



中山大學
SUN YAT-SEN UNIVERSITY

本科生实验报告

实验课程: 计算机图形学

专业名称: 计算机科学与技术

学生姓名: _____

学生学号: _____

教学班级: 计科 3 班

实验成绩: _____

报告时间: 2023 年 12 月 21 日

1 实验任务

1. 使用 DDA 实现三角形边的绘制；
2. 使用 bresenham 实现三角形边的绘制；
3. 使用 edge-walking 填充三角形内部颜色；
4. 使用 Gouraud 模型实现三角形内部的着色；
5. 使用 Phong 模型实现三角形内部的着色；
6. 使用 Blinn-Phong 模型实现三角形内部的着色；
7. 讨论不同模型方法的渲染效果以及运行效率。

2 环境搭建

本次作业的实验环境搭建基于作业 1 的环境，因此省略相关软件的安装以及系统环境的配置。基本搭建步骤如下：

(1) 由于头文件 GL/glew.h 的缺失，我们需要配置 GL/glew.h。在 Qt 中打开 CGTemplate.pro 文件，添加以下代码：

```
1 INCLUDEPATH += "D:\OpenGL\glew-2.1.0\include"
```

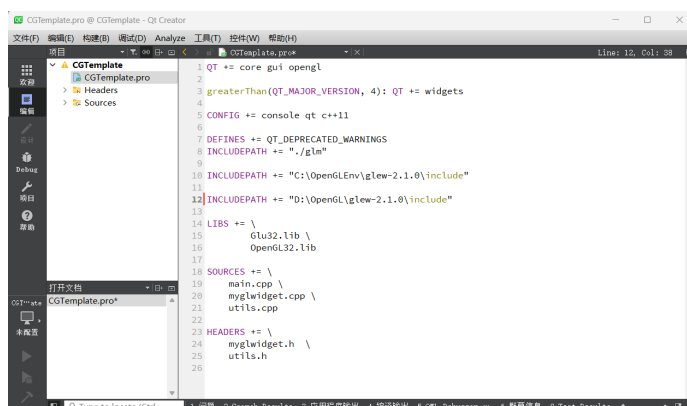
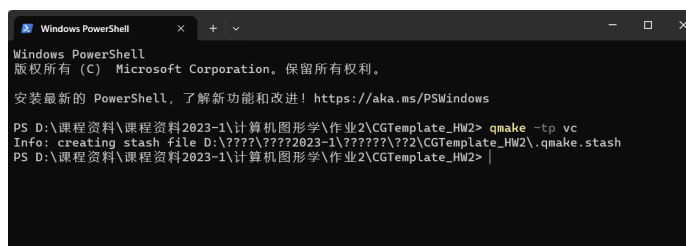


fig 1: 配置 GL/glew.h

(2) 进入工程文件所在的文件夹，打开命令提示符，键入命令 ‘qmake -tp vc’，生成 VS 工程文件。



```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

安装最新的 PowerShell，了解新功能和改进！https://aka.ms/PSWindows

PS D:\课程资料\课程资料2023-1\计算机图形学\作业2\CGTemplate_Hw2> qmake -tp vc
Info: creating stash file D:\????\????2023-1\????\772\CGTemplate_Hw2\.qmake.stash
PS D:\课程资料\课程资料2023-1\计算机图形学\作业2\CGTemplate_Hw2>
```

fig 2: 键入命令：qmake -tp vc 生成 VS 工程文件

(3) 使用 Visual Studio 打开.vcxproj 文件，运行模板文件并保存。

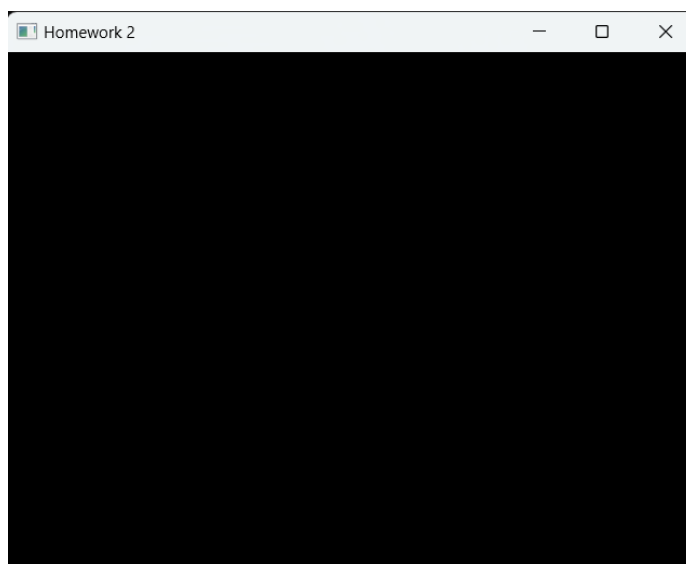


fig 3: Demo 输出结果

至此，环境配置的工作全部完成。

3 实验内容

3.1 DDA 实现边的绘制

3.1.1 算法原理

知直线斜率为 $K = \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$ ，则由 x_i, y_i 可得到 $x_{i+1} = x_i + \Delta x$ ， $y_{i+1} = y_i + K \cdot \Delta x$ 。再由 y_{i+1} 邻近取整，即可得到 x_{i+1} 所对应的 \hat{y}_{i+1} 。

为了降低画线误差, 当 $|y_{end} - y_{start}| \geq |x_{end} - x_{start}|$ 时, 我们沿着 y 轴方向以公式 $\hat{x}_{i+1} = [x_i + \frac{1}{K} \cdot \Delta x]$ 迭代求取 x 值; 当 $|y_{end} - y_{start}| < |x_{end} - x_{start}|$ 时, 我们沿着 x 轴方向以公式 $\hat{y}_{i+1} = [y_i + K \cdot \Delta x]$ 迭代求取 y 值。

3.1.2 算法代码

```

1 void MyGLWidget::DDA(FragmentAttr& start, FragmentAttr
    & end, int id) {
2     int x0 = start.x, x1 = end.x;
3     int y0 = start.y, y1 = end.y;
4     int dx = abs(x1 - x0);
5     int dy = abs(y1 - y0);
6     int sx = (x1 >= x0) ? 1 : -1; //x增量
7     int sy = (y1 >= y0) ? 1 : -1; //y增量
8     float x = start.x, y = start.y, ix, iy;
9
10    if (x < WindowSizeW && x >= 0 && y < WindowSizeH &&
        y >= 0) {
11        temp_render_buffer[WindowSizeW * (int)y + (int)x]
            = vec3(0.0f, 0.0f, 1.0f);
12        FragmentAttr interpolation =
            getLinearInterpolation(start, end, x);
13        temp_z_buffer[WindowSizeW * (int)y + (int)x] =
            interpolation.z;
14        updateRecorder((int)x, (int)y);
15    }
16
17    if (x0 == x1) { //垂直
18        for (int i = 0; i < dy; i++) {
19            y += sy;
20            if (x < WindowSizeW && x >= 0 && y < WindowSizeH
                && y >= 0) {
21                temp_render_buffer[WindowSizeW * (int)y + x0]
                    = vec3(0.0f, 0.0f, 1.0f);
22                FragmentAttr interpolation =

```

```
        getLinearInterpolation(start, end, x);
23     temp_z_buffer[WindowSizeW * (int)y + x0] =
        interpolation.z;
24     updateRecorder(x0, (int)y);
25     }
26 }
27 }
28
29 else if (y0 == y1) { // 水平
30     for (int i = 0; i < dx; i++) {
31         x += sx;
32         if (x < WindowSizeW && x >= 0 && y < WindowSizeH
            && y >= 0) {
33             temp_render_buffer[WindowSizeW * y0 + (int)x]
                = vec3(0.0f, 0.0f, 1.0f);
34             FragmentAttr interpolation =
                getLinearInterpolation(start, end, x);
35             temp_z_buffer[WindowSizeW * y0 + (int)x] =
                interpolation.z;
36             updateRecorder((int)x, y0);
37         }
38     }
39 }
40
41 else if (dx >= dy) { // 偏向 x 轴
42     iy = sy * dy / (float)dx;
43     for (int i = 0; i < dx; i++) {
44         x += sx;
45         y += iy;
46         if (x < WindowSizeW && x >= 0 && y < WindowSizeH
            && y >= 0) {
47             temp_render_buffer[WindowSizeW * (int)y + (int)
                x] = vec3(0.0f, 0.0f, 1.0f);
48             FragmentAttr interpolation =
```

```

        getLinearInterpolation(start, end, x);
49     temp_z_buffer[WindowSizeW * (int)y + (int)x] =
        interpolation.z;
50     updateRecorder((int)x, (int)y);
51 }
52 }
53 }
54 else if (dx < dy) { // 偏向 y 轴
55     ix = sx * dx / (float)dy;
56     for (int i = 0; i < dy; i++) {
57         y += sy;
58         x += ix;
59         if (x < WindowSizeW && x >= 0 && y < WindowSizeH
            && y >= 0) {
60             temp_render_buffer[WindowSizeW * (int)y + (int)
                x] = vec3(0.0f, 0.0f, 1.0f);
61             FragmentAttr interpolation =
                getLinearInterpolation(start, end, x);
62             temp_z_buffer[WindowSizeW * (int)y + (int)x] =
                interpolation.z;
63             updateRecorder((int)x, (int)y);
64         }
65     }
66 }
67 }

```

考虑到可能会出现片段在画布外的情况，我们在对 temp_render_buffer 和 temp_z_buffer 进行赋值前应添加以下 if 条件，以确保坐标位于画布内并防止数组越界。

```

1  if (x < WindowSizeW && x >= 0 && y < WindowSizeH && y
    >= 0)

```

3.1.3 实现效果

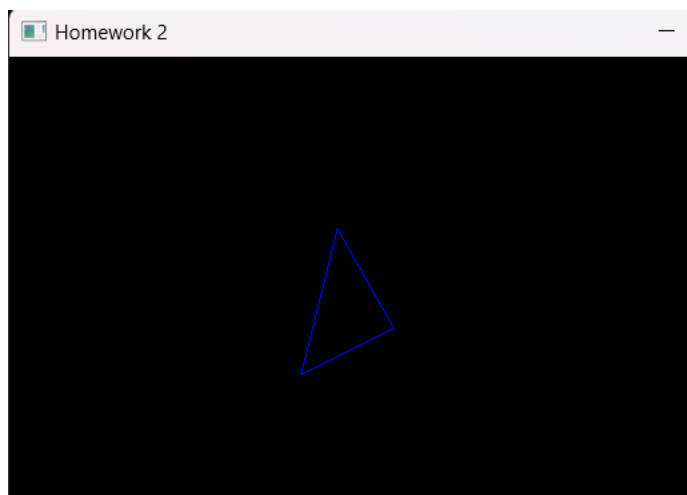
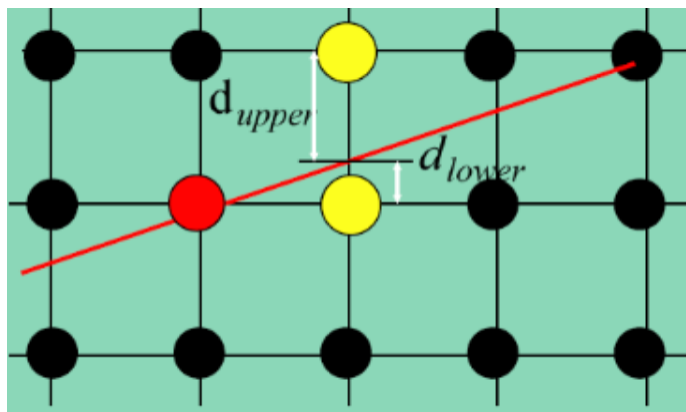


fig 4: DDA 绘制边

3.2 bresenham 实现边的绘制

3.2.1 算法原理

首先考虑斜率大于 0 且小于等于 1 的情况，从顶点开始 x 每次递增一个单位， y 则增加不到一个单位，如图所示。此时将理想 y 值与上下两像素点的距离为 d_{upper} 和 d_{lower} ，选取距离较小的点作为光栅化结果。



$$\begin{aligned}
 d_{upper} &= \hat{y}_i + 1 - \hat{y}_{i+1} \\
 d_{lower} &= y_{i+1} - \hat{y}_i \\
 d_{lower} - d_{upper} &= 2m(x_i + 1) - 2\hat{y}_i = 2B - 1
 \end{aligned}$$

两边同时乘以 Δx 得到累计误差 $P_i = 2\Delta y x_i - 2\Delta x \hat{y}_i + C$, 当 $P_i \leq 0$ 时选取下方的点, 否则选择上方的点。又知 $P_{i+1} - P_i = 2\Delta y - 2\Delta x(\hat{y}_{i+1} - \hat{y}_i)$ 。最终得到:

$$\begin{aligned}
 P_{i+1} &= P_i + 2\Delta y, \text{ 当 } P_i \leq 0 \\
 P_{i+1} &= P_i + 2\Delta y - 2\Delta x, \text{ 当 } P_i > 0
 \end{aligned}$$

因此我们可以通过迭代计算 P_i , 判断像素点的选取。进一步拓展, 当斜率为负数时, 将迭代步长改为-1; 当斜率绝对值大于 1 时, 将迭代计算公式中的 x 和 y 交换位置, 即在 y 轴方向上步进迭代计算 x 值。

3.2.2 算法代码

```

1 void MyGLWidget::bresenham(FragmentAttr& start,
    FragmentAttr& end, int id) {
2     // 根据起点、终点, 计算当前边在画布上的像素
3     int x0 = start.x, x1 = end.x, x = start.x;
4     int y0 = start.y, y1 = end.y, y = start.y;
5     int dx = abs(x1 - x0);
6     int dy = abs(y1 - y0);
7     int sx = (x1 >= x0) ? 1 : -1; //x增量
8     int sy = (y1 >= y0) ? 1 : -1; //y增量
9     int flag;
10
11     if (dy <= dx) {
12         flag = 2 * dy - dx;
13         for (int i = 0; i <= dx; i++) {
14             if (x < WindowSizeW && x >= 0 && y < WindowSizeH
                && y >= 0) {
15                 temp_render_buffer[WindowSizeW * y + x] = vec3
                    (0.0f, 1.0f, 0.0f);
16                 FragmentAttr interpolation =
                    getLinearInterpolation(start, end, x);
            }
        }
    }
}

```



```
17         temp_z_buffer[WindowSizeW * y + x] =
18             interpolation.z;
19         updateRecorder(x, y);
20     }
21     x += sx;
22     if (flag < 0) {
23         flag += 2 * dy;
24     }
25     else {
26         y += sy;
27         flag += 2 * dy - 2 * dx;
28     }
29 }
30 else {
31     flag = 2 * dx - dy;
32     for (int i = 0; i <= dy; i++) {
33         if (x < WindowSizeW && x >= 0 && y < WindowSizeH
34             && y >= 0) {
35             temp_render_buffer[WindowSizeW * y + x] = vec3
36                 (0.0f, 1.0f, 0.0f);
37
38             FragmentAttr interpolation =
39                 getLinearInterpolation(start, end, x);
40             temp_z_buffer[WindowSizeW * y + x] =
41                 interpolation.z;
42
43             updateRecorder(x,y);
44         }
45         y += sy;
46         if (flag < 0) {
47             flag += 2 * dx;
48         }
49         else {
50             x += sx;
51             flag += 2 * dx - 2 * dy;
52         }
53     }
54 }
```

```
46         x += sx;
47         flag += 2 * dx - 2 * dy;
48     }
49 }
50 }
51 }
```

考虑到可能会出现片段在画布外的情况，我们在对 `temp_render_buffe` 和 `temp_z_buffer` 进行赋值前应添加以下 `if` 条件，以确保坐标位于画布内并防止数组越界。

```
1  if (x < WindowSizeW && x >= 0 && y < WindowSizeH && y
    >= 0)
```

3.2.3 实现效果

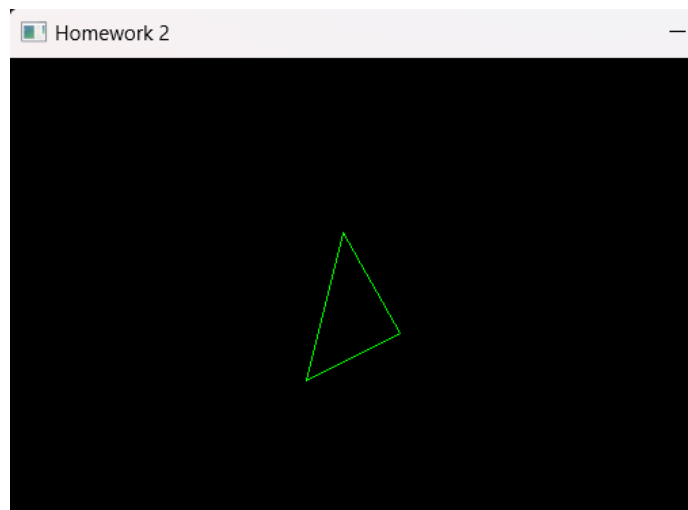


fig 5: bresenham 绘制边

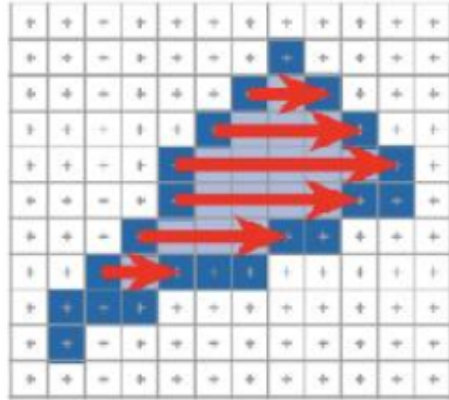
3.3 edge-walking 实现内部颜色的填充

3.3.1 算法原理

在 `edge-walking` 中，我们需要计算三角形内部的每个片段的着色。整体来说，着色方法基于线性插值，并利用相关光照模型计算颜色值。

首先在边绘制的过程中，我们建立二维数组 `edge-recorder`，记录边绘制过程中每一高度 (y) 上最左侧的像素点 x 值 x_{left} ，以及最右侧的像素点 x 值

x_{right} 。然后在 edge-walking 中从上到下、从左到右进行遍历，填充每一高度上 x_{left} 和 x_{right} 之间的像素，如图所示。



3.3.2 算法代码

```

1  int MyGLWidget::edge_walking(FragmentAttr& a,
    FragmentAttr& b, FragmentAttr& c) {
2      // 遍历 edge_recorder 在不同高度的起点、终点，用
    shading model 计算内部每个像素的颜色
3      for (int i = 0; i < WindowSizeH; i++) {
4          if (edge_recorder[i][2] >= 2) {
5              for (int j = edge_recorder[i][0] ; j <
                edge_recorder[i][1]; j++) {
6                  float cur_z = getLinearInterpolation_z(a, b, c
                    , j, i);
7                  vec3 cur_color = getLinearInterpolation_color
                    (a, b, c, j, i);
8                  temp_render_buffer[WindowSizeW * i + j] =
                    cur_color;
9                  temp_z_buffer[WindowSizeW * i + j] = cur_z;
10             }
11         }
12     }
13     return min(min(a.y,b.y),c.y);

```

```
14 }

```

3.3.3 实现效果

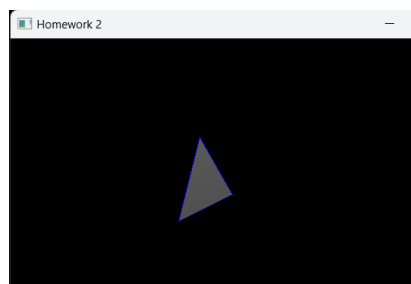


fig 6: DDA-EdgeWalking

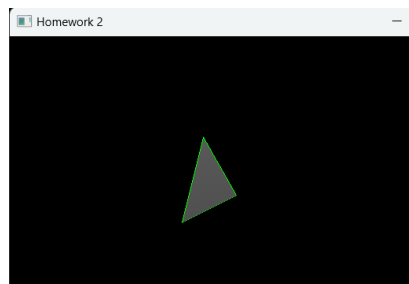


fig 7: bresenham-EdgeWalking

fig 8: EdgeWalking 实现效果

3.4 DDA 与 bresenham 的绘制效率

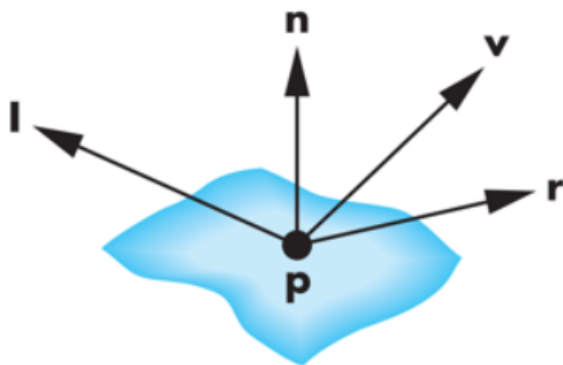
由于 DDA 和 bresenham 都沿着 x 或 y 轴方向以 1 为步进长度进行迭代，因此迭代总次数基本相同。但是 DDA 采用浮点数计算，而 bresenham 采用整型数计算，因此 DDA 的计算开销高于 bresenham。

3.5 光照模型

3.5.1 算法原理

在本次实验中，我们所用光照模型为局部光照模型，采用单点光源，整体光照包括环境光照、漫反射光照和镜面反射光照。

如图所示，曲面上 p 点的光照效果用以下几个向量进行描述：曲面法向量 \mathbf{n} ，视线方向 \mathbf{v} ，光线入射方向 \mathbf{l} ，光线反射方向 \mathbf{r} 。



环境光照公式：

$$I_{ambient} = k_a L_a$$

其中， k_a 表示环境光照系数， L_a 表示环境光照强度。

漫反射光照公式：

$$I_{diffuse} = \frac{1}{a+bd+cd^2} (k_d L_d \max(\mathbf{l} \cdot \mathbf{n}, 0))$$

其中， k_d 表示漫反射光照系数， L_d 表示漫反射光照强度，a、b、c 表示距离衰减参数， \mathbf{l} 表示光线入射方向， \mathbf{n} 表示曲面法向量。

镜面反射光照公式：

$$I_{specular} = \frac{1}{a+bd+cd^2} (k_s L_s \max((\mathbf{l} \cdot \mathbf{n})^\alpha, 0))$$

其中， k_s 表示镜面反射光照系数， L_s 表示镜面反射光照强度，a、b、c 表示距离衰减参数， \mathbf{v} 表示视线方向， \mathbf{r} 表示光线反射方向， α 表示高光系数。

整体光照公式：

$$I = I_{ambient} + I_{diffuse} + I_{specular}$$

3.5.2 算法代码

```
1  vec3 MyGLWidget::GouraudShading(FragmentAttr& vertice)
    {
2
3      vec3 objColor = vec3(0.7, 0.7, 0.7);
4      float dist = vecDistance(lightPosition, vertice.
        pos_mv);
5      float dist_a = 1.0f, dist_b = 0.001f, dist_c =
        0.000001f;
6
7      //环境光照
8      vec3 aLightColor = vec3(1.0, 1.0, 1.0);
9      float amb_k = 0.2f;
10     vec3 amb = amb_k * aLightColor;
11
12     //漫反射光照
13     vec3 dLightColor = vec3(1.0, 1.0, 1.0);
14     float dif_k = 1.0f;
15     vec3 vec_L = vecNormalize(vecSub(lightPosition,
        vertice.pos_mv));
16     vec3 vec_N = vecNormalize(vertice.normal);
17     float dif_alpha = 2.0f;
18     vec3 dif = dif_k * max(pow(vecDot(vec_L, vec_N),
        dif_alpha), 0.0f) * dLightColor;
19
20     //镜面反射光照
21     vec3 sLightColor = vec3(1.0, 1.0, 1.0);
22     float spe_k = 0.8f;
23     vec3 vec_R = vecNormalize(reflect(vec_L, vec_N));
24     vec3 vec_V = vecNormalize(vecSub(camPosition,
        camLookAt));
25     float spe_alpha = 2.0f;
26     vec3 spe = spe_k * max(pow(vecDot(vec_V, vec_R),
        spe_alpha), 0.0f) * sLightColor;
```

```
27  
28 // 计算最终颜色  
29 vec3 color = objColor * ((1 / (dist_a + dist_b *  
    dist + dist_c * dist * dist)) * (dif + spe) +  
    amb);  
30 return color;  
31 }
```

3.6 用 Gouraud 实现内部的着色

3.6.1 算法原理

Gouraud 模型基于上文所说的光照模型，在 edge-walking 中先计算三角形片段三个顶点上的颜色，再通过线性插值填充三角形内部。

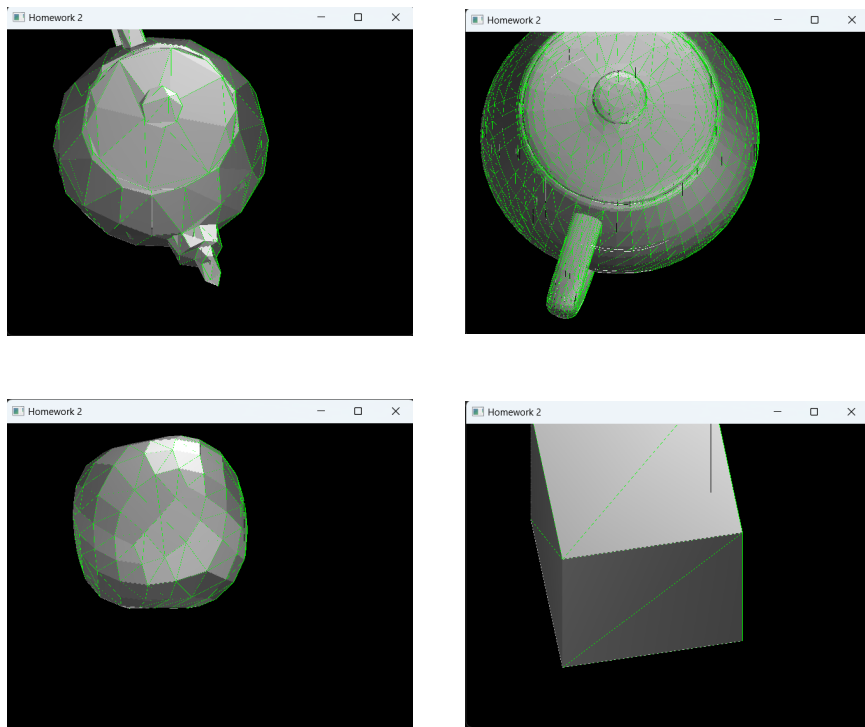
3.6.2 算法代码

光照模型代码已给出，在本实验中 Gouraud 模型的实现主要体现在其相应的 edge-walking 函数中。

```
1 //Gouraud模型的edge_walking方法  
2 int MyGLWidget::Gouraud_edge_walking(FragmentAttr& a,  
    FragmentAttr& b, FragmentAttr& c) {  
3 // 遍历edge_recorder在不同高度的起点、终点，用  
    shading model计算内部每个像素的颜色  
4 a.color = GouraudShading(a);  
5 b.color = GouraudShading(b);  
6 c.color = GouraudShading(c);  
7 for (int i = 0; i < WindowSizeH; i++) {  
8     if (edge_recorder[i][2] >= 2) {  
9         for (int j = edge_recorder[i][0] ; j <  
            edge_recorder[i][1]; j++) {  
10             float cur_z = getLinearInterpolation_z(a, b, c  
                , j, i);  
11             vec3 cur_color = getLinearInterpolation_color  
                (a, b, c, j, i);
```

```
12         temp_render_buffer[WindowSizeW * i + j] =  
            cur_color;  
13         temp_z_buffer[WindowSizeW * i + j] = cur_z;  
14     }  
15 }  
16 }  
17 return min(min(a.y,b.y),c.y);  
18 }
```

3.6.3 实现效果



3.7 用 Phong 实现内部的着色

3.7.1 算法原理

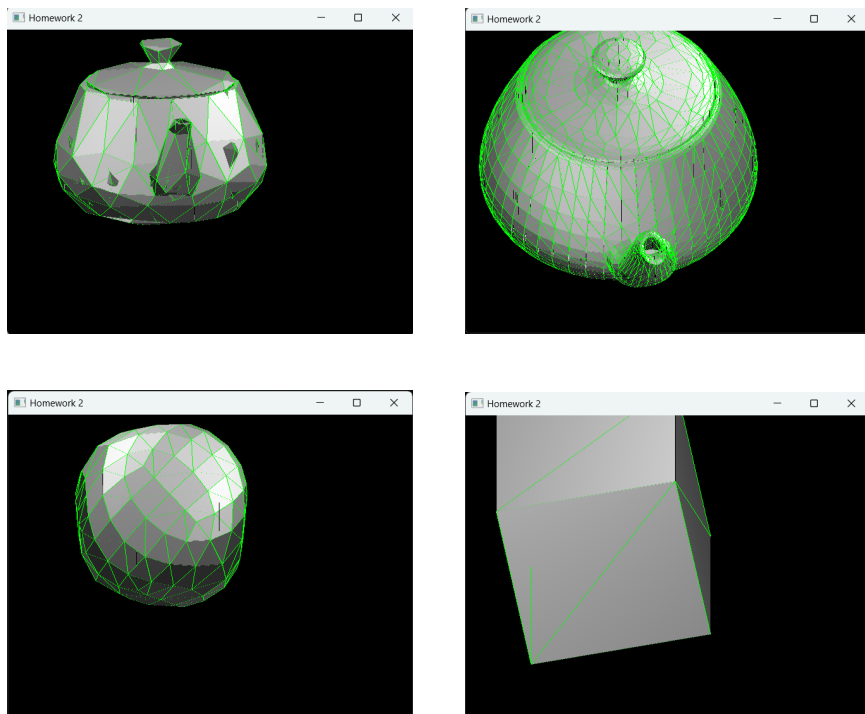
Phong 模型基于上文所说的光照模型，在 edge-walking 中先通过线性插值，由三角形片段三个顶点的法向量和空间位置计算得到三角形内部某一位置的法向量和空间位置，最后利用光照模型计算得到颜色并填充该位置。

3.7.2 算法代码

光照模型代码已给出，在本实验中 Phong 模型的实现主要体现在其相应的 edge-walking 函数中。

```
1 //Phong模型的edge_walking方法
2 int MyGLWidget::Phong_edge_walking(FragmentAttr& a,
   FragmentAttr& b, FragmentAttr& c) {
3     // 遍历edge_recorder在不同高度的起点、终点，用
   shading model计算内部每个像素的颜色
4     for (int i = 0; i < WindowSizeH; i++) {
5         if (edge_recorder[i][2] >= 2) {
6             for (int j = edge_recorder[i][0]+1; j <
   edge_recorder[i][1]; j++) {
7                 vec3 cur_norm = getLinearInterpolation_norm(a,
   b, c, j, i);
8                 vec3 cur_pos = getLinearInterpolation_pos(a, b,
   c, j, i);
9                 float cur_z = getLinearInterpolation_z(a, b, c,
   j, i);
10                vec3 cur_color = PhongShading(cur_norm,
   cur_pos);
11                temp_render_buffer[WindowSizeW * i + j] =
   cur_color;
12                temp_z_buffer[WindowSizeW * i + j] = cur_z;
13            }
14        }
15    }
16    return min(min(a.y, b.y), c.y);
17 }
```

3.7.3 实现效果



3.8 用 Blinn-Phong 实现内部的着色

3.8.1 算法原理

Blinn-Phong 模型的实现流程与 Phong 模型基本相同，不同的地方在于对镜面反射光照的计算，Blinn-Phong 模型中使用半角向量 \mathbf{h} 近似 Phong 模型中的镜面反射。

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{|\mathbf{l} + \mathbf{v}|}$$

$$I_{\text{specular}} = \frac{1}{a + b|\mathbf{d} + c\mathbf{d}^2} (k_s L_s \max((\mathbf{n} \cdot \mathbf{h})^\alpha, 0))$$

3.8.2 算法代码

光照模型代码已给出，在本实验中 Blinn-Phong 模型的实现主要体现在其相应的 edge-walking 函数中。

```
1 //Phong模型的edge_walking方法
2 int MyGLWidget::Blinn_Phong_edge_walking(FragmentAttr&
    a, FragmentAttr& b, FragmentAttr& c) {
```

```

3 // 遍历 edge_recorder 在不同高度的起点、终点, 用
   shading model 计算内部每个像素的颜色
4 for (int i = 0; i < WindowSizeH; i++) {
5     if (edge_recorder[i][2] >= 2) {
6         for (int j = edge_recorder[i][0] + 1; j <
           edge_recorder[i][1]; j++) {
7             vec3 cur_norm = getLinearInterpolation_norm(a,
               b, c, j, i);
8             vec3 cur_pos = getLinearInterpolation_pos(a, b
               , c, j, i);
9             float cur_z = getLinearInterpolation_z(a, b, c
               , j, i);
10            vec3 cur_color = BlinnPhongShading(cur_norm,
               cur_pos);
11            temp_render_buffer[WindowSizeW * i + j] =
               cur_color;
12            temp_z_buffer[WindowSizeW * i + j] = cur_z;
13        }
14    }
15 }
16 return min(min(a.y, b.y), c.y);
17 }

```

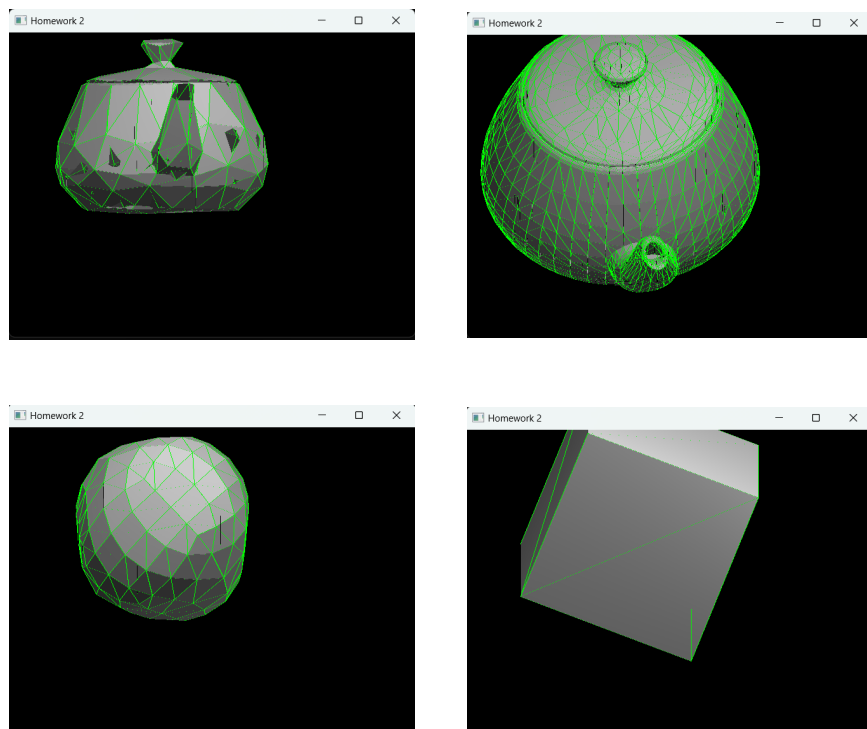
此外, Blinn-Phong 模型对镜面反射光照的计算与 Phong 模型不同:

```

1 // 镜面反射光照
2 vec3 sLightColor = vec3(1.0, 1.0, 1.0);
3 float spe_k = 0.6f;
4 vec3 vec_R = vecNormalize(reflect(vec_L, vec_N));
5 vec3 vec_V = vecNormalize(vecSub(camPosition,
   camLookAt));
6 vec3 vec_H = vecNormalize(vecDiv(vecAdd(vec_R, vec_V),
   2.0));
7 float spe_alpha = 2.0f;
8 vec3 spe = spe_k * max(pow(vecDot(vec_H, vec_N),
   spe_alpha), 0.0f) * sLightColor;

```

3.8.3 实现效果



3.9 Gouraud、Phong 与 Blinn-Phong 的效果及效率

由于 Gouraud 模型对一个三角形片段只需要计算其三个顶点上的颜色值，内部颜色填充通过线性插值完成，而 Phong 模型和 Blinn-Phong 模型需要由线性插值得到的法向量和空间坐标计算每一内部位置的颜色值，因此 Gouraud 模型的计算开销远小于 Phong 模型和 Blinn-Phong 模型。

当然，因为 Phong 模型和 Blinn-Phong 模型通过三角形每一内部位置的法向量和空间坐标计算其相应的颜色值，所以渲染效果上与 Gouraud 模型相比更为平滑和真实。

而对于 Phong 模型和 Blinn-Phong 模型，两者的区别仅在于 Blinn-Phong 模型采用半角向量近似 Phong 模型中的镜面反射，在某些情况下能减少一定计算量。在渲染效果上，Blinn-Phong 模型弥补了 Phong 模型会出现光照截断或者过度不自然的问题，从而显得更为柔和，没有断层现象。

4 其他代码

4.1 edge-recorder

与 edge-recorder 相关的函数在 **myglwidget.h** 的 MyGLWidget 类中已进行声明，并在 **myglwidget.cpp** 进行实现。

清空 edge-recorder:

```
1 void MyGLWidget::clearRecorder(vec3* now_buffer) {
2     for (int i = 0; i < WindowSizeH; i++) {
3         now_buffer[i] = vec3(-1, -1, 0);
4     }
5 }
```

在 DDA 和 bresenham 中使用 updateRecorder 更新 edge-recorder 的信息:

```
1 void MyGLWidget::updateRecorder(int x, int y) {
2     int r0 = edge_recorder[y][0];
3     int r1 = edge_recorder[y][1];
4
5     //若在高度y上没有起点
6     if (r0 == -1) {
7         edge_recorder[y][0] = x;
8     }
9     //若在高度y上已有起点
10    else {
11        //若在高度y上没有终点
12        if (r1 == -1) {
13            edge_recorder[y][1] = max(r0, x);
14            edge_recorder[y][0] = min(r0, x);
15        }
16        //若在高度y上已有终点
17        else {
18            edge_recorder[y][0] = min(min(r0, r1), x);
19            edge_recorder[y][1] = max(max(r0, r1), x);
20        }
21    }
```

```

21     }
22     edge_recorder[y][2] += 1;
23 }

```

4.2 插值工具

与线性计算相关的代码放于 `utils.cpp` 文件中。

深度的二维线性插值：

```

1  float getLinearInterpolation_z(FragmentAttr& a,
    FragmentAttr& b, FragmentAttr& c, int x_position,
    int y_position) {
2  float A, B, C, D;
3  int x1 = a.x, x2 = b.x, x3 = c.x;
4  int y1 = a.y, y2 = b.y, y3 = c.y;
5  float z1 = a.z, z2 = b.z, z3 = c.z;
6
7  A = (y2 - y1) * (z3 - z1) - (z2 - z1) * (y3 - y1);
8  B = (z2 - z1) * (x3 - x1) - (x2 - x1) * (z3 - z1);
9  C = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
10 D = -(A * x1 + B * y1 + C * z1);
11
12 return -(A * x_position + B * y_position + D) / C;
13 }

```

颜色的二维线性插值：

```

1  vec3 getLinearInterpolation_color(FragmentAttr& a,
    FragmentAttr& b, FragmentAttr& c, int x_position,
    int y_position) {
2  float A, B, C, D;
3  int x1 = a.x, x2 = b.x, x3 = c.x;
4  int y1 = a.y, y2 = b.y, y3 = c.y;
5  vec3 result = vec3(0, 0, 0);
6

```

```

7   for (int i = 0; i <= 2; i++) {
8       float z1 = a.color[i], z2 = b.color[i], z3 = c.
          color[i];
9       A = (y2 - y1) * (z3 - z1) - (z2 - z1) * (y3 - y1);
10      B = (z2 - z1) * (x3 - x1) - (x2 - x1) * (z3 - z1);
11      C = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
12      D = -(A * x1 + B * y1 + C * z1);
13
14      result[i] = -(A * x_position + B * y_position + D)
          / C;
15  }
16  return result;
17 }

```

法向量二维线性插值:

```

1  vec3 getLinearInterpolation_norm(FragmentAttr& a,
      FragmentAttr& b, FragmentAttr& c, int x_position,
      int y_position) {
2      float A, B, C, D;
3      int x1 = a.x, x2 = b.x, x3 = c.x;
4      int y1 = a.y, y2 = b.y, y3 = c.y;
5      vec3 result = vec3(0, 0, 0);
6
7      for (int i = 0; i <= 2; i++) {
8          float z1 = a.normal[i], z2 = b.normal[i], z3 = c.
              normal[i];
9          A = (y2 - y1) * (z3 - z1) - (z2 - z1) * (y3 - y1);
10         B = (z2 - z1) * (x3 - x1) - (x2 - x1) * (z3 - z1);
11         C = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
12         D = -(A * x1 + B * y1 + C * z1);
13
14         result[i] = -(A * x_position + B * y_position + D)
            / C;
15     }
16     return result;

```

```
17 }
```

空间坐标二维线性插值:

```
1  vec3 getLinearInterpolation_pos(FragmentAttr& a,
    FragmentAttr& b, FragmentAttr& c, int x_position,
    int y_position) {
2  float A, B, C, D;
3  int x1 = a.x, x2 = b.x, x3 = c.x;
4  int y1 = a.y, y2 = b.y, y3 = c.y;
5  vec3 result = vec3(0, 0, 0);
6
7  for (int i = 0; i <= 2; i++) {
8      float z1 = a.pos_mv[i], z2 = b.pos_mv[i], z3 = c.
        pos_mv[i];
9      A = (y2 - y1) * (z3 - z1) - (z2 - z1) * (y3 - y1);
10     B = (z2 - z1) * (x3 - x1) - (x2 - x1) * (z3 - z1);
11     C = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
12     D = -(A * x1 + B * y1 + C * z1);
13
14     result[i] = -(A * x_position + B * y_position + D)
        / C;
15 }
16 return result;
17 }
```

4.3 向量工具

与向量计算相关的代码放于 `utils.cpp` 文件中。

向量归一化:

```
1  vec3 vecNormalize(vec3 a) {
2      float len = sqrt(a[0] * a[0] + a[1] * a[1] + a[2] *
        a[2]);
3  }
```



```
4   return vec3(a[0] / len, a[1] / len, a[2] / len);  
5 }
```

向量点乘:

```
1 float vecDot(vec3 a, vec3 b) {  
2   return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];  
3 }
```

向量叉乘:

```
1 vec3 vecCross(vec3 a, vec3 b) {  
2   return vec3(a[1] * b[2] - a[2] * b[1], a[2] * b[0] -  
               a[0] * b[2], a[0] * b[1] - a[1] * b[0]);  
3 }
```

向量空间距离:

```
1 float vecDistance(vec3 a, vec3 b) {  
2   return sqrt((a[0] - b[0]) * (a[0] - b[0]) + (a[1] - b[1]) *  
               (a[1] - b[1]) + (a[2] - b[2]) * (a[2] - b[2]));  
3 }
```

向量加:

```
1 vec3 vecAdd(vec3 a, vec3 b) {  
2   return vec3(a[0] + b[0], a[1] + b[1], a[2] + b[2]);  
3 }
```

向量减:

```
1 vec3 vecSub(vec3 a, vec3 b) {  
2   return vec3(a[0] - b[0], a[1] - b[1], a[2] - b[2]);  
3 }
```

向量除以系数:

```
1 vec3 vecDiv(vec3 a, float f) {  
2   return vec3(a[0] / f, a[1] / f, a[2] / f);  
3 }
```