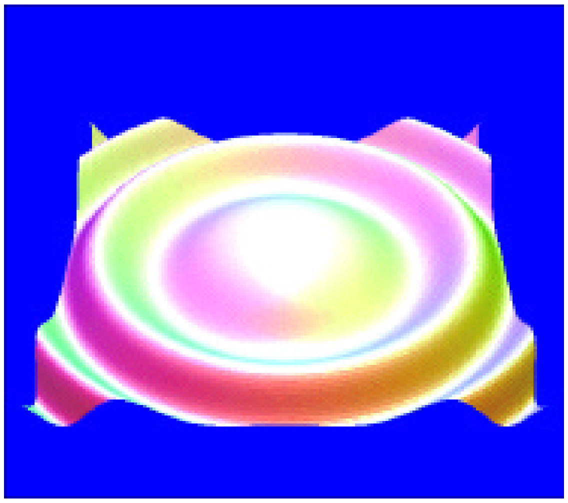

Figure 8.2: a flat-shaded image with resolution three times as great as the previous figure

Other shading models

You cannot and must not assume that the OpenGL smooth shading model is an accurate representation of smooth surfaces. It assumes that the surface of the polygon varies uniformly, it only includes information at the vertices in calculating colors across the polygon, and it relies on a linear behavior of the RGB color space that is not accurate, as we described when we talked about colors. Like many of the features of any computer graphics system, including OpenGL, it approximates a reality, but there are better ways to achieve the effect of smooth surfaces. For example, there is a shading model called *Phong shading* that requires the computation of one normal per vertex and uses the interpolated values of the normals themselves to compute the color at each pixel in the polygon, instead of simply interpolating the vertex colors. This model behaves like a genuinely smooth surface across the polygon, including picking up specular highlights within the polygon and behaving smoothly along the edges of the polygon. The details of how Gouraud and Phong shading operate are discussed in any graphics textbook. We encourage you to read them as an excellent example of the use of interpolation as a basis for many computer graphics processes.

The Phong shading model assumes that normals change smoothly across the polygon, but another shading model is based on controlling the normals across the polygon. Like the texture map that we describe in a later chapter, we may create a mapping that alters the normals in the polygon so the shading model can create the effect of a bumpy surface. This is called a *bump map*, and like Phong shading the normal for each individual pixel is computed as the normal from Phong shading plus the normal from the bump map. The color of each individual pixel is then computed from the lighting model.

Code examples

The two issues in using OpenGL shading are the selection of the shading model and the specification of a color at each vertex, either explicitly with the `glColor*(...)` function or by setting a normal per vertex with the `glNormal*(...)` function. Sample code to set up smooth shading by the latter approach, from the code in `flatSmooth.c` that generated the figures in this example, is shown below. To begin, note that we will have to generate one normal per vertex with smooth shading, so we define the two partial derivatives for the function in order to get tangent vectors at each vertex:

```
      #define f(x,y) 0.3*cos(x*x+y*y+t)         // original function
      #define fx(x,y) -0.6*x*sin(x*x+y*y+t)     // partial derivative in x
      #define fy(x,y) -0.6*y*sin(x*x+y*y+t)     // partial derivative in y
```

We then use the following function call in the `init()` fuction to ensure that we automatically normalize all our normals in order to avoid having to do this computation ourselves:

```
      glEnable(GL_NORMALIZE); //make normals one unit long after transform
```

In the display function, we first compute the values of `x` and `y` with the functions `XX(i)` and `YY(j)` that compute the grid points in our domain, and then we do the following (fairly long) computation for each triangle in the surface, using an inline cross product operation. We are careful to compute the triangle surface normal as (X–partial cross Y–partial), in that order, so we get the correct direction for it.

```
      glBegin(GL_POLYGON);
            x = XX(i);
            y = YY(j);
            vec1[0] = 1.0;
            vec1[1] = 0.0;
            vec1[2] = fx(x,y); // partial in X-Z plane
            vec2[0] = 0.0;
            vec2[1] = 1.0;
            vec2[2] = fy(x,y); // partial in Y-Z plane
            triNormal[0] = vec1[1] * vec2[2] - vec1[2] * vec2[1];
            triNormal[1] = vec1[2] * vec2[0] - vec1[0] * vec2[2];
            triNormal[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0];
            glNormal3fv(triNormal);
            glVertex3f(XX(i  ),YY(j  ),vertices[i  ][j  ]);
            ... // do the code above two more times for each vertex of the
            ... // two triangles in a quad
      glEnd();
```

Of course, there are many other ways to deal with normals, but you cannot simply compute a normal as a cross product of two edges because this cross product will be the same for each vertex of a triangle — any triangle is planar, after all.

Science projects

Shading is one of the aspects of graphics that can greatly enhance images used for communicating science — or can turn the science into just “pretty pictures” that cover the science with information-hiding graphical techniques. It is critically important that when you use graphics for science, you keep foremost in your mind the need to represent the science accurately and not use graphics just to get nice images.