Tutorial 05

Aditya Prakash

September 20, 2015

1 About

In this tutorial we shade a Sphere, using two methods, Gouraud Shading and PerPixel shading

2 Running the Code

If you have a driver supporting OpenGL 3.2+, you can run the gouraud shading code as

```
./05_gouraud 30
```

here the parameter 30 specifies the amount of subdivisions to the sphere, we'll explain that later. The code will run even without this parameter too. The value of subdivisions may vary from 10 to 80. The code will run with less than 10 subdivisions but it will not be a sphere.

3 Understanding the code

To perform the Shading we would need to pass the normals to the various vertices in the sphere too. As the color depends on the orientation of vertex compared to light. Where the vertex lies compared to light. So we add normals to attributes of our shaders. In order to pass the normals, we create a normal_matrix and add normals of the vertices to it. Now since the sphere is centered at origin, the normals will be same as the coordinates of the vertex. So, for every point(x, y, z) we do the following,

```
  glm::vec4 pt(x, y, z, 1.0);
  v_colors[tri_idx] = white; v_positions[tri_idx] = pt;
  v_normals[tri_idx] = pt; tri_idx++;
```

Also, we create a uniform normal matrix, which is used in order to transform normals. We add the required code for initializing the normalMatrix, getting its position in shader, and do the following,

```
  normal_matrix = glm::transpose (glm::inverse(glm::mat3(
     modelview_matrix)));
  glUniformMatrix3fv(normalMatrix, 1, GL_FALSE, glm::
     value_ptr(normal_matrix));
```

We set the value of normal matrix as transpose of inverse of modelview_matrix. Why we do this? To transform normal. Since, vertices have transformed from model space to world space, in doing so they could have been rotated, translated and scaled(depending upon modelview_matrix). But the normals are still in model space. We multiply the normals to the above calculated matrix in shader. To ensure that the finally transformed normal is still a unit vector, we use the *normalize()* function in the shader.

3.1 Drawing Sphere

The piece of code for drawing sphere basically uses the polar coordinates of a sphere to draw the points. If you recall polar coordinates, we can express x,y,z on surface of a sphere as,

$$x = radius * cos\theta * sin\phi$$

$$y = radius * sin\theta * sin\phi$$

$$z = radius * cos\phi$$

The variable slices specify the number of different planes on the sphere(including poles) while sectors, specify number of different points on each plane.

How we draw the sphere is simple, in one iteration we plot one point on one plane and next point on the following plane. Hence making a zig-zag pattern between two planes. We iterate the value of θ from 0 to π and ϕ from 0 to 2π.

```
void sphere(double radius, int Lats, int Longs)
{
```

```
float slices=(180/(float(Lats)*10))/2;
float sectors=(180/(float(Longs)*10))/2;

float l;

for (lats = 0.0; lats <= PI; lats+=sectors)
{
    for(longs = 0.0; longs <= 2.0*PI; longs+=slices)
  {
    float x = radius * sin(lats) * cos(longs);
    float y = radius * sin(lats) * sin(longs);
    float z = radius * cos(lats);
    glm::vec4 pt(x, y, z, 1.0);

    v_colors[tri_idx] = white; v_positions[tri_idx] = pt;
    v_normals[tri_idx] = pt; tri_idx++;

    w_colors[wire_idx] = black; w_positions[wire_idx] = pt;
    w_normals[wire_idx] = pt; wire_idx++;


    if(lats+sectors>PI)
      l=PI;
    else
      l=lats+sectors;
    x = radius * sin(l) * cos(longs);
    y = radius * sin(l) * sin(longs);
    z = radius * cos(l);
    pt =glm::vec4(x, y, z, 1.0);
    v_colors[tri_idx] = white; v_positions[tri_idx] = pt;
    v_normals[tri_idx] = pt; tri_idx++;

    w_colors[wire_idx] = black; w_positions[wire_idx] = pt;
    w_normals[wire_idx] = pt; wire_idx++;

  }
  }
// To Complete the wireframe
for (lats = 0.0; lats <= PI; lats+=sectors)
  {
    for(longs = 0.0; longs <= 2.0*PI; longs+=slices)
  {
    float x = radius * sin(lats) * cos(longs);
    float y = radius * sin(lats) * sin(longs);
    float z = radius * cos(lats);
```

```
        glm::vec4 pt(x, y, z, 1.0);

        w_colors[wire_idx] = black; w_positions[wire_idx] = pt;
        w_normals[wire_idx] = pt; wire_idx++;

    }
    }
}
```

Note that there is an addtional loop at the end, what this loop does is basically draw circles for the wireframe. That loop draws a circle for each plane on the sphere.

4 Shaders

I highly recommened that you read up the slides on shading and get your head around the basics of computing the colors. Essentially, in gouraud shading the color computation is on vertices of polygons, while in per-pixel shading, the color for each pixel is computed individually, hence a lot more computation compared to Gouraud shading.

4.1 Shaders in Gouraud Shading

In gouraud shading we will compute the colors at vertices of polygons and interpolate the colors throughout. This will be done in the vertex shader itself. The fragment shader in Gouraud shading will remain untouched.

4.1.1 Vertex Shader

In the main function we start with defining the materials for the surface. Note that, since we have only one object so we are defining the material in the shader itself. If you want to create more materials, you can define them in cpp, and pass them to shader, you can pass different components separately, or you can pass them as single structure. Same goes for the lighting also. So the first few lines specify the material diffuse, ambient, specular and shininess. Essentially the final color is a combination of these components.

```
  // Defining Materials
  vec4 diffuse = vec4(0.5, 0.0, 0.0, 1.0);
```

```
vec4 ambient = vec4(0.1, 0.0, 0.0, 1.0);
vec4 specular = vec4(1.0, 0.5, 0.5, 1.0);
float shininess = 0.05;
```

In the following lines we specify the light position to be at 1.0, 1.0, 1.0. So, we will cast the light from the point (1.0, 1.0, 1.0). We follow it up by multiplying the light position to view matrix. We do this so that when we move the object, the light doesn't move. As, it is supposed to be stationary.

```
// Defining Light
vec4 lightPos = vec4(1.0, 1.0, 1.0, 0.0);
vec3 lightDir = vec3(viewMatrix * lightPos);
lightDir = normalize(lightDir);
```

The next line is the same as we have been doing in the past, assigning the vertex positions. The following steps, we compute the diffuse component of light,

```
vec3 n = normalize(normalMatrix * normalize(vNormal));
float dotProduct = dot(n, lightDir);
float intensity =  max( dotProduct, 0.0);
```

The first line is already explained before. In the next line we compute the intensity(in variable dotProduct) of diffuse component of color which is essentiallyL.N (L is light position and N is Normal). Nextly we make sure that the intensity is not negative, as if the light doesn't fall on a vertex we just set the intensity at that vertex to 0(Black).

Now, we need to compute the specular component only if light falls on the vertex, otherwise, we can just ignore specular compuation. The following lines do, the specular computation.

```
vec3 eye = normalize( vec3(-gl_Position));
vec3 h = normalize(lightDir + eye );
float intSpec = max(dot(h,n), 0.0);
spec = specular * pow(intSpec, shininess);
```

Now finally we assign the final color of pixel simply as sum of diffuse component(intensity*diffuse) and specular component (spec). We can also add the ambient component to this. Note that we are assigning it to max((intensity * diffuse + spec)*vColor, ambient). This is to make sure that if a pixel is on screen it gets some color. Next thing to note is that we multiply the vColor to the computed color, this is a simple blending, we can assign weights of vColor and get different blending.

```
  color = max((intensity * diffuse  + spec)*vColor, ambient*
     vColor);
```

4.2 Shaders in Per-Pixel Shading

Now the computation of color remains same in per-pixel shading, except the fact that here it is done in the fragment shader. i.e. We compute color of each pixel individually.

4.2.1 Vertex Shader

In the vertex shader we compute the modified normal, and eye as previously and pass these to fragment shader.

4.2.2 Fragment Shader

Here, initially, we create materials and lights as previously. We follow it by the diffuse and specular computation. And finally we assign the computed colors and blend it with input color to the frag shader. We, don't do any additional computation here. We just do the same computation as before and get much better results

5 Output

You can see the difference in the output of both shading methods. Note that the specular in the output of gouraud shading seems peculiar. But to if you see it with the wireframe on, you can see that the specular is being applied to complete polygons as can be seen in the following output.

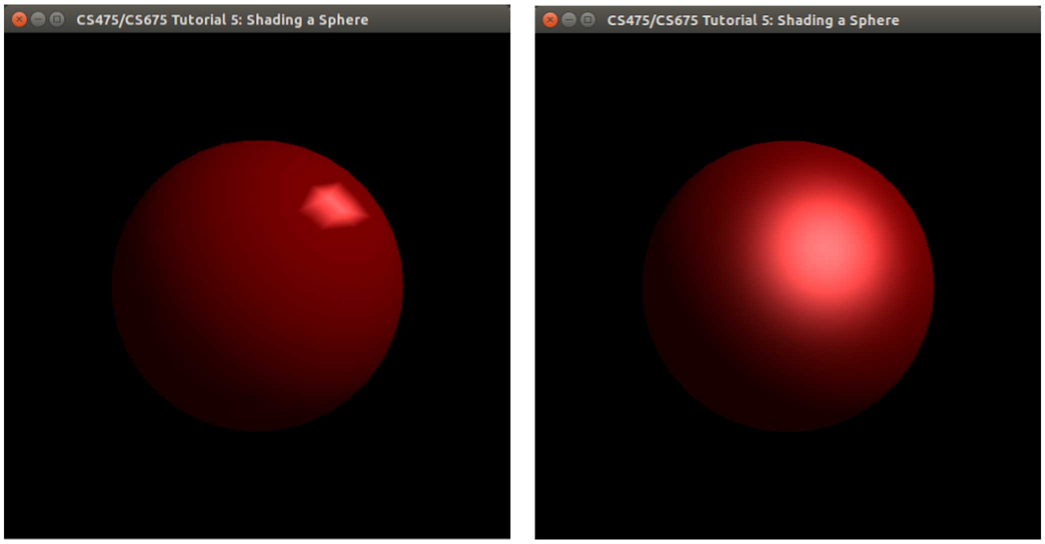
CS475/CS675 Tutorial 5: Shading a Sphere
CS475/CS675 Tutorial 5: Shading a Sphere

Figure 1: Outputs of gouraud shading (left) and per-pixel shading(right)

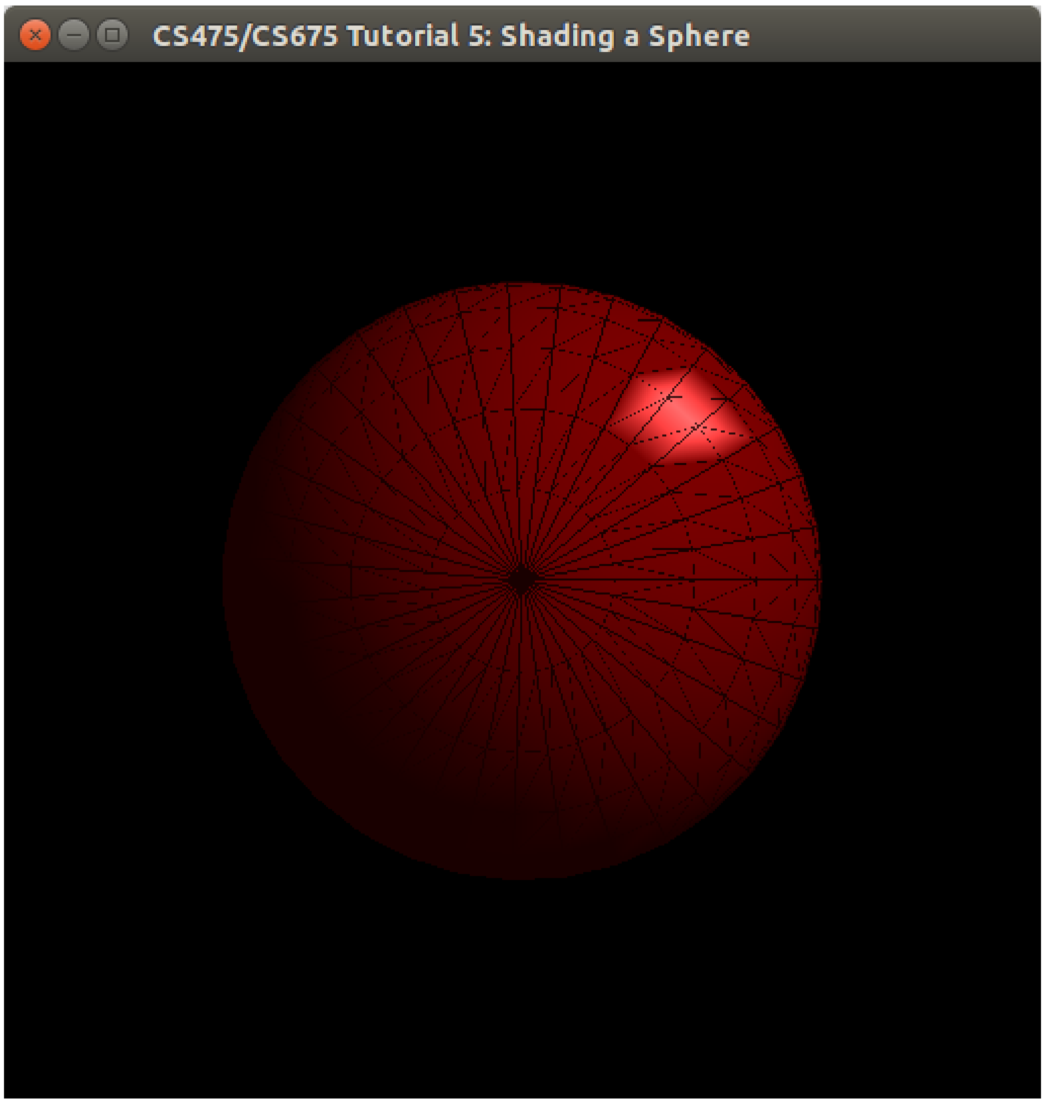

Figure 2: Output of gouraud shading with wireframe