

- 计算机图形学 作业1
 - 1. 实现三角形的光栅化算法
 - 1.1: 用 DDA 实现三角形边的绘制
 - 实现思路
 - 实现过程
 - 1.2: 用 bresenham 实现三角形边的绘制
 - 实现思路
 - 实现过程
 - 1.3: 用 edge-walking 填充三角形内部颜色
 - 实现思路
 - 实现过程
 - 运行结果
 - 1.4讨论:
 - 2. 实现光照、着色
 - 2.1: 用 Gouraud 实现三角形内部的着色
 - 实现思路
 - 实现过程
 - 2.2: 用 Phong 模型实现三角形内部的着色
 - 实现思路
 - 实现过程
 - 2.3: 用 Blinn-Phong 实现三角形内部的着色
 - 实现思路
 - 实现过程
 - 2.4 讨论

计算机图形学 作业1

21312620 熊蔚然

1. 实现三角形的光栅化算法

1.1: 用 DDA 实现三角形边的绘制

实现思路

首先计算直线在 x 和 y 方向上的增量 dx 和 dy ，步数取 dx 和 dy 中绝对值较大者，以确保在增量较大的方向上有足够的步数。再计算 x 和 y 方向上每一步的增量，之后从起点开始，通过逐步增加 x 和 y 的坐标，计算每一步的像素位置，然后在该位置设置像素的颜色。

实现过程

函数接受两个 `FragmentAttr` 类型的参数 `start` 和 `end`，表示直线的起点和终点，之后按照算法原理实现DDA的具体过程即可。最开始时实现的DDA函数如下：

```
void MyGLWidget::DDA(FragmentAttr& start, FragmentAttr& end, int id) {
    int x1 = start.x;
    int y1 = start.y;
    int x2 = end.x;
    int y2 = end.y;

    int dx = x2 - x1, dy = y2 - y1; // 计算 x 和 y 方向的增量
    int steps;
    float delta_x, delta_y, x = x1, y = y1;
    vec3 color(1.0f, 0.0f, 0.0f); // 设置像素颜色为红色

    steps = (abs(dx) > abs(dy)) ? (abs(dx)) : (abs(dy)); // 计算步数，取增量较大
    的方向
    delta_x = dx / (float)steps; // 计算 x 方向每步的增量
    delta_y = dy / (float)steps; // 计算 y 方向每步的增量

    set_pixel(round(x), round(y), color, 0); // 设置起点像素颜色

    for (int i = 0; i < steps; ++i) {
        x += delta_x; // 更新 x 坐标
        y += delta_y; // 更新 y 坐标
        set_pixel(round(x), round(y), color, 0); // 设置当前像素颜色
    }
}
```

1.2: 用 bresenham 实现三角形边的绘制

实现思路

思路与课件中给出的相同。首先要计算直线在 x 和 y 方向上的增量，并确定 x 和 y 的增量的正负。之后根据斜率的绝对值是否大于等于1分两种情况分析，并用课件中给出的公式判断哪些像素需要包含在直线中。

实现过程

函数接受的参数与DDA相同，最初的bresenham实现如下：

```
// 使用Bresenham算法在屏幕上绘制直线
void MyGLWidget::bresenham(FragmentAttr& start, FragmentAttr& end, int id) {
    // 提取起点和终点坐标
    int x1 = start.x;
    int y1 = start.y;
    int x2 = end.x;
    int y2 = end.y;

    // 初始化当前像素的坐标
    int x = x1, y = y1;

    // 计算 x 和 y 方向上的增量
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);

    // 判断斜率的变化是否大于等于1
    int k = (dx >= dy) ? 1 : 0;

    // 计算 x 和 y 方向的增量方向
    int sx = (x2 > x1) ? 1 : -1;
    int sy = (y2 > y1) ? 1 : -1;

    // 初始化 Bresenham 算法中的判别式参数 p
    int p = (k) ? (2 * dy - dx) : (2 * dx - dy);

    // 设置像素颜色为蓝色
    vec3 color(0.0f, 0.0f, 1.0f);

    // 设置起点像素颜色
    set_pixel(x, y, color, 0);

    // 根据斜率情况循环绘制直线
    if (k) {
        while (x != x2 || y != y2) {
            // 根据判别式的值的正负选择不同的更新规则
            if (p <= 0) {
                x += sx;
                set_pixel(x, y, color, 0);
                p = p + 2 * dy;
            } else {
                x += sx;
                y += sy;
                set_pixel(x, y, color, 0);
                p = p + 2 * (dy - dx);
            }
        }
    } else {
        while (x != x2 || y != y2) {
            // 根据判别式的值的正负选择不同的更新规则
            if (p <= 0) {
                y += sy;
                set_pixel(x, y, color, 0);
                p = p + 2 * dx;
            } else {
                y += sy;
                x += sx;
                set_pixel(x, y, color, 0);
                p = p + 2 * (dx - dy);
            }
        }
    }
}
```

```

        x += sx;
        y += sy;
        set_pixel(x, y, color, 0);
        p = p + 2 * (dx - dy);
    }
}
}

```

1.3: 用 edge-walking 填充三角形内部颜色

实现思路

由于每次只画一个三角形，所以可以采用遍历整个屏幕的方法。从屏幕底部向上遍历每一行像素，在每行中从左到右遍历每个像素，查找不是黑色的像素。在找到颜色变化的像素后，通过非黑色的像素间是否连在一起来确定左边界和右边界，并将边界范围内的像素着色。返回结果：返回第一个发生颜色变化的水平线位置。

实现过程

由于是遍历整个屏幕，函数没有传入参数。具体实现时除了按实现思路的步骤，还需要额外处理一下一行只有一个点在三角形范围内的情况。函数返回值为第一个发生颜色变化的水平线位置，用于在后续渲染时能少遍历一些空白的行。

```

int MyGLWidget::edge_walking() {
    // 初始化第一个颜色变化的水平线位置为窗口高度
    int firstChangeLine = WindowSizeH;

    vec3 black(0.0f, 0.0f, 0.0f);
    vec3 grey(0.5f, 0.5f, 0.5f);

    // 从底部向上遍历每一行
    for (int i = WindowSizeH - 1; i >= 0; i--) {
        int left_boarder = -1;
        int right_boarder = -1;

        // 遍历当前行的每个像素
        for (int x = 0; x <= WindowSizeW; x++) {
            // 如果当前像素不是黑色
            if (temp_render_buffer[i * WindowSizeW + x] != black) {
                firstChangeLine = i;

                // 如果左边界已经记录，说明当前这个非黑色点对应的是右边
                if (left_boarder != -1) {
                    right_boarder = x;
                    break;
                }
                // 记录右边界并结束循环
            }
        }
    }
}

```

```

        }

        // 如果左边界未记录, 记录左边界
        if (temp_render_buffer[i * WindowSizeW + x + 1] ==
black) {

            left_boarder = x ;

        }

    }

    // 如果左边界已记录而右边界未记录, 则表示这一行只有一个三角形顶点, 不需要
    用edge_walking上色, 所以将left_boarder设为-1
    if (left_boarder != -1 && right_boarder == -1) {
        //right_boarder = left_boarder;
        left_boarder = -1;
    }

    // 如果左边界已记录, 则将该行在三角形范围内的像素设置为灰色
    if (left_boarder != -1) {
        for (int x = left_boarder; x <= right_boarder; x++) {
            temp_render_buffer[i * WindowSizeW + x] = grey;
        }
    }

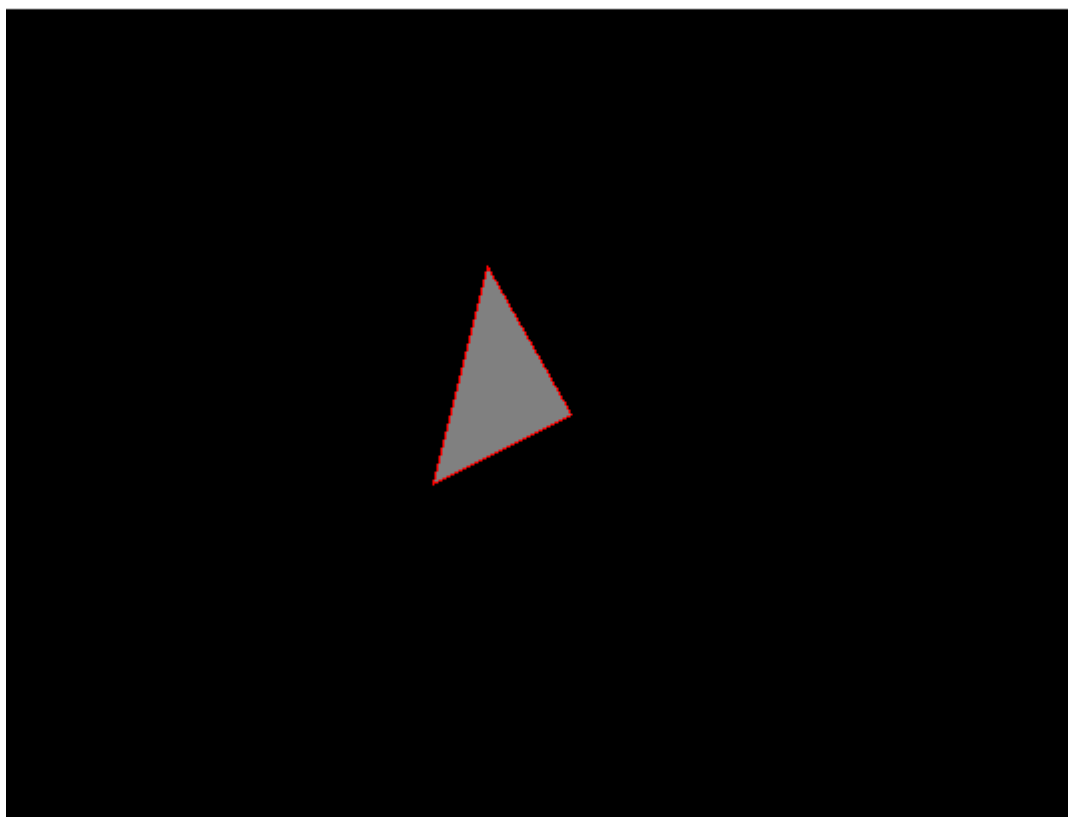
    }

    // 返回第一个发生颜色变化的水平线位置
    return firstChangeLine;
}

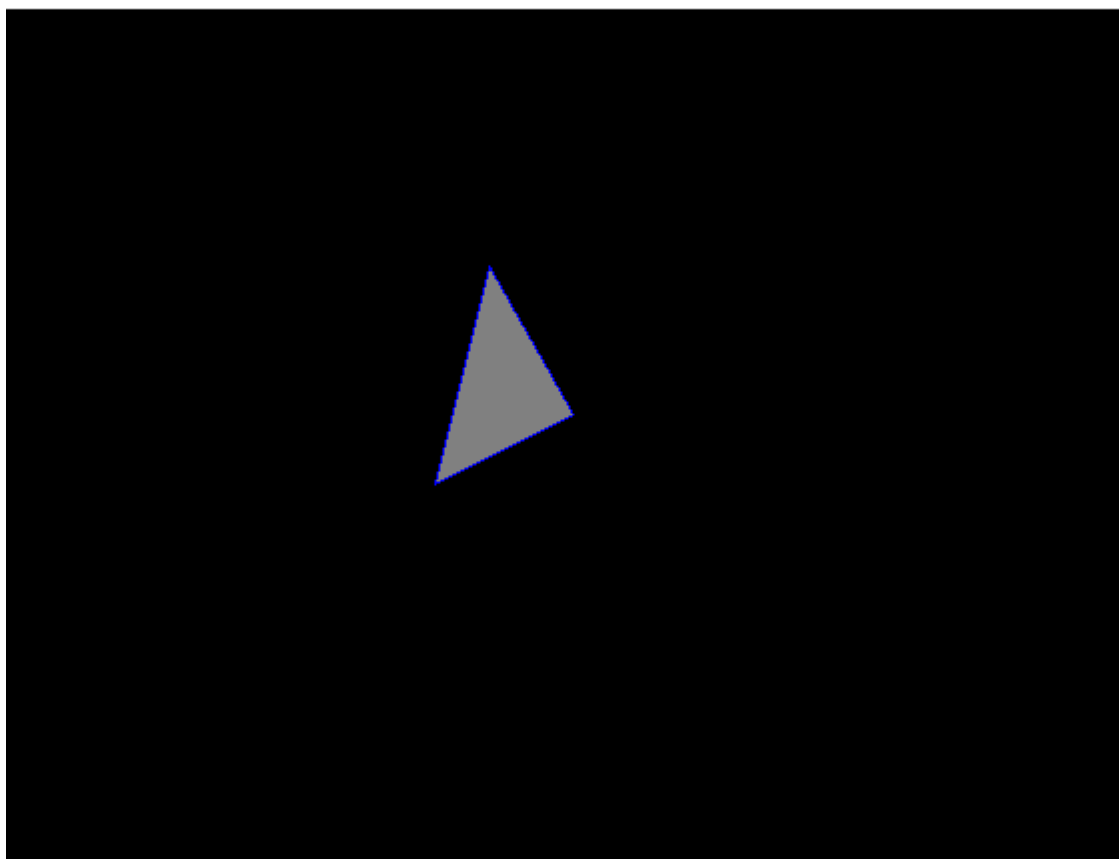
```

运行结果

加载单一三角形的情况下, DDA-EdgeWalking的运行结果如下:



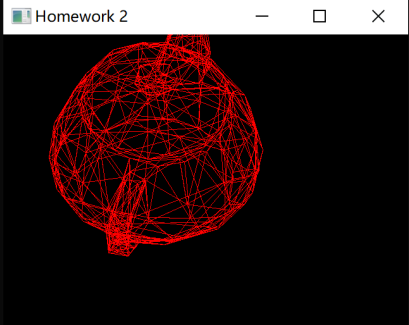
Bresenham-EdgeWalking的运行结果如下：



在scene_1中加入记录运行时间的代码。

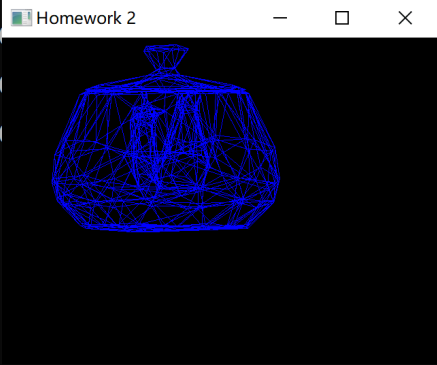
绘制teapot_600时，使用DDA的运行时间为3.3s左右：

```
E:\CGTemplate_HW2\debug\CGTemplate.exe
read 1 triangles
read 612 triangles
Time taken by scene_1: 3269 milliseconds
Time taken by scene_1: 3362 milliseconds
Time taken by scene_1: 3319 milliseconds
Time taken by scene_1: 3318 milliseconds
Time taken by scene_1: 3545 milliseconds
```

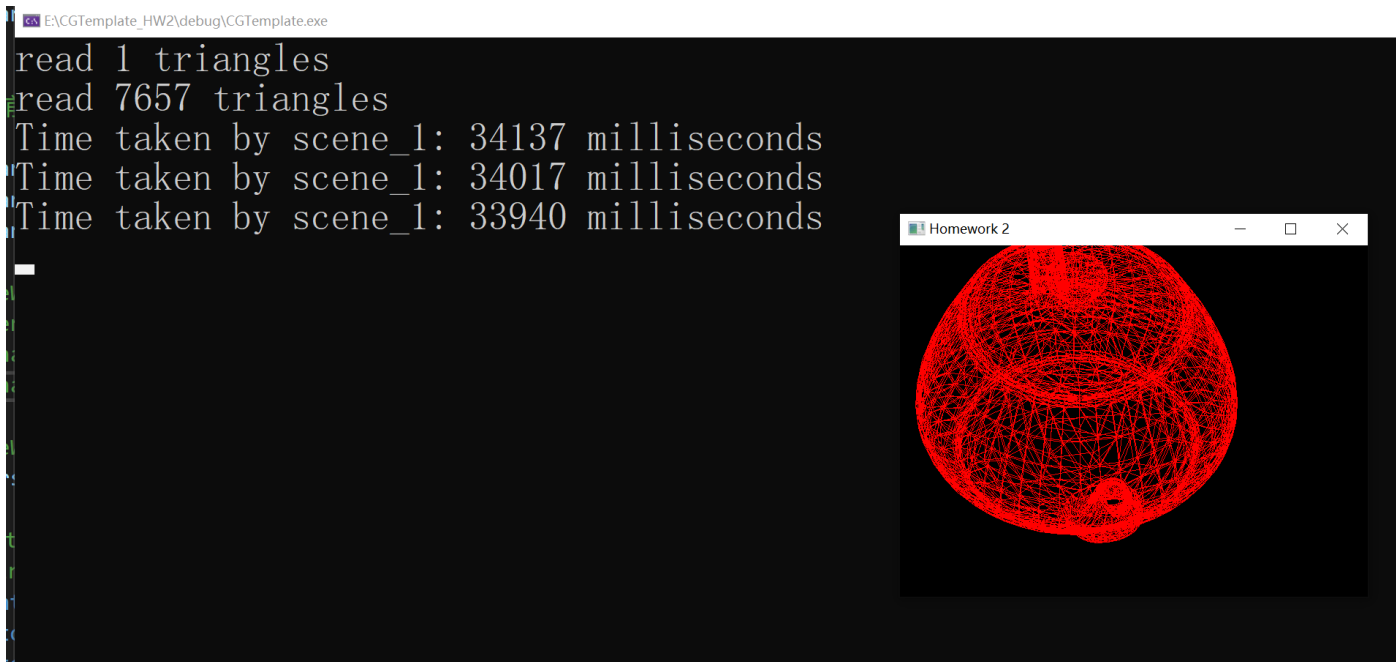


而使用Bresenham的运行时间也非常接近：

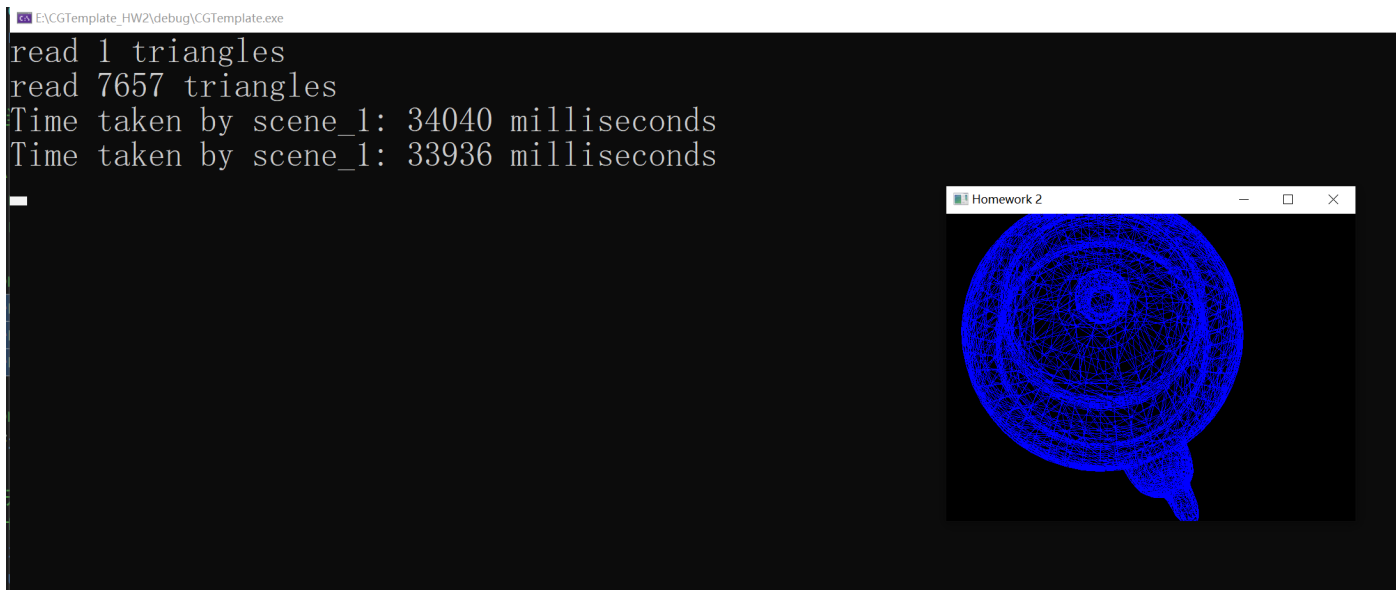
```
E:\CGTemplate_HW2\debug\CGTemplate.exe
read 1 triangles
read 612 triangles
Time taken by scene_1: 3314 milliseconds
Time taken by scene_1: 3963 milliseconds
Time taken by scene_1: 3373 milliseconds
Time taken by scene_1: 3284 milliseconds
Time taken by scene_1: 3234 milliseconds
Time taken by scene_1: 3220 milliseconds
```



换成8000面的模型后，DDA的运行时间为34s左右：



Bresenham的运行时间也同样在34s左右:



2. 实现光照、 着色

2.1: 用 Gouraud 实现三角形内部的着色

实现思路

对于每个三角形，首先通过Phone模型用光照等信息计算其三个顶点的颜色。之后在使用bresenham根据顶点绘制三角形的边时，利用线性插值得到每条边上每个点的颜色值。绘制好三边之后，在edge_walking中根据每横行中三角形的两个点，对该行中三角形内部的每个像素进行线性插值，计算其颜色、深度等信息，最后调用set_pixel函数在对应buffer中更新信息。

实现过程

首先要实现用**Phone**光照模型计算顶点颜色的函数。这需要根据课件上的公式计算一系列变量，并通过算出的各光照分量合成最终颜色。

函数具体实现如下：

```
vec3 MyGLWidget::phong_model(const vec3& normal, const vec3& pos_mv, const vec3&
original_color, float ka, float kd, float ks)
{
    // 定义光源颜色
    vec3 light_source(1.0f, 1.0f, 1.0f);

    // 对法向量进行标准化
    vec3 normal_normalized = normalize(normal);

    // 计算视线方向
    vec3 view_direction = normalize(camPosition - pos_mv);

    // 计算光线方向
    vec3 light_direction = normalize(lightPosition - pos_mv);

    // 计算环境光照
    vec3 ambient = ka * light_source * original_color;

    // 计算漫反射光照
    vec3 diffuse = kd * max(dot(normal_normalized, light_direction), 0.0f) *
light_source * original_color;

    // 计算镜面反射光照
    vec3 reflect_direction = reflect(-light_direction, normal_normalized);
    float spec = pow(max(dot(view_direction, reflect_direction), 0.0f), 32);
    vec3 specular = ks * spec * light_source;

    // 合成最终颜色并限制在合理范围
    vec3 phong_model_color = ambient + diffuse + specular;
    return clamp(phong_model_color, 0.0f, 1.0f);
}
```

为使用这一函数，需要在draw_triangle函数中设定三个光照系数的值，并在给transformedVertices[i]赋值时调用phong_model函数，将计算出的颜色赋给三角形顶点。

```

void MyGLWidget::drawTriangle(Triangle triangle) {
    // 三维顶点映射到二维平面
    vec3* vertices = triangle.triangleVertices;
    vec3* normals = triangle.triangleNormals;
    FragmentAttr transformedVertices[3];
    float ka, kd, ks;
    ka = 0.1;
    kd = 0.5;
    ks = 0.5;
    clearBuffer(this->temp_render_buffer);
    clearZBuffer(this->temp_z_buffer);
    /*clearBuffer(this->normal_buffer);
    clearBuffer(this->pos_mv_buffer);*/
    mat4 viewMatrix = glm::lookAt(camPosition, camLookAt, camUp);

    for (int i = 0; i < 3; ++i) {
        vec4 ver_mv = viewMatrix * vec4(vertices[i], 1.0f);
        float nowz = glm::length(camPosition - vec3(ver_mv));
        vec4 ver_proj = projMatrix * ver_mv;
        transformedVertices[i].x = ver_proj.x + offset.x;
        transformedVertices[i].y = ver_proj.y + offset.y;
        transformedVertices[i].z = nowz;
        transformedVertices[i].pos_mv = ver_mv;
        mat3 normalMatrix = mat3(viewMatrix);
        vec3 normal_mv = normalMatrix * normals[i];
        transformedVertices[i].normal = normal_mv;

        transformedVertices[i].color = phong_model(transformedVertices[i].normal, transformedVertices[i].pos_mv, transformedVertices[i].color, ka, kd, ks);
    }
}

```

顶点的初始颜色在utils.h中设定：

```

struct FragmentAttr {
    int x;
    int y;
    float z;
    int edgeID;
    vec3 color = vec3(1.0f, 1.0f, 0.0f);
    vec3 normal;
    vec3 pos_mv;
    FragmentAttr(){}
    FragmentAttr(int xx, int yy, float zz, int in_edgeID) :x(xx), y(yy), z(zz), edgeID(in_edgeID) {
    }
};

```

之后修改bresenham函数，执行set_pixel前先执行线性插值计算当前点的颜色。

```

if (p <= 0) {
    x += sx;
    //t = fabs(float(x-x1)/(x1-x2));
    interpolate = getLinearInterpolation(start, end, x,y);
    /*if (interpolate.color == black)
    {
        std::cout << "black" << endl;
    }*/
    //normal_buffer[interpolate.y * WindowSizeW + x] = interpolate.normal;
    //pos_mv_buffer[interpolate.y * WindowSizeW + x] = interpolate.pos_mv;
    set_pixel(x, y, interpolate.color, interpolate.z, interpolate.normal, interpolate.pos_mv);
    //if (std::isnan(interpolate.color.r)) {
    //    //std::cout << "left:" << temp_z_buffer[i * WindowSizeW + left_boarder-1] << std::endl;
    //    for (int j = 0; j < WindowSizeW; j++)
    //    {
    //        std::cout << j << temp_z_buffer[y * WindowSizeW + j] << std::endl;
    //    }
    //    std::cout << "right:" << temp_z_buffer[y * WindowSizeW + x] << std::endl;
    //}
    p = p + 2 * dy;
}

```

这里的set_pixel函数已经进行了修改，首先判断传入的点是否在显示窗口范围内，若在窗口内再根据传入的颜色、深度、法线、model_view坐标系中的位置，对对应的buffer进行设置。这里用到了两个新的buffer，是由于后续需要实现的算法中需要用到三角形边上的点的法线和pos_mv信息，所以用两个新的buffer来存储这些结果。这两个buffer的定义、创建、清空等相关操作在程序中对应的部分也已添加。

修改后的set_pixel函数如下：

```
void MyGLWidget::set_pixel(int x, int y, const vec3& color, float depth, vec3
normal, vec3 pos_mv) {
    // 检查像素是否在窗口范围内
    if (x < 0 || x >= WindowSizeW || y < 0 || y >= WindowSizeH) {
        // 像素位于窗口之外，因此不执行任何操作

    }
    if (x >= 0 && x < WindowSizeW && y >= 0 && y < WindowSizeH) {
        // 计算像素在帧缓冲数组中的线性索引
        int index = y * WindowSizeW + x;
        // 在缓冲中设置像素的颜色、深度、法线、model_view中位置
        temp_render_buffer[index] = color;
        temp_z_buffer[index] = depth;
        normal_buffer[index] = normal;
        pos_mv_buffer[index] = pos_mv;
    }
}
```

我在实现这一部分时还遇到了一个小问题，原本我是在调用了set_pixel函数的地方直接用两行语句加上对normal_buffer和pos_mv_buffer对应位置的赋值，但这样会导致加载的模型中如果有部分点在窗口之外，就会在赋值时出现normal_buffer和pos_mv_buffer的index超出数组范围的错误，程序无法正确运行。所以我将这两个buffer的赋值也一同放到set_pixel函数中，这样就可以先进行判断，跳过在窗口外的点。

然后在edge_walking中，遍历时需要获取每横行中在三角形边上的两个点的信息，并对该行中三角形内部的每个像素进行线性插值，从而实现用gouraud对三角形内部进行着色。edge_walking函数的核心修改如下：

```
if (left_boarder != -1) {
    //获取该横行上在三角形上的两个点的信息
    temp1.x = left_boarder;
    temp1.y = i;
    temp1.color = temp_render_buffer[i * WindowSizeW + left_boarder];
    temp1.z = temp_z_buffer[i * WindowSizeW + left_boarder];
    temp1.normal = normal_buffer[i * WindowSizeW + left_boarder];
    temp1.pos_mv = pos_mv_buffer[i * WindowSizeW + left_boarder];
    temp2.x = right_boarder;
```

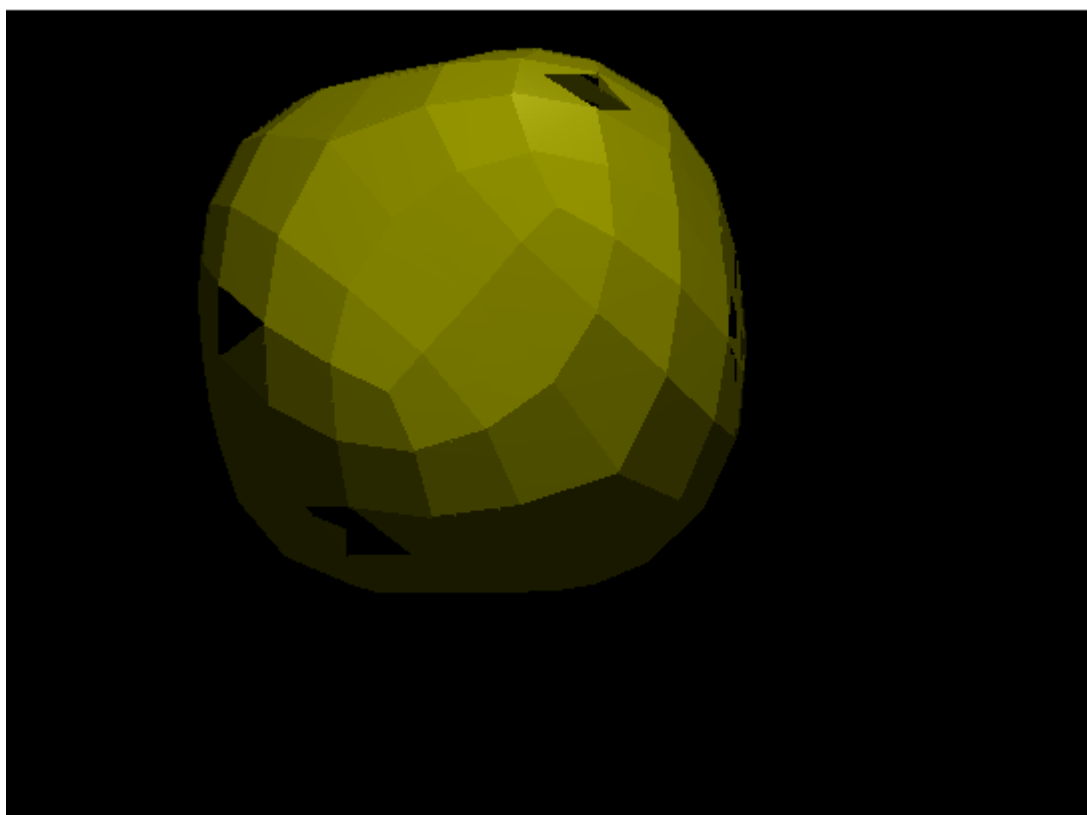
```

temp2.y = i;
temp2.color = temp_render_buffer[i * WindowSizeW + right_boarder];
temp2.z = temp_z_buffer[i * WindowSizeW + right_boarder];
temp2.normal = normal_buffer[i * WindowSizeW + right_boarder];
temp2.pos_mv = pos_mv_buffer[i * WindowSizeW + right_boarder];
for (int x = left_boarder; x <= right_boarder; x++)
{
    interpolate = getLinearInterpolation(temp1, temp2, x,i);
    set_pixel(x, i, interpolate.color, interpolate.z,
interpolate.normal, interpolate.pos_mv);
}
}

```

在调试这一部分时，我一开始得到的运行结果中有些三角形显示出来仍然是黑色的，类似下面这种效果：

Homework 2



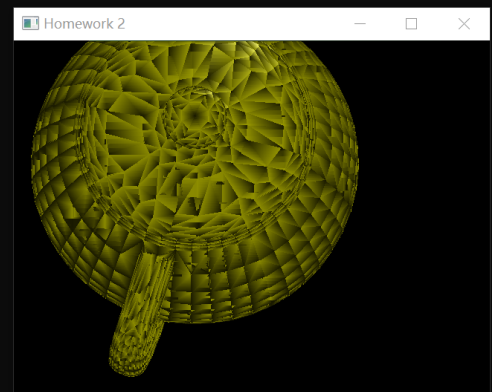
后来我通过输出edge_walking过程中的一些颜色值发现有些地方的颜色的rgb都为nan，进行了进一步的调试后，我发现这些颜色为nan的点都是通过两个x值相等的点线性插值得到的，于是我想到去检查getLinearInterpolation函数中的代码。在原始的代码中并没有对两个点的x值相等的情况进行特殊处理，所以计算t时如果a.x与b.x相等，会出现除以0的错误，导致通过t计算得到的color,z等插值点的属性都变为nan。于是我将插值的代码修改成传入x_position和y_position两个参数，当传入的两个点连成的直线斜率比1小时，就使用横坐标来计算相应的t；当传入的两个点连成的直线斜率比1大时，就通过纵坐标计算，这样就可以避免出现除以0的情况，也可以使插值得到的视觉效果更好。

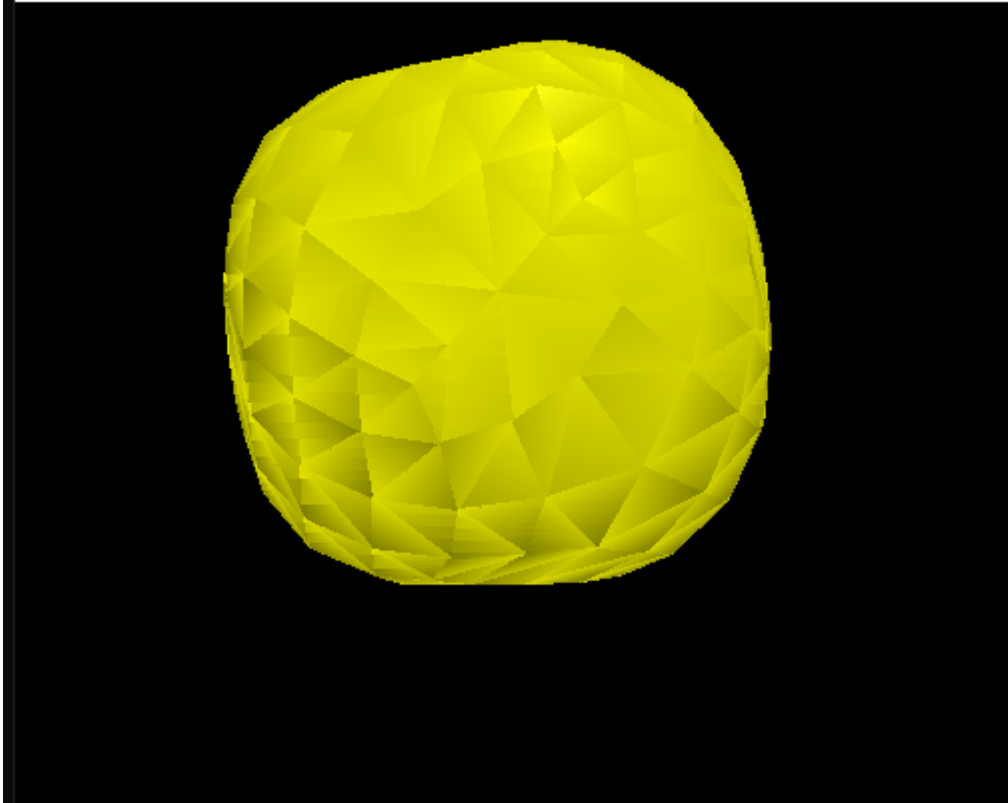
```
if (abs(b.x - a.x) < abs(b.y - a.y))
{
    t = (y_position - a.y) / float(b.y - a.y);
}
else
{
    t = (x_position - a.x) / float(b.x - a.x);
}
```

解决了以上问题后，我发现得到的结果的边界非常突兀，看起来很不真实，类似下面这样：

E:\CGTemplate_HW2\debug\CGTemplate.exe

```
read 1 triangles
read 7657 triangles
Time taken by scene_1: 44053 milliseconds
```





我尝试了很久，还是没有找出导致这个错误的原因。后来请教了同样出现这种问题的同学后，发现要把utils.h中的struct Triangle以及getTriangleByID函数的实现修改一下，将其中的指针改为数组，就解决了这个问题。

```
struct Triangle {  
    /*vec3* triangleVertices;  
    vec3* triangleNormals;*/  
    vec3 triangleVertices[3];  
    vec3 triangleNormals[3];  
};
```

```

Triangle getTriangleByID(int id) {
    assert(id < triangleCount);
    int* nowTirVerIDs = triangles[id];
    int* nowTriNormIDs = triangle_normals[id];
    /*vec3 nowTriangleVertices[3];
    vec3 nowTriNorms[3];
    nowTriangleVertices[0] = vertices_data[nowTirVerIDs[0]];
    nowTriangleVertices[1] = vertices_data[nowTirVerIDs[1]];
    nowTriangleVertices[2] = vertices_data[nowTirVerIDs[2]];
    nowTriNorms[0] = normals_data[nowTriNormIDs[0]];
    nowTriNorms[1] = normals_data[nowTriNormIDs[1]];
    nowTriNorms[2] = normals_data[nowTriNormIDs[2]];*/

    Triangle nowTriangle;
    nowTriangle.triangleVertices[0] = vertices_data[nowTirVerIDs[0]];
    nowTriangle.triangleVertices[1] = vertices_data[nowTirVerIDs[1]];
    nowTriangle.triangleVertices[2] = vertices_data[nowTirVerIDs[2]];
    nowTriangle.triangleNormals[0] = normals_data[nowTriNormIDs[0]];
    nowTriangle.triangleNormals[1] = normals_data[nowTriNormIDs[1]];
    nowTriangle.triangleNormals[2] = normals_data[nowTriNormIDs[2]];
    return nowTriangle;
}

```

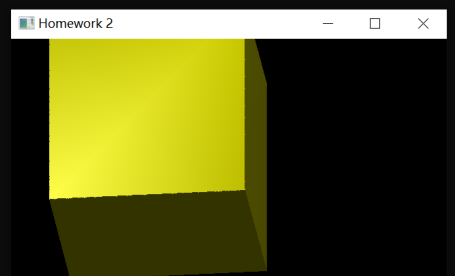
导致这个错误的具体原因我还没有很理解，大概是因为函数中把局部变量转换成指针传出去了，导致后面用到的法线异常，就无法正确计算光照等因素。

最终的运行结果如下：

```

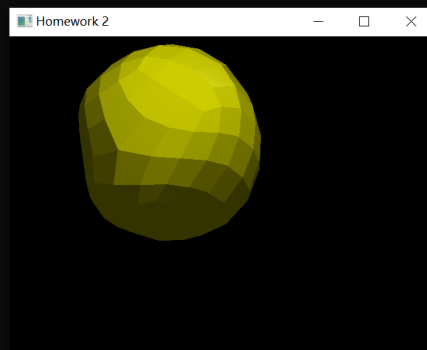
E:\CGTemplate_HW2\debug\CGTemplate.exe
read 1 triangles
read 12 triangles
Time taken by scene_1: 82 milliseconds
Time taken by scene_1: 100 milliseconds
Time taken by scene_1: 92 milliseconds
Time taken by scene_1: 79 milliseconds
Time taken by scene_1: 84 milliseconds
Time taken by scene_1: 87 milliseconds
Time taken by scene_1: 82 milliseconds
Time taken by scene_1: 84 milliseconds
Time taken by scene_1: 90 milliseconds
Time taken by scene_1: 85 milliseconds

```



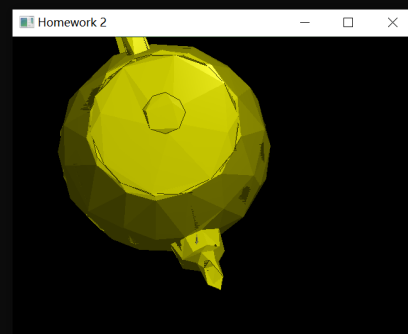
E:\CGTemplate_HW2\debug\CGTemplate.exe

```
read 1 triangles
read 288 triangles
Time taken by scene_1: 1749 milliseconds
Time taken by scene_1: 1731 milliseconds
```



E:\CGTemplate_HW2\debug\CGTemplate.exe

```
read 1 triangles
read 612 triangles
Time taken by scene_1: 3710 milliseconds
Time taken by scene_1: 3641 milliseconds
```



2.2: 用 Phong 模型实现三角形内部的着色

实现思路

使用Phong光照模型中的公式分别计算环境光、漫反射和镜面反射三个分量。环境光是场景中的全局光照，对每个像素都有一个固定的影响。计算方式是用环境光系数 k_a 乘以光的颜色和当前像素的颜色。漫反射用于模拟光线直接照射到表面并散射的情况。计算方式是用漫反射系数 k_d 乘以光线方向和法线方向的点积，再乘以光的颜色和当前像素的

颜色。镜面反射模拟了光线从表面反射回观察者的情况。计算方式是用观察方向和反射方向的点积的幂乘以光的颜色，这里通常使用一个较高的指数来获得较小而明亮的高光。最后，将以上三个分量相加得到最终的颜色。

实现过程

传入当前需要上色的像素点，按各分量的公式计算需要的变量值即可。相加得到最终颜色后需要调用`clamp`函数保证颜色的每个分量值在0-1之间。

```
vec3 MyGLWidget::PhoneShading(FragmentAttr& nowPixelResult) {
    // 定义光源颜色
    vec3 light(1.0f, 1.0f, 1.0f);

    // 定义环境光、镜面反射的系数和幂指数
    float ka, s, alpha;

    // 计算观察方向
    vec3 view_direction = normalize(camPosition - nowPixelResult.pos_mv);

    // 计算光线方向
    vec3 light_direction = normalize(lightPosition - nowPixelResult.pos_mv);

    // 计算法线的规范化向量
    vec3 normal_normalized = normalize(nowPixelResult.normal);

    // 计算反射方向
    vec3 reflect_direction = reflect(-light_direction, normal_normalized);

    // 设置环境光系数和镜面反射的幂指数
    ka = 0.2;
    alpha = 100;

    // 计算环境光分量
    vec3 ambient = ka * light * nowPixelResult.color;

    // 计算漫反射分量
    vec3 diffuse = max(dot(normal_normalized, light_direction), 0.0f) * light *
nowPixelResult.color;

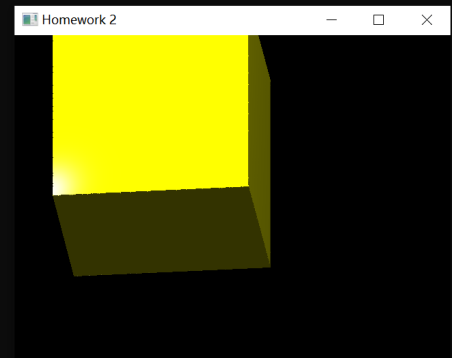
    // 计算镜面反射分量
    s = pow(max(dot(view_direction, reflect_direction), 0.0f), alpha);
    vec3 specular = s * light; // 计算镜面反射的颜色分量

    // 最终颜色是环境光、漫反射和镜面反射三者相加
    vec3 finalColor = ambient + diffuse + specular;

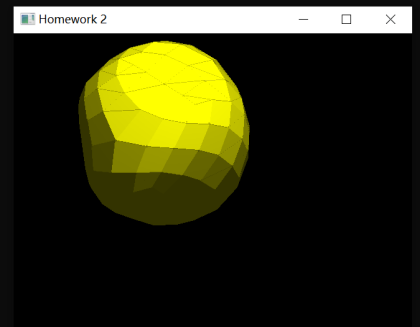
    // 限制颜色范围在 [0, 1] 内
    return clamp(finalColor, 0.0f, 1.0f);
}
```

运行结果：

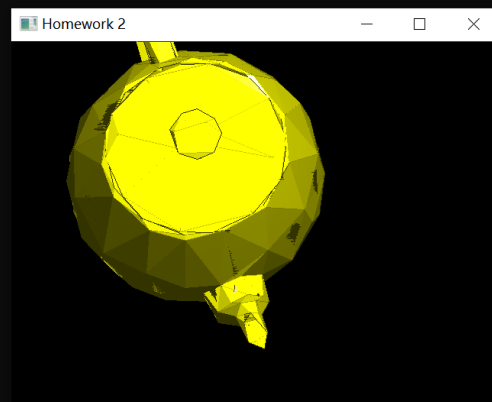
```
read 1 triangles
read 12 triangles
Time taken by scene_1: 133 milliseconds
Time taken by scene_1: 143 milliseconds
Time taken by scene_1: 144 milliseconds
Time taken by scene_1: 129 milliseconds
Time taken by scene_1: 138 milliseconds
Time taken by scene_1: 131 milliseconds
Time taken by scene_1: 138 milliseconds
Time taken by scene_1: 135 milliseconds
```



```
read 1 triangles
read 288 triangles
Time taken by scene_1: 1766 milliseconds
Time taken by scene_1: 1758 milliseconds
```



```
read 1 triangles
read 612 triangles
Time taken by scene_1: 3584 milliseconds
Time taken by scene_1: 3784 milliseconds
```



2.3: 用 Blinn-Phong 实现三角形内部的着色

实现思路

与Phong模型非常相似，只是计算镜面反射分量时使用的是半角向量而不是反射向量，其他计算步骤相同。

实现过程

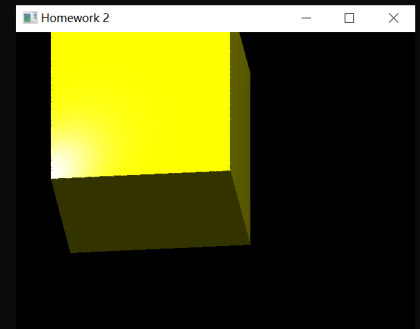
同样与Phong模型非常相似，将计算反射方向的部分改为计算半角向量即可。

```
vec3 h = normalize(light_direction + view_direction);
s = pow(max(dot(normal_normalized, h), 0.0f), alpha);
```

运行结果：

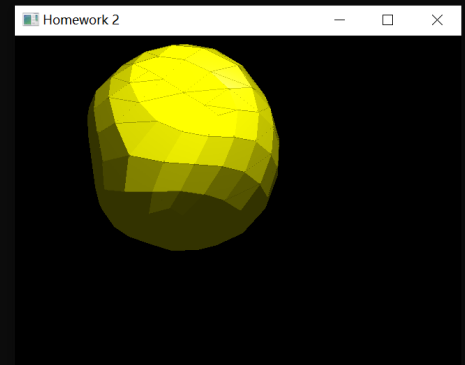
E:\CGTemplate_HW2\debug\CGTemplate.exe

```
read 1 triangles
read 12 triangles
Time taken by scene_1: 136 milliseconds
Time taken by scene_1: 151 milliseconds
Time taken by scene_1: 144 milliseconds
Time taken by scene_1: 129 milliseconds
Time taken by scene_1: 140 milliseconds
Time taken by scene_1: 128 milliseconds
Time taken by scene_1: 143 milliseconds
Time taken by scene_1: 128 milliseconds
```

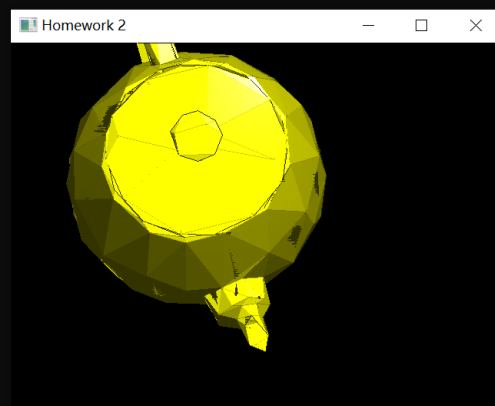


E:\CGTemplate_HW2\debug\CGTemplate.exe

```
read 1 triangles
read 288 triangles
Time taken by scene_1: 1752 milliseconds
Time taken by scene_1: 1779 milliseconds
```



```
read 1 triangles  
read 612 triangles  
Time taken by scene_1: 3602 milliseconds  
Time taken by scene_1: 3574 milliseconds
```



2.4 讨论

由运行结果可看到三种着色方法在各模型下的运行时间都比较相近，着色效率差别不大。**gouraud**的着色效果相对差一些，不能很好的反映出光照对物体颜色的影响。**phong**和**Blinn-Phong**模型的着色效果相近，都能较好地体现出光照的作用，在实验的运行结果中看不出明显的区别。