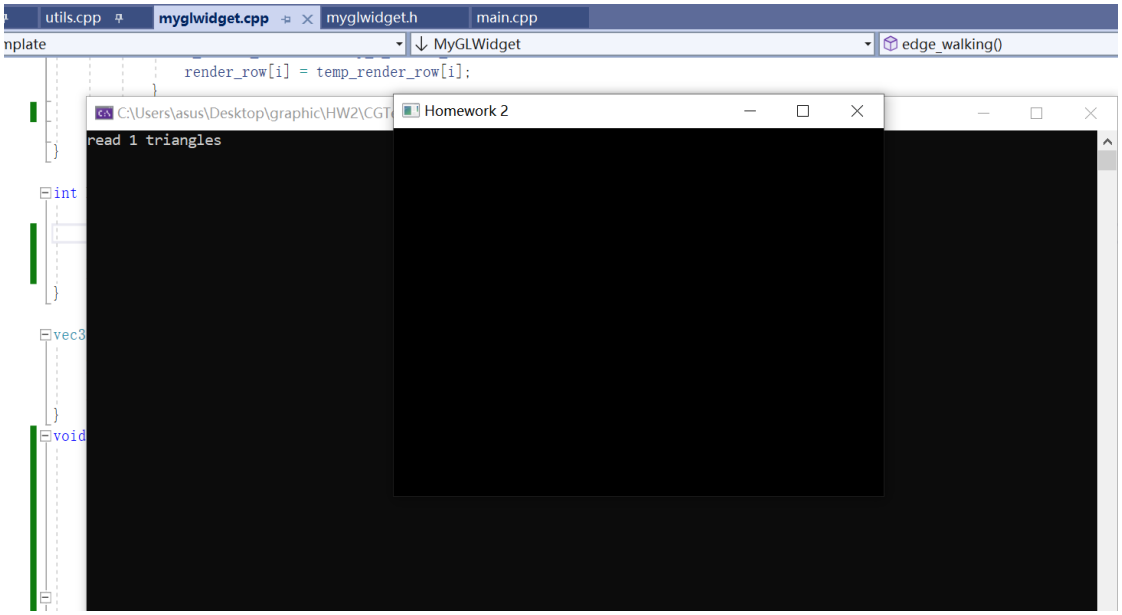
# HW2-Report

21307347 陈欣宇

## 一、环境搭建

依照上一次实验配置环境搭建项目，将 CGTemplate.pro 中路径修改为本电脑对应路径后 qmake 生成 vcxproj 文件即可。编译运行成功如下：




## 二、作业内容

### 1．实现三角形的光栅化算法

**实验要求：**

### 1.1：用 DDA 实现三角形边的绘制

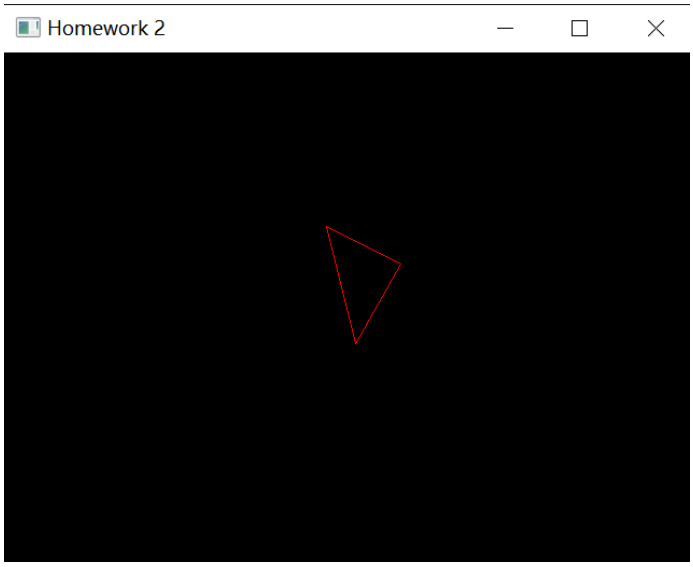
实现 DDA 函数：计算好Δx 和Δy，遍历直线上四舍五入的点 intx、inty 并为 temp_render_buffer 和 temp_z_buffer 进行赋值，其中定义了 FragmentAttr px ，通过 getLinearInterpolation 计算出该点数据，这里主要用于 z 赋值，在后续 edgewalking 中用于记录边的信息。

```
void MyGLWidget::DDA(FragmentAttr& start, FragmentAttr& end, int id,map<int,map<int, FragmentAttr>>& line_l,map<int,map<int, FragmentAttr>>& line_r) {
    int dx = end.x - start.x, dy = end.y - start.y, steps;
    double delta_x, delta_y, x = start.x, y = start.y;
    steps = (abs(dx) > abs(dy)) ? (abs(dx)) : (abs(dy));
    delta_x = dx / (float)steps;delta_y = dy / (float)steps;
    std::map<int, FragmentAttr> lside, rside;
    for (int i = 0; i <= steps; i++) {
        int intx = round(x), inty = round(y);
        FragmentAttr px = getLinearInterpolation(start, end, intx, inty);
        if (abs(delta_y) == 1) {lside[inty] = px;rside[inty] = px;}//对rside、lside进行赋值
        else{ ... }
        if (inty <= WindowSizeH && inty > 0 && intx < WindowSizeW && intx > 0) {//在窗口访问内进行temp_render_buffer和temp_z_buffer赋值
            temp_render_buffer[(WindowSizeH - inty) * WindowSizeW + intx] = vec3(1, 0, 0);
            temp_z_buffer[(WindowSizeH - inty) * WindowSizeW + intx] = px.z;
        }
        x += delta_x;
        y += delta_y;
    }
    line_l[id] = lside;
    line_r[id] = rside;
}
```

调用 DDA 绘制三角形，其中 line_l，lin_r 之后作出解释

```
map<int, map<int, FragmentAttr>> line_l;
map<int, map<int, FragmentAttr>> line_r;
DDA(transformedVertices[0], transformedVertices[1], 2, line_l, line_r);
DDA(transformedVertices[1], transformedVertices[2], 0, line_l, line_r);
DDA(transformedVertices[2], transformedVertices[0], 1, line_l, line_r);
```

运行结果如下：




**1.2：用 bresenham 实现三角形边的绘制**

实现 bresenham 函数：考虑线段角度较缓(dx>=dy)和较陡(dx<dy)的情况，对于较缓情况，计算Δx=1，Δy>0，Iy 表示 y 是递增还是递减，利用Δx、Δy 迭代计算 p，p<=0 则 y 不变，p>0 则 y+=Iy。遍历过程同样对窗口内 temp_render_buffer 和 temp_z_buffer 进行赋值。对较陡情况则变换 x 和 y 坐标，进行同样逻辑的遍历和赋值。

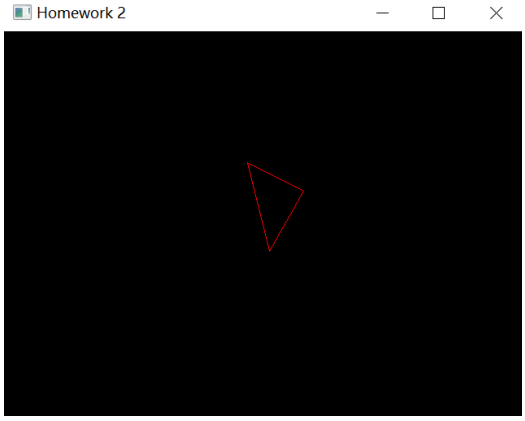
```
void MyGLWidget::bresenham(FragmentAttr& start, FragmentAttr& end, int id,map<int,map<int, FragmentAttr>>& line_l,map<int,map<int, FragmentAttr>>& line_r)
    int dx = abs(end.x - start.x), dy = abs(end.y - start.y), x, y, Iy, steps;
    double delta_x, delta_y, p;
    map<int, FragmentAttr> lside, rside;
    steps = (dx > dy) ? dx : dy;
    if (dx >= dy) {//斜率小于1
        x = (start.x < end.x) ? start.x : end.x;
        y = (start.x < end.x) ? start.y : end.y;
        delta_x = dx / (float)steps;delta_y = dy / (float)steps;
        p = 2 * delta_y - delta_x;
        if ((end.x - start.x) * (end.y - start.y) >= 0)Iy = 1;//判断y是+还是-
        else Iy = -1;
        for (int i = 0; i <= steps; i++) {
            FragmentAttr px = getLinearInterpolation(start, end, x, y);
            if (i == 0)lside[y] = px;
            if (i == steps)rside[y] = px;
            if (y <= WindowSizeH && y > 0 && x < WindowSizeW && x > 0) {//temp_render_buffer和temp_z_buffer赋值
                temp_render_buffer[(WindowSizeH - y) * WindowSizeW + x] = vec3(1, 0, 0);
                temp_z_buffer[(WindowSizeH - y) * WindowSizeW + x] = px.z;
            }
            if (p <= 0) p += 2 * delta_y;
            else if (i < steps) {
                rside[y] = px;
                y += Iy;
                lside[y] = getLinearInterpolation(start, end, x+1, y);
                p += 2 * (delta_y - delta_x);
            }
            x += 1;
        }
```

```
    }
    else {//斜率大于1
        x = (start.y < end.y) ? start.y : end.y;
        y = (start.y < end.y) ? start.x : end.x;
        delta_x = dy / (float)steps; delta_y = dx / (float)steps;
        p = 2 * delta_y - delta_x;
        if ((end.x - start.x) * (end.y - start.y) >= 0)Iy = 1;
        else Iy = -1;
        for (int i = 0; i <= steps; i++) {
            FragmentAttr px = getLinearInterpolation(start, end, y, x);
            rside[x] = lside[x] = px;
            if (x <= WindowSizeH && x > 0 && y < WindowSizeW && y > 0) {//temp_render_buffer和temp_z_buffer赋值
                temp_render_buffer[(WindowSizeH - x) * WindowSizeW + y] = vec3(1, 0, 0);
                temp_z_buffer[(WindowSizeH - x) * WindowSizeW + y] = px.z;
            }
            if (p <= 0) p += 2 * delta_y;
            else {
                y += Iy;
                p += 2 * (delta_y - delta_x);
            }
            x += 1;
        }
    }
    line_l[id] = lside;
    line_r[id] = rside;
}
```

调用 bresenham 绘制三角形

```
bresenham(transformedVertices[0], transformedVertices[1], 2, line_l, line_r);
bresenham(transformedVertices[1], transformedVertices[2], 0, line_l, line_r);
bresenham(transformedVertices[2], transformedVertices[0], 1, line_l, line_r);
```

运行结果如下：



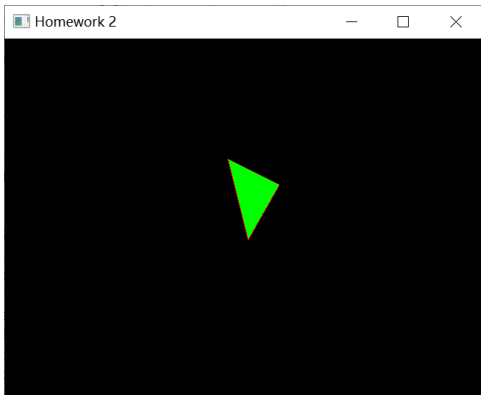

**1.3：用 edge-walking 填充三角形内部颜色**

实现 edge_walking 函数：

初始实现方案：遍历每行 temp_render_buffer，由左右两端遍历得到存在边的左右点，对左右点之间的 temp_render_buffer 赋相同值以及 temp_z_buffer 计算插值

```
int MyGLWidget::edge_walking() {
    int firstChangeLine = WindowSizeH;
    bool nochange = true;
    for (int h = 0; h < WindowSizeH; h++) {
        auto temp_render_row = &temp_render_buffer[h * WindowSizeW];
        auto temp_z_buffer_row = &temp_z_buffer[h * WindowSizeW];
        int l = 0, r = WindowSizeW - 1;
        bool lp = true, rp = true;
        while (l < r && (lp || rp)) {
            if (lp && temp_render_row[l] == vec3(0, 0, 0))l++;
            else lp = false;
            if (rp && temp_render_row[r] == vec3(0, 0, 0))r--;
            else rp = false;
        }
        float z_l = temp_z_buffer_row[l], z_r = temp_z_buffer_row[r];
        for (int i = l + 1; i < r; i++) {
            temp_render_row[i] = vec3(0, 1, 0);
            temp_z_buffer_row[i] = z_l + (i - l) * (z_r - z_l) / (float)(r - l);
        }
        if (nochange && !(lp && rp)) { firstChangeLine = h; nochange = false; }
    }
    return firstChangeLine;
}
```

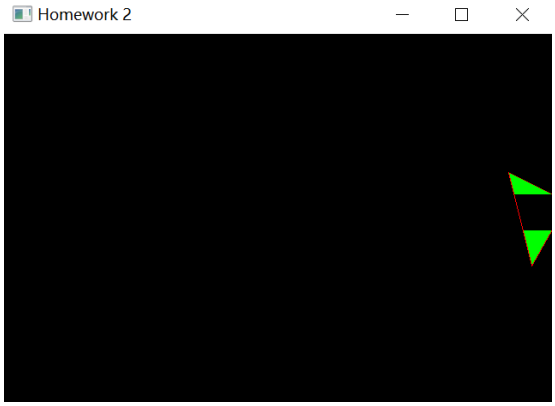
运行结果如下：



考虑到中间的插值计算非常依赖已渲染线段的值，对于部分线段超出窗口的情况则行不通，加入按键上下左右以达到移动图像的目的，之后可明显看出图像超出窗口会出现上面问题

```
void MyGLWidget::keyPressEvent(QKeyEvent *e) {

    switch (e->key()) {
        case Qt::Key_0: scene_id = 0;update(); break;
        case Qt::Key_1: scene_id = 1;update(); break;
        case Qt::Key_9: degree += 35;update(); break;
        case Qt::Key_Up: offset.y -= 5; update(); break;
        case Qt::Key_Down: offset.y += 5; update(); break;
        case Qt::Key_Left: offset.x -= 5; update(); break;
        case Qt::Key_Right: offset.x += 5; update(); break;
    }
}
```

如下图，暴露初始方案的明显弊端

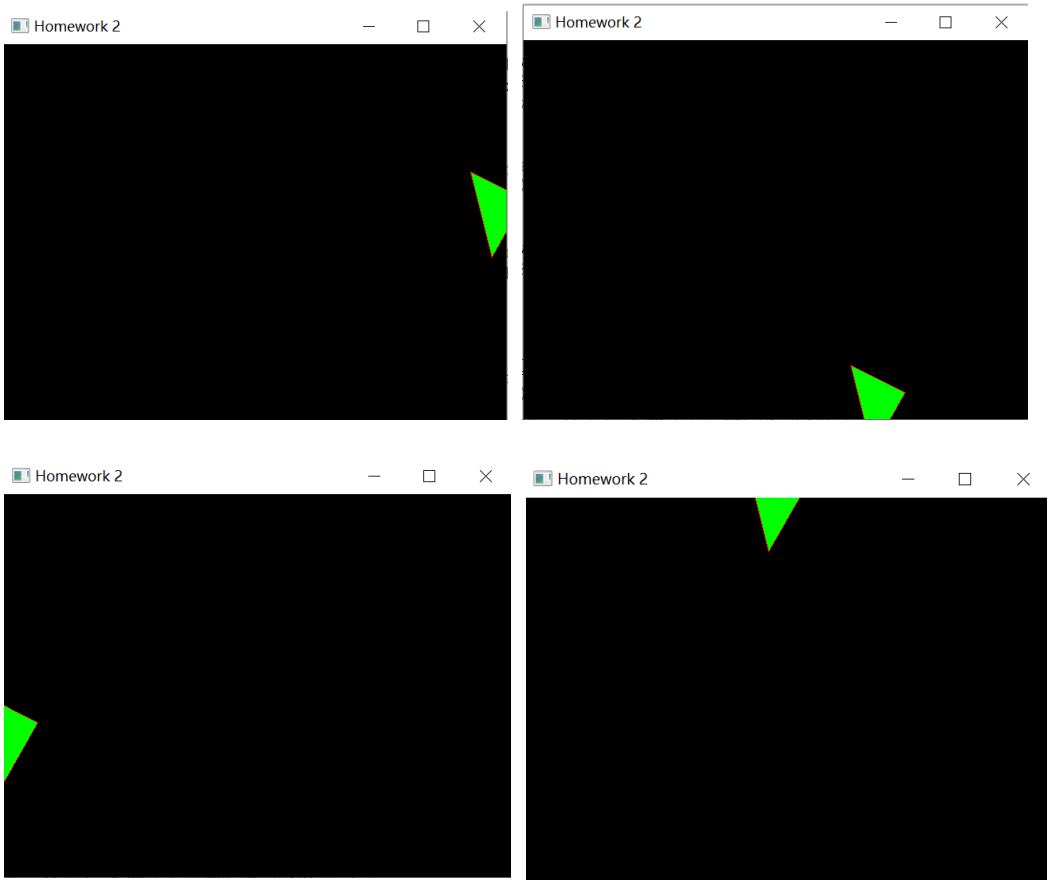



因此考虑存储三角形线段信息，即上面出现过的 line_l 和 line_r，存储了对应 id 线段在每一高度下最左点和最右点的信息，类型为 FragmentAttr。在调用 DDA 或 bresenham 存储其中，在 edgewalking 中即可通过扫描线段每行，根据该高度下两条线段上最接近的两点计算这两点之间所有点的信息。重载一个 edge_walking 函数，如下部分：

```
int MyGLWidget::edge_walking(FragmentAttr* v, std::map<int, std::map<int, FragmentAttr>> l, std::map<int, std::map<int, FragmentAttr>> r) {
    //line_id: v0-v1=2 ,v1-v2=0,v0-v2=1
    int max_id = ((v[0].y >= v[1].y) && (v[0].y >= v[2].y)) ? 0 : ((v[1].y >= v[2].y) ? 1 : 2);
    int mid_id = (max_id != 0 && (v[0].y- v[1].y)*(v[0].y - v[2].y)<=0) ? 0 : (((v[1].y - v[0].y) * (v[1].y - v[2].y) <= 0) ? 1 : 2);//高度位于中间的点id
    int min_id = 3 - max_id - mid_id;
    if (v[mid_id].x > r[mid_id][v[mid_id].y].x) {//向右扩展
        for (int h = v[min_id].y +1; h <= v[mid_id].y; h++) {//最低点到中间点的高度扫描，另一边line_id为max_id
            if(h <= WindowSizeH && h > 0)
            for (int i = r[mid_id][h].x + 1; i < WindowSizeW && i<l[max_id][h].x; i++) {
                if (i < 0)continue;
                temp_render_buffer[(WindowSizeH-h) * WindowSizeW + i] = vec3(0, 1, 0);//FragmentAttr插值或暂定为绿色
                temp_z_buffer[(WindowSizeH - h) * WindowSizeW + i] = r[mid_id][h].z+(i- r[mid_id][h].x)*(l[max_id][h].z -r[mid_id][h].z)/(float)(l[max_id
                //z_l + (i - l) * (z_r - z_l) / (float)(r - l);
            }
        }
        for (int h = v[mid_id].y; h < v[max_id].y; h++) {//中间点到最高点的高度扫描，另一边line_id为min_id
            if (h <= WindowSizeH && h > 0)
            for (int i = r[mid_id][h].x + 1; i < WindowSizeW && i < l[min_id][h].x; i++) {
                if (i < 0)continue;
                temp_render_buffer[(WindowSizeH - h) * WindowSizeW + i] = vec3(0, 1, 0);//FragmentAttr插值或暂定为绿色
                temp_z_buffer[(WindowSizeH - h) * WindowSizeW + i] = r[mid_id][h].z + (i - r[mid_id][h].x) * (l[min_id][h].z - r[mid_id][h].z) / (float)(
            }
        }
    }
```

```
    else if(v[mid_id].x < l[mid_id][v[mid_id].y].x){//向左扩展
        for (int h = v[min_id].y + 1; h <= v[mid_id].y; h++) {//最小点到中间点的高度扫描，另一边line_id为max_id
            if (h <= WindowSizeH && h > 0)
                for (int i = l[mid_id][h].x - 1; i >= 0 && i > r[max_id][h].x; i--) {
                    if (i >= WindowSizeW)continue;
                    temp_render_buffer[(WindowSizeH - h) * WindowSizeW + i] = vec3(0, 1, 0);//FragmentAttr插值或暂定为绿色
                    temp_z_buffer[(WindowSizeH - h) * WindowSizeW + i] = r[max_id][h].z + (i - r[max_id][h].x) * (l[mid_id][h].z - r[max_id][h].z) / (floa
                }
        }
        for (int h = v[mid_id].y+1; h < v[max_id].y; h++) {//最大点到中间点的高度扫描，另一边line_id为min_id
            if (h <= WindowSizeH && h > 0)
                for (int i = l[mid_id][h].x - 1; i >= 0 && i > r[min_id][h].x; i--) {
                    if (i >= WindowSizeW)continue;
                    temp_render_buffer[(WindowSizeH - h) * WindowSizeW + i] = vec3(0, 1, 0);//FragmentAttr插值或暂定为绿色
                    temp_z_buffer[(WindowSizeH - h) * WindowSizeW + i] = r[min_id][h].z + (i - r[min_id][h].x) * (l[mid_id][h].z - r[min_id][h].z) / (floa
                }
        }
    }
    return (WindowSizeH - v[max_id].y) > 0 ? (WindowSizeH - v[max_id].y) : 0;
}
```

最终结果正常：




**讨论要求：**

**1.4：针对不同面数的模型，从实际运行时间角度讨论 DDA、bresenham 的绘制效率**

通过多次记录多次旋转更新图形的时间，取平均值来衡量 DDA、bresenham 的效率，可看出二者的时间没有在正常误差范围内，效率相当。

| 模型 | DDA | Bresenham |
|---|---|---|
| teapot_600 | 3.0053 s | 2.6868 s |
| teapot_8000 | 28.2075 s | 28.2768 s |
| rock | 1.28 s | 1.2163 s |
| cube | 0.2696 s | 0.2712 s |
| singleTriangle | 0.014 s | 0.0142 s |

**2．实现光照、着色**

**2.1：用 Gouraud 实现三角形内部的着色**

**2.2：用 Phong 模型实现三角形内部的着色**

**2.3：用 Blinn-Phong 实现三角形内部的着色**

为 MyGLWidget 加入变量 `string Shadingtype; // Gouraud / Phong / blinnphong`

首先完成 PhoneShading 函数：依照环境光、漫反射和镜面反射的公式进行计算

$$\boldsymbol{I} = k_{ar}\boldsymbol{L}_{ar} + k_{dr}\boldsymbol{L}_{dr}(\boldsymbol{n}\cdot\boldsymbol{l}) + k_{sr}\boldsymbol{L}_{sr}\cdot\max\left((\boldsymbol{n}\cdot\boldsymbol{h})^{\alpha},0\right)$$

$$\boldsymbol{I} = k_{ag}\boldsymbol{L}_{ag} + k_{dg}\boldsymbol{L}_{dg}(\boldsymbol{n}\cdot\boldsymbol{l}) + k_{sg}\boldsymbol{L}_{sg}\cdot\max\left((\boldsymbol{n}\cdot\boldsymbol{h})^{\alpha},0\right)$$

$$\boldsymbol{I} = k_{ab}\boldsymbol{L}_{ab} + k_{db}\boldsymbol{L}_{db}(\boldsymbol{n}\cdot\boldsymbol{l}) + k_{sb}\boldsymbol{L}_{sb}\cdot\max\left((\boldsymbol{n}\cdot\boldsymbol{h})^{\alpha},0\right)$$

通过 Shadingtype 在该函数中选择使用 Phong 或是 Blinn-Phong 来计算镜面反射

```
vec3 MyGLWidget::PhoneShading(FragmentAttr& nowPixelResult) {
    mat4 viewMatrix = glm::lookAt(camPosition, camLookAt, camUp);
    vec3 pos_l = vec3(viewMatrix * vec4(lightPosition, 1.0f));//光源在相机坐标位置
    vec3 L = normalize(pos_l - nowPixelResult.pos_mv);//光线单位向量
    vec3 V = normalize(-nowPixelResult.pos_mv);//视线单位向量
    vec3 N = normalize(nowPixelResult.normal);//单位法向量
    vec3 La = vec3(0.4f, 0.4f, 0.4f), Ld = vec3(1.0f, 1.0f, 1.0f), Ls = vec3(1.0f, 1.0f, 1.0f);//光照强度
    vec3 Ka = vec3(0.2f, 0.2f, 0.2f), Kd = vec3(0.8f, 0.8f, 0.8f), Ks = vec3(0.8f, 0.8f, 0.8f);//光照系数
    int alpha = 4;
    vec3 Ia, Id, Is;
    Ia.x = Ka.x * La.x; Ia.y = Ka.y * La.y; Ia.z = Ka.z * La.z;
    float dot_nl = dot(N, L);
    if (dot_nl > 0) {
        Id.x = Kd.x * Ld.x * dot_nl; Id.y = Kd.y * Ld.y * dot_nl; Id.z = Kd.z * Ld.z * dot_nl;
        if (Shadingtype == "phong" || Shadingtype == "Gouraud") {
            vec3 vec_2d = vec3(N.x * dot_nl * 2, N.y * dot_nl * 2, N.z * dot_nl * 2);
            vec3 R = vec_2d - L;//镜面反射单位向量
            float dot_rv = dot(R, V);
            if (dot_rv > 0) {
                Is.x = Ks.x * Ls.x * pow(dot_rv, alpha); Is.y = Ks.y * Ls.y * pow(dot_rv, alpha); Is.z = Ks.z * Ls.z * pow(dot_rv, alpha);
            }else Is = vec3(0, 0, 0);
        }else if(Shadingtype == "blinnphong"){
            vec3 H = normalize(L + V);//半程向量
            float dot_nh = dot(N, H);
            if (dot_nh > 0) {
                Is.x = Ks.x * Ls.x * pow(dot_nh, alpha); Is.y = Ks.y * Ls.y * pow(dot_nh, alpha); Is.z = Ks.z * Ls.z * pow(dot_nh, alpha);
            }else Is = vec3(0, 0, 0);
        }else Is = vec3(0, 0, 0);
    }
    else {Id = vec3(0, 0, 0);Is = vec3(0, 0, 0);}
    vec3 color = Ia + Id + Is;
    return color;
}
```

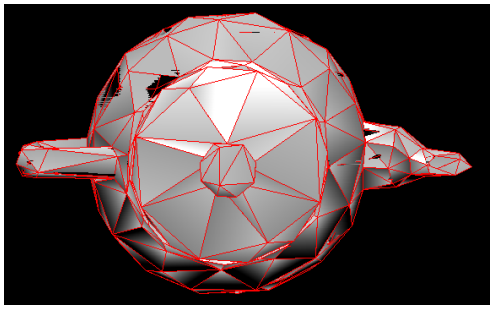
在 drawTriangle 函数中指定 Shadingtype，并为 transformedVertices 的 color 属性调用 PhoneShading 计算颜色，(以及下面注释为原代码计算的 z 深度，其中 camPosition 和 ver_mv 并不属于同一个坐标系下，相机在相机坐标系坐标为原点，故直接算 ver_mv 到原点的距离即可)

```
void MyGLWidget::drawTriangle(Triangle triangle) {
    Shadingtype = "Gouraud";// Gouraud / Phong / blinnphong

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

在 edge_walking 函数中可以根据前面定义的 FragmentAttr 类型方便赋值，若选择 Phong 或 blinnphong 则需重新根据光照模型计算颜色值。如以下 DDA 函数所示：

```
int MyGLWidget::edge_walking(FragmentAttr* v, std::map<int, std::map<int, FragmentAttr>> l, std::map<int, std::map<int, FragmentAttr>> r) {
    //line_id: v0-v1=2 ,v1-v2=0,v0-v2=1
    int max_id = ((v[0].y >= v[1].y) && (v[0].y >= v[2].y)) ? 0 : ((v[1].y >= v[2].y) ? 1 : 2);
    int mid_id = (max_id != 0 && (v[0].y- v[1].y)*(v[0].y - v[2].y)<=0) ? 0 : (((v[1].y - v[0].y) * (v[1].y - v[2].y) <= 0) ? 1 : 2);//高度位于中间的点id
    int min_id = 3 - max_id - mid_id;
    if (v[mid_id].x > r[mid_id][v[mid_id].y].x) {//向右扩展
        for (int h = v[min_id].y +1; h <= v[mid_id].y; h++) {//最低点到中间点的高度扫描，另一边line_id为max_id
            if(h <= WindowSizeH && h > 0)
            for (int i = r[mid_id][h].x + 1; i < WindowSizeW && i<l[max_id][h].x; i++) {
                if (i < 0)continue;
                FragmentAttr px = getLinearInterpolation(r[mid_id][h], l[max_id][h], i, h); if (Shadingtype != "Gouraud")px.color = PhoneShading(px);
                temp_render_buffer[(WindowSizeH-h) * WindowSizeW + i] = px.color;//FragmentAttr插值或暂定为绿色
                temp_z_buffer[(WindowSizeH - h) * WindowSizeW + i] = px.z; //r[mid_id][h].z+(i- r[mid_id][h].x)*(l[max_id][h].z -r[mid_id][h].z)/(float)(
                //z_l + (i - l) * (z_r - z_l) / (float)(r - l);
            }
        }
```

使用 Gouraud 得到以下效果：

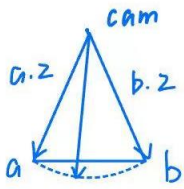


效果基本达到，在进行讨论要求前对效果进行完善，可看出上图在几处地方出现黑块，经分析判断这些黑块是通过 z 值的误差，导致被覆盖的部分反过来覆盖最前面的部分，因此尝试通过减少 z 值误差来改善该情况。

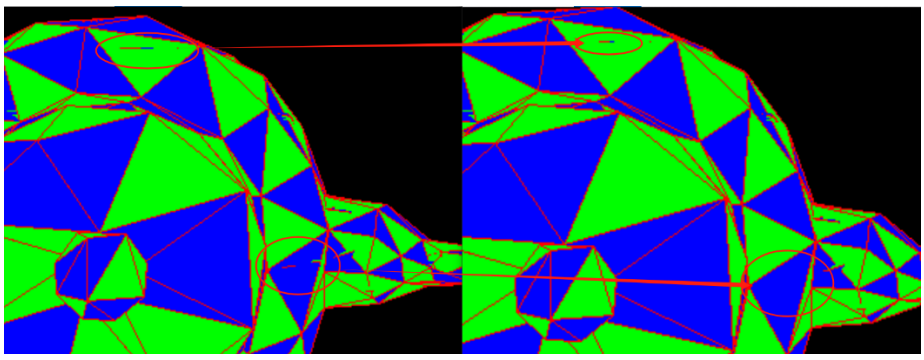
修改 getLinearInterpolation 函数，将除以浮点数运算移到最后一步，不使用插值计算 z 值，pos_mv 表示为点在相机坐标系的位置，通过该位置计算 z 值更准确。

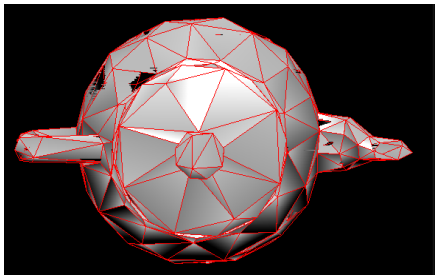
```
FragmentAttr getLinearInterpolation(const FragmentAttr& a, FragmentAttr& b, int x_position, int y_position) {
    FragmentAttr result;
    result.x = x_position;
    result.y = y_position;
    int mol,den;
    if (abs(a.x - b.x) >= abs(a.y - b.y)) {
        mol = x_position - a.x; den =b.x - a.x;
    }
    else {
        mol = y_position - a.y; den = b.y - a.y;
        result.z = ((b.y - a.y)*a.z + (y_position - a.y) * (b.z - a.z))/ float(b.y - a.y);
    }
    //result.z = a.z + t * (b.z - a.z);
    //插值
    result.color.r = (den * a.color.r + mol * (b.color.r - a.color.r)) / float(den);
    result.color.g = (den * a.color.g + mol * (b.color.g - a.color.g)) / float(den);
    result.color.b = (den * a.color.b + mol * (b.color.b - a.color.b)) / float(den);

    result.normal.x = (den * a.normal.x + mol * (b.normal.x - a.normal.x)) / float(den);
    result.normal.y = (den * a.normal.y + mol * (b.normal.y - a.normal.y)) / float(den);
    result.normal.z = (den * a.normal.z + mol * (b.normal.z - a.normal.z)) / float(den);

    result.pos_mv.x = (den * a.pos_mv.x + mol * (b.pos_mv.x - a.pos_mv.x)) / float(den);
    result.pos_mv.y = (den * a.pos_mv.y + mol * (b.pos_mv.y - a.pos_mv.y)) / float(den);
    result.pos_mv.z = (den * a.pos_mv.z + mol * (b.pos_mv.z - a.pos_mv.z)) / float(den);

    result.z = glm::length(result.pos_mv);
    return result;
}
```

从理论上看插值计算的 z 值会比实际值偏大，如考虑下面模型，z 的实际值与计算插值会有明显的误差。

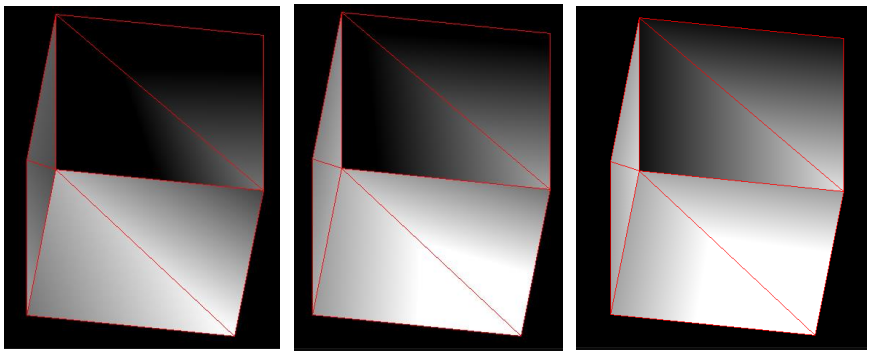



改进效果：一些异常颜色块得到消除



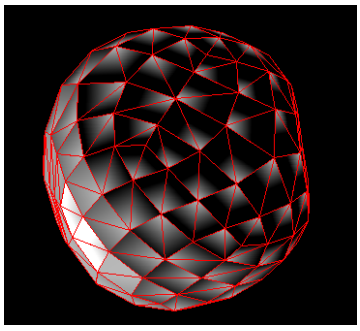
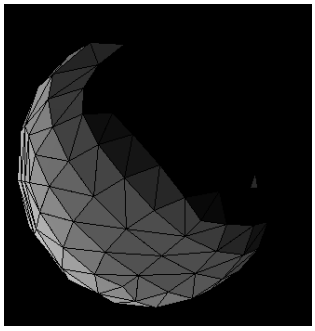
此时效果： 

关于光线位置的选择，目前仍是单个光源，对于一些立体且面数较少的时候，若光源离立体很近的时候，插值的误差会变大，导致一些颜色变化不自然，通过将 ray 位置调远可缩小光线角度，缓解物体一个三角形内顶点颜色差距过大的问题。下图为逐渐调远光源位置对比

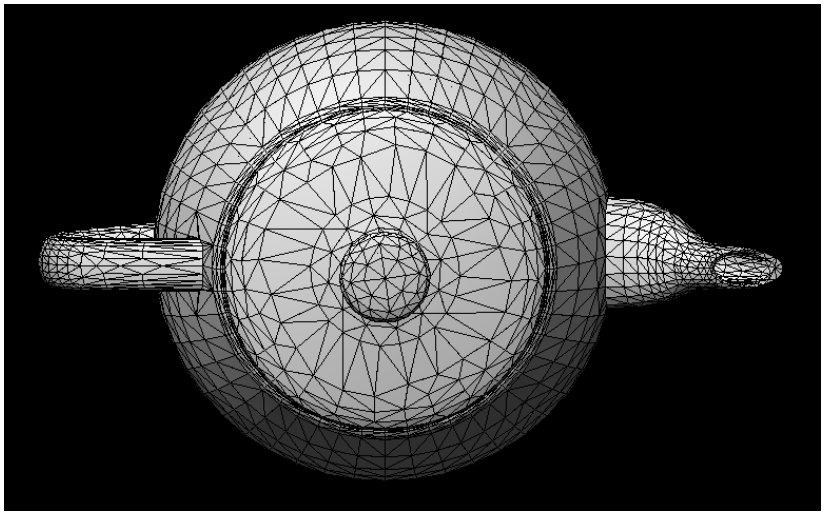


考虑 rock 的背部，会出现一些边角的白色，原因是根据顶点法向量进行计算颜色得到白色，无论是插值还是映射都会使得三角面中存在白色到黑色的渐变，考虑将三个顶点求平均法向量作为整个三角面的法向量再进行颜色计算即可解决。（边颜色改成了黑色）

```
vec3 avg_normal = normalize(normalize(v[0].normal) + normalize(v[1].normal) + normalize(v[2].normal));
```

改前： 
改后： 

结果：光源设置在适中位置体现高光，由左上角到右下角体现高光、漫反射和环境光的特点



**讨论要求：**

**2.4：结合实际运行时间讨论三种不同着色方法的效果、着色效率**

使用 Bresenham 绘制三角形

Gouraud 根据光照模型计算三角形的顶点颜色，对内部进行插值

Phong 计算三角形内部法向量，再根据光照模型用 phong 计算内部所有点颜色

Blinn-Phong 计算三角形内部法向量，根据光照模型用 Blinn-Phong 计算内部所有点颜色

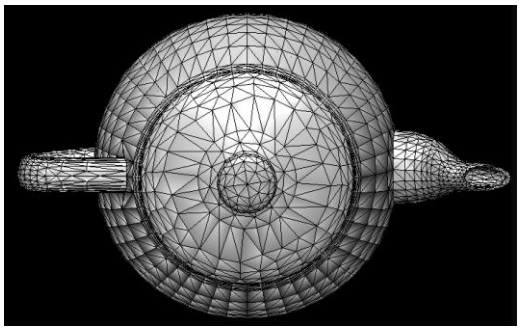
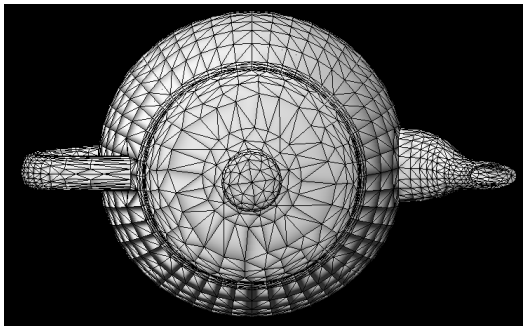
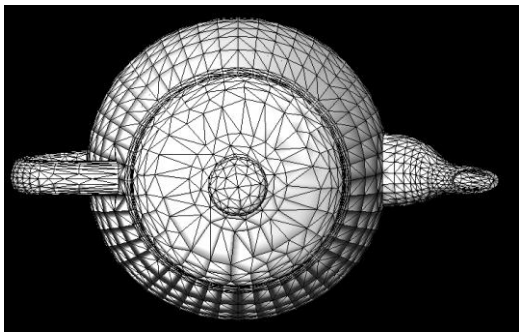
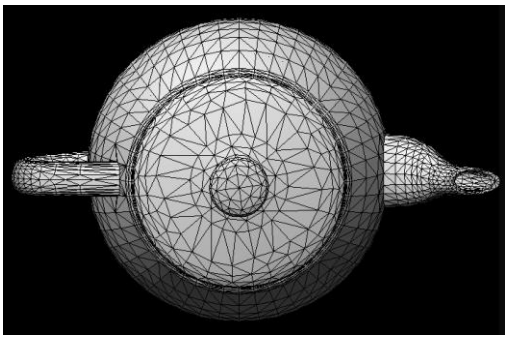
| | Gouraud | Phong | Blinn-Phong |
|---|---|---|---|
| rock | **1.1878 s** | **1.2817 s** | **1.2836 s** |
| teapot_600 | **2.5857 s** | **2.6519 s** | **2.7058 s** |
| teapot_8000 | **26.8785 s** | **27.2188 s** | **27.1017 s** |

Gouraud 省去了内部法向量的计算，着色效率要快于另外两种，Phong 与 Blinn-Phong 差别只在于计算 shading 镜面反射的不同，因此效率差距无明显区别。

下面展示[1][2]分别为 Gouraud、Phong、Blinn-Phong 以及自己改进之后的效果图

[3][4]

可以看出不管是对颜色插值还是对法向量插值，都容易导致颜色黑白分明的块状，再对一整个的三角形面统一法向量后再通过光照着色，得出来的颜色分布更加自然。

**3．最终效果**

**3.1：在 scene_0 中，加载单一三角形情况下辖，展示 DDA-EdgeWalking 与 Bresenham-EdgeWalking 两种方法下，三角形光栅化是否正确。**

**3.2：在 scene_1 中，加载复杂立体图形，用 Bresenham-EdgeWalking 方法，展示三种模型，在同一角度的光照效果（角度要能体现环境光、散射光、高光等）**

结果均在以上展示过，便不重复展示

**4．问题与总结：**

本次作业主要在于对光栅化算法以及光照着色原理的理解，在一开始实现光栅化的过程中对 DDA 和 bresenham 的理解还有些偏差，在实际实现中需要考虑的情况也不少，从画线段开始就频频出错，简单地绘线过程需要在很多细节上琢磨清楚，考虑极端情况，在 edgeWalking 中的情况更是，实现的大致方法不难，比较耗费心思的是实现以后看到立体图像冒出的一个又一个的问题，需要反复排查，仍有一些地方有待完善，因为时间问题就先到此结束。