Computer Graphics – Lab Assignment
Shading

1 Introduction

The way an object looks is largely determined by its material properties and the way light interacts with the object's surface. Examples of material properties are an object's color, if it is shiny or reflective (or perhaps event transparent).

The framework for this assignment displays a scene consisting of a deformed sphere with a house on top. When the framework program is run, the scene will look a bit dark and dull. It's also hard to tell if the green sphere is actually rounded or not, as it has the same color everywhere. The sphere actually consists of many small rectangles. Notice that the surfaces of both the house and sphere have an intrinsic color of their own that is independent on the viewing angle or the position of the eye of the observer.

OpenGL (and other 3D libraries) distinguish between four different types of light effects, and each light source can be composed of these different components. These effects can be adjusted for different surfaces. Thereby it is possible to define different materials.

The following two sections about OpenGL lighting and materials are an excerpt from the OpenGL "Red Book", Chapter 6 on lighting.[1]

2 Light sources

Emitted light is the simplest form - it is light that originates from an object and is unaffected by any light sources.

The *ambient* component is the light from that source that has been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions. Backlighting in a room has a large ambient component, since most of the light that reaches your eye has bounced off many surfaces first. A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since you're outdoors, very little of the light reaches your eye after bouncing off other objects. When ambient light strikes a surface, it's scattered equally in all directions.

[1] `http://fly.cc.fer.hr/~unreal/theredbook/`

Diffuse light comes from one direction (the direction of the light source), so it's brighter if it comes squarely down on a surface than if it barely glances off the surface. Once it hits a surface, however, it's scattered equally in all directions, so it appears equally bright, no matter where the eye is located. Any light coming from a particular position or direction probably has a diffuse component.

Finally, *specular* light comes from a particular direction, and it tends to bounce off the surface in a preferred direction. A well-collimated laser beam bouncing off a high-quality mirror produces almost 100 percent specular reflection. Shiny metal or plastic has a high specular component, but chalk or carpet has almost none. You can think of specularity as shininess.

Although a light source delivers a single distribution of frequencies, the ambient, diffuse, and specular components might be different. For example, if you have a white light in a room with red walls, the scattered light tends to be red, although the light directly striking objects is white. OpenGL allows you to set the red, green, and blue values for each component of light independently.

3 Material Colors

The OpenGL lighting model makes the approximation that a material's color depends on the percentages of the incoming red, green, and blue light it reflects. For example, a perfectly red ball reflects all the incoming red light and absorbs all the green and blue light. If you view such a ball in white light (composed of equal amounts of red, green, and blue light), all the red is reflected, and you see a red ball. If the ball is viewed in pure red light, it also appears to be red. If, however, the red ball is viewed in pure green light, it appears black (all the green is absorbed, and there's no incoming red, so no light is reflected).

Like lights, materials have different ambient, diffuse, and specular colors, which determine the ambient, diffuse, and specular properties of the material. A material's ambient reflectance is combined with the ambient component of each incoming light source, the diffuse reflectance with the light's diffuse component, and similarly for the specular reflectance and component. Ambient and diffuse reflectances define the color of the material and are typically similar if not identical. Specular reflectance is usually white or gray, so that specular highlights end up being the color of the light source's specular intensity. If you think of a white light shining on a shiny red plastic sphere, most of the sphere appears red, but the shiny highlight is white.

The lighting model used in OpenGL is called *Gouraud shading*, after Henri Gouraud who described it in a paper in 1971 [2]. In this model a color and normal vector is assigned to each of the vertices in a polygon. Light reflected from the surface of a polygon is calculated based on the object's material properties (ambient, diffuse and specular properties), the surrounding light sources and

[2] H. Gouraud, "Continuous shading of curved surfaces," IEEE Transactions on Computers, C-20(6):623-629, 1971.

the location of the camera. Then the colors at each of the vertices are taken and interpolated across the whole polygon.

4 Part 1 - Flat shading

In the framework program, only the ambient component of the light source is working independently of vertex normals. As diffuse and specular lighting require normals, your first task now is to enable *flat shading* all over the scene.

Flat shading means that a single surface normal is calculated for a polygon, and that all vertices defining that polygon are assigned that same normal. As the normal is used for lighting computations this means that the whole polygon reflects light in the same way (because all points on the polygon have roughly the same angle between the surface normal and the light source).

Implement function `calcNormalsFlat` and assign correct vertex normals for each polygon (fill the `normals` array of each `poly`). Remember that normals should have unit length for shading to work correctly!

4.1 Correcting normal directions

When you are finished implementing flat shading, look at the shading of the house. You will probably see that the side walls are not shaded correctly, for example you will see that there are no dark unilluminated walls (or perhaps you have *only* dark walls).

Imagine that if one wall of the house is shown illuminated, the opposite wall should be dark (as the light is not on the front side of that dark wall). But both wall polygons receive the same shading because when they are created the vertices are entered in the same order, and therefore get the same normal vector.

To fix this, you must change the sign of some of the normal vectors, to make sure that the normal always points in the direction of the outside of the object. This way we can guarantee consistent shading.

To find out which normals must be reversed, the polygon data structure provides a vector variable, *inside*. This variable describes the center point of the object, so for example in the case of the house it lies somewhere in the middle, in the case of the sphere *inside* is its center.

The *inside* point can be used to correct normal directions. Try to think up a simple formula that uses the *inside* point and that enables your program to check whether computed normal vectors point to the inside or to the outside of an object. Update your program to make sure the correct normal vectors are assigned to all polygon vertices.

If done correctly, your program should display the scene as shown in Figure 1.

4.2 Hints

The file *normals.c* contains a skeleton for the correct computation of normals.

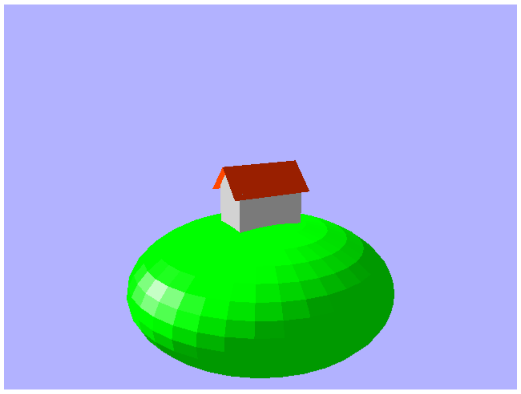

Figure 1: Flat shading.

It may a good idea to begin writing a little algebra library with vector functions. You can use the data type *point* (defined in *polys.h*) for the variables. Later you will find it useful to have functions like $normalizeVector()$, $addVectors()$, $substractVectors()$, $scalarMultiply()$, $dotProduct()$ and $crossProduct()$.

You may assume that all vertices of a polygon lie in the same plane, so you shouldn't have any problems calculating the plane normal.

5 Part 2 - Gouraud shading

Flat shading works fine for flat surfaces like the walls of the house, but it looks ugly for the round sphere surface. The small rectangles that make up the sphere are clearly visible. The problem with the current shading of the sphere is that there is no smooth change of the normal across polygons. For this we can use *Gouraud shading*, which assigns each vertex its own normal vector independent of a polygon's other vertex normals. This will make it possible for the video card rendering the 3D scene to interpolate lighting effects very smoothly over the surface of a curved object, such as the sphere.

There are different ways to compute a normal vector per vertex based on normals of adjacent polygons. Here, you can simply compute an average normal vector, based on the polygons in which a vertex is used. Make sure the resulting normal vector has length one, otherwise the lighting will be influenced.

When you search through the polygon list to spot corresponding vertices, bear in mind that two points could be the same even though they are not completely equal. This can happen due to rounding errors in the floating-point numbers. Therefore, to check whether two points are equivalent it is better to check whether their distance is below some fixed small value epsilon (for example 10^{-6}).

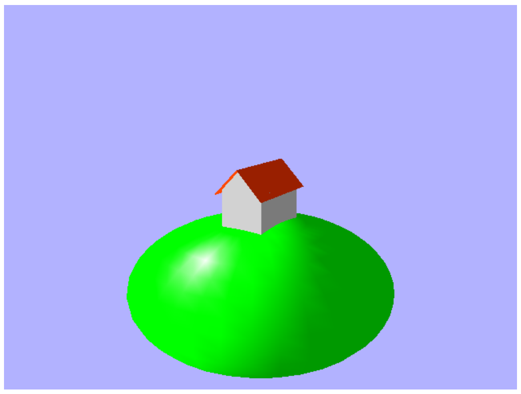

Figure 2: The final result: flat shading (the house) and gouraud shading (the sphere) in action.

6 Grading

You can get up to 4 points for a working implementation of flat shading in your program, and up to 5 points for a working Gouraud shading implementation.

You can get one additional point for writing clean, well-structured and well-commented code (on the opposite side, unreadable or overly complex code might cost you a point).