Assignment 4: Shading

Mai Trinh

COSC 6372 Computer Graphics[*], University of Houston

April 8, 2021

1 Problem

The goal of this assignment is to add the shading function into the *Gz* library. Given an input file containing the information of several triangles including the coordinates of 3 vertices, and its color, the application should be able to apply Gouraud shading and Phong shading to the image. Several classes, functions, and data types are already given in the *Gz* library. We were to complete the implementation of the functions in *Gz* class and *GzFrameBuffer* class to enable the shader and produce the expected output images.

2 Methods

The main challenge of this assignment is to implement functions to support light sources, more specially, we need to implement Gouraud shading and Phong shading. The goal is to produce the image under different lighting. To do so, we need to add new drawing functions that deal with lighting. To do so, we first need to understand the basic pipeline for Gouraud shading and Phong shading.

Gouraud shading is a per-vertex color computation. That is the shader function uses light information to determine a color for each vertex and linearly interpolate across the surface to give a smooth appearance [2]. Pipeline for Gouraud shading is as follow:

- The shader takes the normal vectors, the light information, and vertex color to compute a new vertex color with illumination according to the Phong lighting model.
- Interpolate colors from vertices across triangle.

Phong shading is a per-fragment (pixel) lighting. It interpolates surface normals across rasterized polygons and computes pixel colors based on the interpolated normals and a reflection model [5]. This is a more accurate method and computationally expensive for rendering compare to Gouraud shading. The pipeline for Phong shading is shown below [5].

- Normal vector is linearly interpolated across the surface of the triangle from the triangle's vertex normals.
- The surface normal is interpolated and normalized at each pixel and used in a reflection model to get the final pixel color.

3 Implementation

Assume that the task of reading input from a file and storing it in appropriate vectors are correctly implemented in the main application. The coordinates *(x,y,z)* of each pixel is stored in *vertexQueue*, the normal vector of the vertex is stored in *normalQueue* and the color of each pixel is stored in *colorQueue* in *Gz* library. When GZ_LIGHTING is enabled, we pass to the framebuffer the light source information as well as vertex information including coordinates, normal, and colors. Note that we also need to transform the normal vectors and the light sources based on the transformation matrix.

3.1 Transform normal vectors and light sources

Since we apply transformation to the vertex, we also need to transform the normal vector. To do so, simply multiply the normal vector with the the inverse transpose of the upper 3x3 submatrix of the transformation matrix. The reason for this being that normal vector is perpendicular to the surface, thus we need the inverse transpose to retain that property. Since normal vectors are direction and will not be translated, we only need the upper 3x3 matrix of the translation matrix.

$$N_{transformed} = Inverse(transpose(transMatrix_{upper3x3})) * N$$

The implementation for transformaing normal vector is shown below.

*Course assignments

```
GzVector Gz::transformNorm(const GzVector& norm) {
    GzVector transNorm;
    GzMatrix M;
    M.resize(3, 3);
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            M[i][j] = transMatrix[i][j];}}

    M = M.inverse3x3().transpose();
    transNorm[0] = M[0][0] * norm[0] + M[0][1] * norm[1] + M[0][2] * norm[2];
    transNorm[1] = M[1][0] * norm[0] + M[1][1] * norm[1] + M[1][2] * norm[2];
    transNorm[2] = M[2][0] * norm[0] + M[2][1] * norm[1] + M[2][2] * norm[2];
    return transNorm;
}
```

Similarly, we can perform the tranformation for all of the light sources. Note that we only transform the light source direction, not the color.

3.2 Shader function

The shader function is the most important aspect of this assignment. The goal of the shader function is to compute the surface's color at a particular shading point. This is developed based on Phong Reflectance Model [4].

There are three basis types of lighting: ambient lighting, diffuse lighting and specular lighting. The final color result from the shading function of different lights is the combination of these 3 lighting type. Note that we are given the light sources informations including the light direction, light color as well as the material property: kA (ambient coefficient), kD (diffuse coefficient), kS (specular coefficient), s (specular power). Also note that the light sources have been transformed following the previous section. First, ambient lighting is light that comes from all direction. Think of it as a non-direction light that pervades the entire scene. The amount of ambient light incident on each object is constant for all surfaces and over all directions. Thus, the ambient illumination can be computed as

$$L_a = kA * \text{material color} \qquad [1]$$

In diffuse lighting, light is scattered uniformly in all directions. Thus, the surface appears equally bright from all viewing directions. To compute diffuse reflected light, we use Lambertian shading. We first need to calculate the point light intensity I_p, in this case, it's the color of the light. Since we only consider directional lights from infinity, then we need to get the the direction direction L from vector to light source. Since we are given the direction of the light, we just need to invert it so that it becomes the direction from vector to light source. We also need the surface normal N, which is given by the main program and then transformed by the previous section. Then we can compute the diffuse reflected light as:

$$L_d = kD * I_p * dotProduct(N, L) \qquad [1]$$

Note that since the dot product can have a range of -1 to 1, we need to clamp it to a range of 0 to 1. Finally, specular lighting provides the appearance of highlight, i.w. a reflect of the strongest sources of light. Imagine modeling shiny and glossy surfaces like metal. Reflectance intensity depends on view direction, that is it changes with reflected angle. To compute specular reflected light, we need viewer direction V, which can be calculated as the distance between eye to light source. We also need reflection direction R, which can be computed as $R = 2 * \text{dotProduct(L,N)} * N - L$. Finally, the equation to compute the specular reflected light is:

$$L_s = kS * I_p * (dotProduct(R, V))^s \qquad [4]$$

Finally, the final color is the combination for all 3 reflection model: $color = L_a + L_d + L_s$. The implementation for the shader function is shown below.

```
GzColor GzFrameBuffer::shader(const GzColor& c, const GzVector& n) {
    GzColor color;
    // Ambient light - light that comes from all directions
    for (int i = 0; i < 4; i++)
        color[i] = c[i] * kA;

    for (int i = 0; i < transLightSources.size();i++) {
        GzColor lightColor = transLightSources[i].col;

        // direction vector
        GzVector L = - transLightSources[i].dir;
        L.normalize();
        GzVector eye = GzVector(-_transMatrix[0][3], -_transMatrix[1][3], -_transMatrix[2][3]);
        eye.normalize();

        // viewer direction
        GzVector V = L + eye;
        V.normalize();

        // Calculate the reflection direction for an incident vector
        GzReal lightIntensity = dotProduct(n, L);
        GzVector R = 2 * lightIntensity * n - L;
        R.normalize();

        for (int j = 0; j < 4; j++) {
            // Diffuse light
            GzReal L_d = kD * lightColor[j] * max(dotProduct(n, L), 0.0);
            L_d = clamp(L_d, 0.0, 1.0);
            color[j] += L_d;

            // Specular light
            GzReal L_s = lightColor[j] * kS * pow(max(dotProduct(R, V), 0.0), s);
            L_s = clamp(L_s, 0.0, 1.0);
            color[j] += L_S;}
    }
    return color;}
```

3.3 Gouraud Shading

As mention in the methods section, to perform Gouraud shading we used in a reflection model to get the final vertex color. To do so, we utilize the shader function that was described above. Once we get a new color, we passed it to *drawTriangle* function (provided). This function will interpolate the color from vertices across the triangle. I will not discuss the implementation of *drawTriangle* function since it was provided.

```
    vector<GzColor> color(3);
    for (int i = 0; i < 3; i++)
        color[i] = shader(c[i], n[i]); \\ use shader function to get new color
    drawTriangle(v, color, status);
```

3.4 Phong Shading

As mention above, Phong shading is per-pixel lighting. Thus, for every pixel, we performing the following step: linearly interpolate the normal vector across the surface of the triangle from the triangle's vertex normal, then use in a reflection model to get the final pixel color. Then apply the illumination model along each scan line.

3.4.1 Vector interpolation

The surface normal is interpolated and normalized at each pixel. Use the linear interpolation between 2 known points method to perform vector interpolation [3]. The implementation is shown below.

```
// Similar to colorInterpolate with minor changes
void GzFrameBuffer::vectorInterpolate(GzReal key1, GzVector& val1, GzReal key2, GzVector& val2,
                                      GzReal key, GzVector& val) {
    GzReal k = (key - key1) / (key2 - key1);
    for (GzInt i = 0; i < 3; i++)
        val[i] = val1[i] + (val2[i] - val1[i]) * k;
    val.normalize();}
```

3.4.2 Apply the illumination model

For this implementation, I utilize the provided functions *drawPoint, drawTriangle, drawRasLine* and apply additional changes to support lighting model. I will discuss in detail the changes applied to the new *drawRasLine* method, the other 2 methods follow a similar approach, so I will not discuss it in this report. The change to this function compare to the traditional method without lighting is the addition of normal vector interpolation and the use of shader function. As discuss above, Phong shading require the surface normal vector to be linearly interpolated across the surface of the triangle. Which is then be used in the shader function to get the final pixel color. The implementation is shown below.

```
void GzFrameBuffer::drawRasLine(GzInt y, GzReal xMin, GzReal zMin, GzColor& cMin, GzVector& nMin,
                        GzReal xMax, GzReal zMax, GzColor& cMax, GzVector& nMax, GzFunctional status) {

    if ((y < 0) || (y >= image.sizeH())) return;
    if ((GzInt)floor(xMin) == (GzInt)floor(xMax)) {
        if (zMin > zMax) drawPoint(GzVertex(floor(xMin), y, zMin), cMin, nMin, status);
        else drawPoint(GzVertex(floor(xMin), y, zMax), cMax, nMax, status);}
    else {
        GzReal z;
        GzColor c;
        GzVector n;

        y = image.sizeH() - y - 1;
        int w = image.sizeW();
        if (status & GZ_DEPTH_TEST) {
            for (int x = max(0, (GzInt)floor(xMin)); x <= min(w - 1, (GzInt)floor(xMax)); x++) {
                realInterpolate(xMin, zMin, xMax, zMax, x, z);
                if (z >= depthBuffer[x][y]) {
                        colorInterpolate(xMin, cMin, xMax, cMax, x, c);
                        vectorInterpolate(xMin, nMin, xMax, nMax, x, n);
                        c = shader(c, n);
                        image.set(x, y, c);
                        depthBuffer[x][y] = z;}}}
        else {
            for (int x = max(0, (GzInt)floor(xMin)); x <= min(w - 1, (GzInt)floor(xMax)); x++) {
                realInterpolate(xMin, zMin, xMax, zMax, x, z);
                colorInterpolate(xMin, cMin, xMax, cMax, x, c);
                vectorInterpolate(xMin, nMin, xMax, nMax, x, n);

                c = shader(c, n);
                image.set(x, y, c);
                depthBuffer[x][y] = z;}}}}
```

4 Results

The results of the implementation are shown below. 2 image in the top row displays the result of Gouraud shading, and the 2 images on the bottom row displays the result of Phong shading. From the results, given the same settings (i.e light sources, transformation, position, etc.) we can see that Phong shading produces smoother lighting results. In Gouraud shading, since it's per-vertex lighting, the specular highlights appear to jump from vertex to vertex, thus we can clearly see a line where the specular highlight changes. As for Phong shading, since it's per-pixel lighting, we see a smoother appearance, there is no obvious line that indicate the changes in specular lighting.

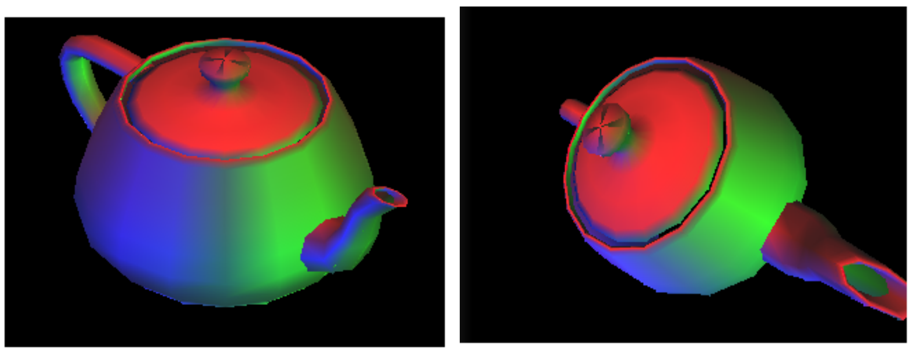
Figure 1: Gouraud shading - Result images

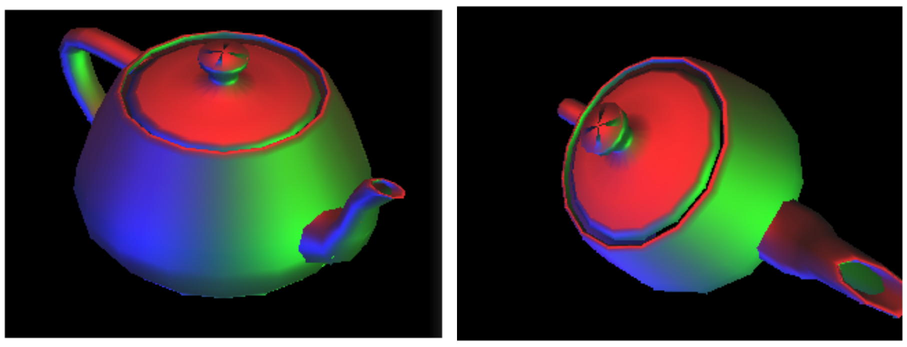
Figure 2: Phong shading - Result images

References

[1] *Android Lesson Two: Ambient and Diffuse Lighting.* URL: http://www.learnopengles.com/android-lesson-two-ambient-and-diffuse-lighting/.

[2] *Gouraud shading.* URL: https://en.wikipedia.org/wiki/Gouraud_shading.

[3] *Linear interpolation.* URL: https://en.wikipedia.org/wiki/Linear_interpolation.

[4] *Phong reflection model.* URL: https://en.wikipedia.org/wiki/Phong_reflection_model.

[5] *Phong shading.* URL: https://en.wikipedia.org/wiki/Phong_shading.