计算机图形学作业二

【专业】计算机科学与技术　　【学号】22336259　　【姓名】谢宇桐

一、实现三角形的光栅化算法

1.1 用 DDA 实现三角形边的绘制

算法原理：

DDA算法（Digital Differential Analyzer）： DDA算法是一种基于差分的算法，通过逐个增加x和y的步长来绘制直线。其基本原理是计算出两点之间的x和y的差值，然后根据斜率选择较大的差值来确定下一点的位置。

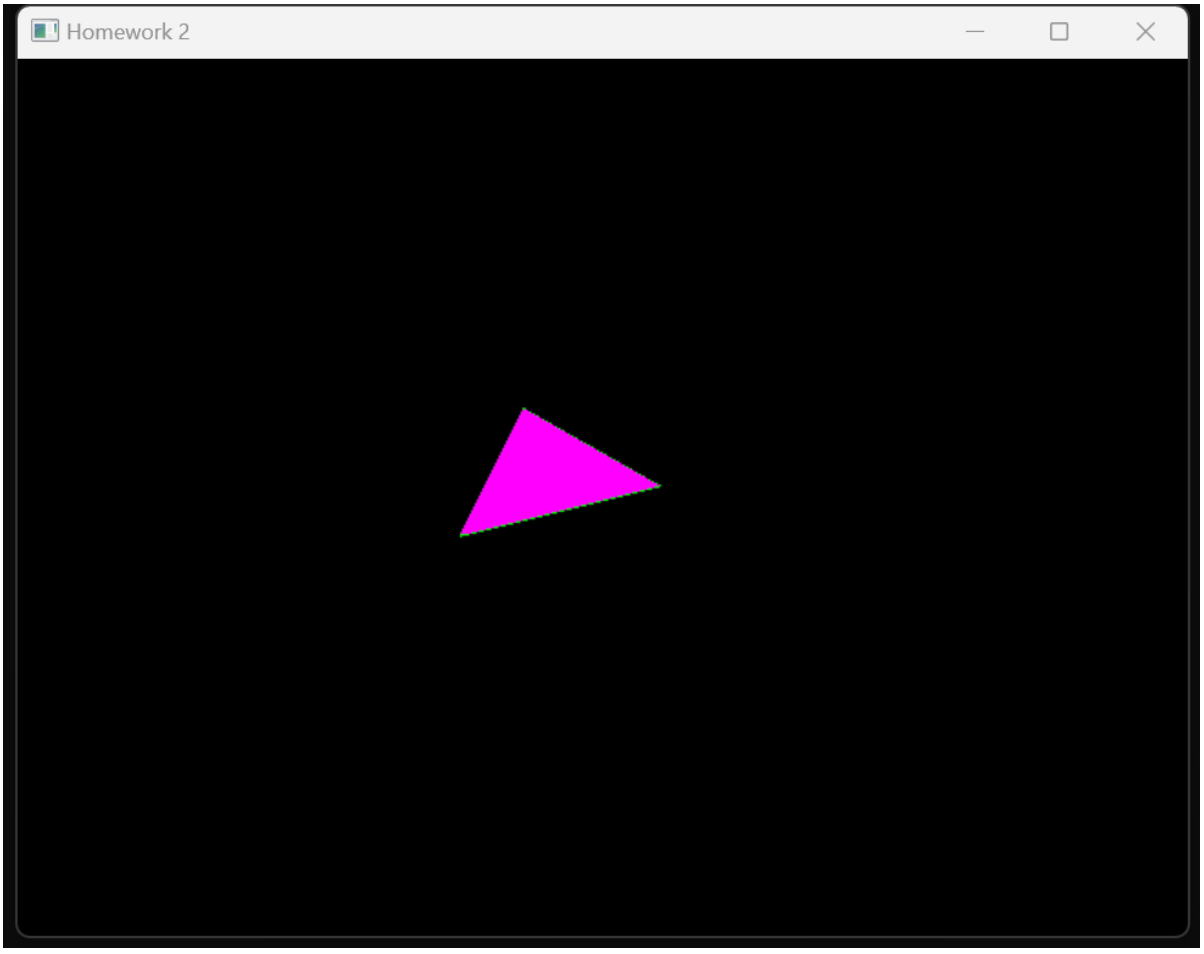
```
void MyGLWidget::DDA(FragmentAttr& start, FragmentAttr& end, int id) {
    // 计算直线的增量和步数
    int dx = end.x - start.x;
    int dy = end.y - start.y;
    int steps = std::max(abs(dx), abs(dy)); // 步骤数基于最大的dx或dy

    // 计算增量
    float xIncrement = dx / (float)steps;
    float yIncrement = dy / (float)steps;
    float zIncrement = (end.z - start.z) / (float)steps; // 添加深度变化率

    // 初始化坐标和深度
    float x = start.x;
    float y = start.y;
    float z = start.z; // 初始化z为起点深度

    //光栅化直线
    for (int i = 0; i <= steps; i++) {
        int ix = static_cast<int>(x);
        int iy = static_cast<int>(y);
        // 确保坐标在视窗范围内
        if (ix >= 0 && ix < WindowSizeW && iy >= 0 && iy < WindowSizeH) {
            temp_render_buffer[ix * WindowSizeW + iy] = vec3(0, 255, 0); // 绘制绿
色像素
            temp_z_buffer[ix * WindowSizeW + iy] = z; // 更新深度信息
        }
        x += xIncrement;
        y += yIncrement;
        z += zIncrement; // 更新深度值
    }
}
```

在drawTriangle函数中仿照代码中给出的bresenham调用方式补充：

```
DDA(transformedVertices[0], transformedVertices[1], 1);
DDA(transformedVertices[1], transformedVertices[2], 2);
DDA(transformedVertices[2], transformedVertices[0], 3);
```

这里要搭配edge_walking函数一起使用。因为DDA和Bresenham算法主要用于绘制直线，而edge_walking算法用于填充三角形内部的像素。没有edge_walking算法，程序将只能绘制三角形的边界，而无法填充三角形内部。edge_walking代码我将在后面给出。

代码运行如下，可以看到绿色的边框和粉色的填充：



1.2 用 bresenham 实现三角形边的绘制

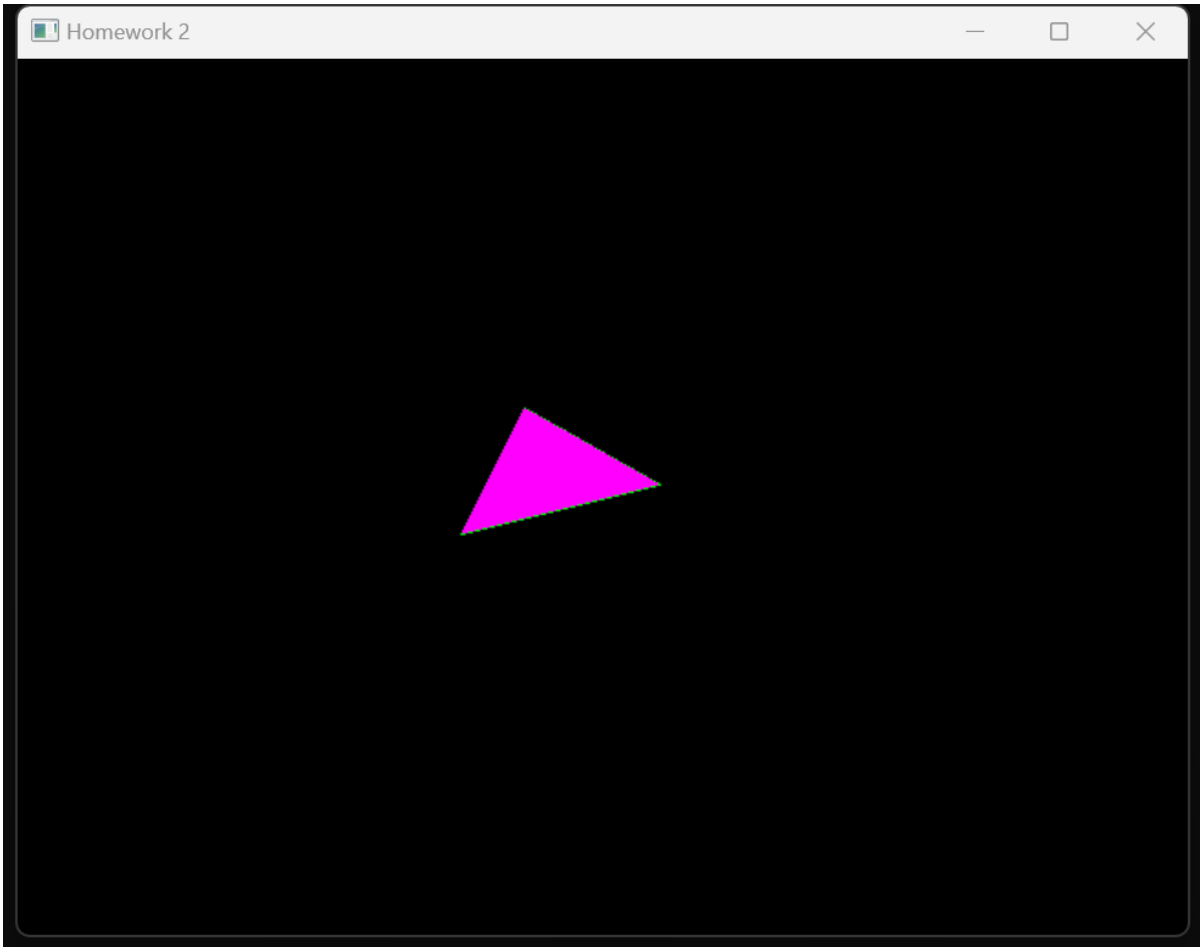
算法原理：

Bresenham算法： Bresenham算法是一种整数算法，通过判断像素点到直线的距离来确定下一个像素点的位置。该算法通过利用递推关系式，避免了浮点数运算，提高了绘制速度。

```
void MyGLWidget::bresenham(FragmentAttr& start, FragmentAttr& end, int id) {
    int x0 = static_cast<int>(start.x);
    int y0 = static_cast<int>(start.y);
    int x1 = static_cast<int>(end.x);
    int y1 = static_cast<int>(end.y);

    // 计算差值
    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);
    // 确定步进方向，1为向右或向下
    int sx = x0 < x1 ? 1 : -1;
    int sy = y0 < y1 ? 1 : -1;
    // 初始化误差值
    int err = (dx > dy ? dx : -dy) / 2;
    // 绘制直线
```

```
    int e2;
    while (true) {
        temp_render_buffer[x0 * WindowSizeW + y0] = vec3(0, 255, 0); // 绘制绿色像素，表示直线的一个像素
        temp_z_buffer[x0 * WindowSizeW + y0] = start.z; // 更新深度信息

        if (x0 == x1 && y0 == y1) break;

        e2 = err;// 当前的误差值。如果e2大于-dx，则减少err的值，并在x方向上移动一个像素。如果e2小于dy，则增加err的值，并在y方向上移动一个像素。
        if (e2 > -dx) {
            err -= dy;
            x0 += sx;
        }
        if (e2 < dy) {
            err += dx;
            y0 += sy;
        }
    }
}
```

代码运行结果如下，同样可以看到绿色的边框和粉色的填充：



1.3 用 edge-walking 填充三角形内部颜色

原理：

edge_walking 是一种用于填充多边形内部像素的算法，特别是在光栅化过程中。它基于扫描线算法，通过确定多边形每条扫描线（水平线）上的左右边界，然后填充这两个边界之间的像素。

1. **扫描线**：算法从屏幕的最顶端开始，逐行向下扫描，直到达到屏幕的底部。
2. **边界检测**：对于每一行，算法会检测多边形的左右边界。这通常是通过检查多边形的边与当前扫描线的交点来实现的。
3. **像素填充**：一旦确定了一行的左右边界，算法就会填充这两个边界之间的像素，从而实现多边形内部的填充。

```
int MyGLWidget::edge_walking() {
    int firstChangeLine = WindowSizeH; // 初始化第一个变化行的位置为屏幕底部
    for (int x = 0; x < WindowSizeH; x++) { // 遍历每一行
        bool inside = false; // 是否在多边形内部的标志
        int start = 0; // 一行中多边形的左边界
        int end = 0; // 一行中多边形的右边界
        bool foundStart = false; // 是否找到左边界的标志
        float startDepth = 99999.0f, endDepth = 99999.0f; // 存储边界点的深度值

        for (int y = 1; y < WindowSizeW; y++) { // 遍历每一列
            if (!inside && temp_render_buffer[x * WindowSizeW + y] != vec3(0, 0,
0) && temp_render_buffer[x * WindowSizeW + y - 1] == vec3(0, 0, 0)) {
                inside = true; // 发现多边形左边界
                start = y; // 记录左边界位置
                foundStart = true; // 标记已找到左边界
                startDepth = temp_z_buffer[x * WindowSizeW + y]; // 获取起始边界的深
度值
            } else if (inside && temp_render_buffer[x * WindowSizeW + y] !=
vec3(0, 0, 0) && temp_render_buffer[x * WindowSizeW + y - 1] == vec3(0, 0, 0)) {
                end = y; // 发现多边形右边界
                endDepth = temp_z_buffer[x * WindowSizeW + y]; // 获取结束边界的深度
值
                break; // 找到一行中的起始和结束边界后退出循环
            }
        }

        if (foundStart) { // 如果找到了边界
            firstChangeLine = std::min(firstChangeLine, x); // 更新第一个变化行的位置

            for (int y = start; y <= end; y++) { // 遍历当前行的边界之间
                float depth = startDepth + (endDepth - startDepth) * ((y - start)
/ (float)(end - start + 1)); // 计算插值深度
                temp_z_buffer[x * WindowSizeW + y] = depth; // 更新深度缓冲区
                temp_render_buffer[x * WindowSizeW + y] = vec3(10, 0, 255); // 填
充像素为粉色
            }
        }
    }
    return firstChangeLine; // 返回第一个变化行的位置
}
```

运行结果如上。

1.4 讨论：从实际运行时间角度讨论 DDA、bresenham 的绘制效率。

我将在代码中添加每个方案的运行时间，并使用按键操作进行对比：

```
// 添加2 3按键分别代表DDA和bresenham
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_2: draw_id = 2; std::cout << "DDA:" << std::endl; update();
break;
    case Qt::Key_3: draw_id = 3; std::cout << "bresenham:" << std::endl;
update(); break;
    case Qt::Key_9: degree += 35; update(); break;
    }
}
```

别忘了在myglwidget.h中添加：

```
class MyGLWidget : public QOpenGLWidget{
    // ...
private:
    // ...
    int draw_id;
    // ...
}
```

在drawTriangle函数修改：

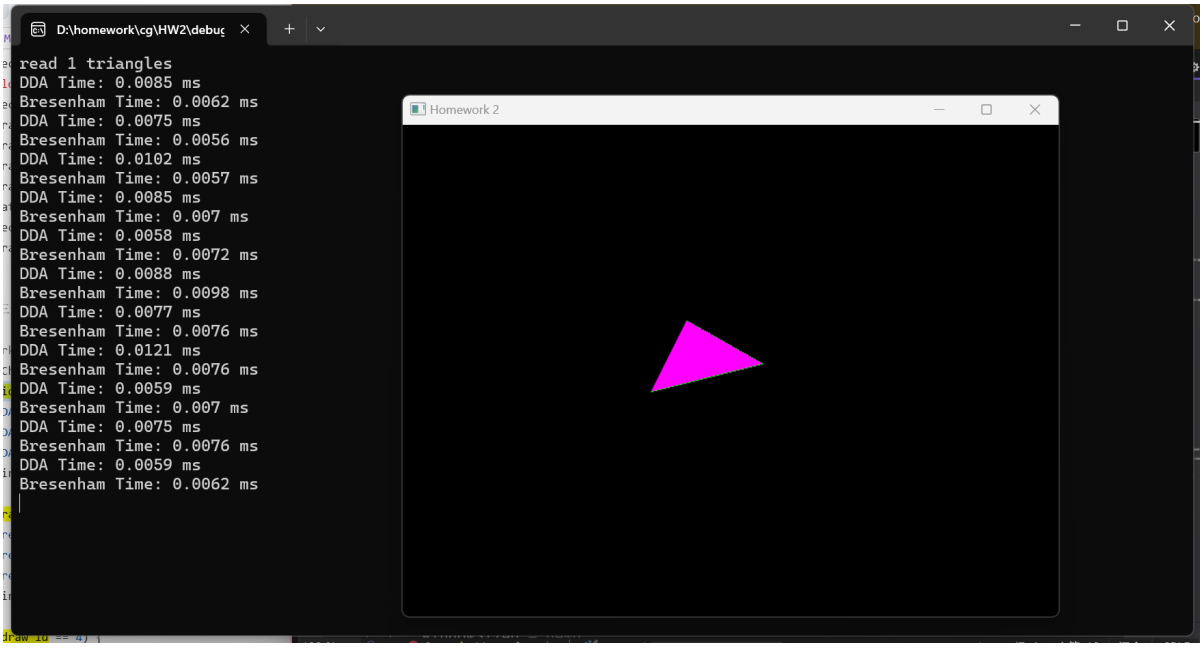
```
    // HomeWork: 1、绘制三角形三边
    // 使用DDA算法绘制三角形
    if(draw_id == 2){
        DDA(transformedVertices[0], transformedVertices[1], 1);
        DDA(transformedVertices[1], transformedVertices[2], 2);
        DDA(transformedVertices[2], transformedVertices[0], 3);
    }

    // 使用Bresenham算法绘制三角形
    if(draw_id == 3){
        bresenham(transformedVertices[0], transformedVertices[1], 1);
        bresenham(transformedVertices[1], transformedVertices[2], 2);
        bresenham(transformedVertices[2], transformedVertices[0], 3);
    }


    // HomeWork: 2: 用edge-walking填充三角形内部到temp_buffer中
    int firstChangeLine = edge_walking();
```

在scene_0中加入计时代码：

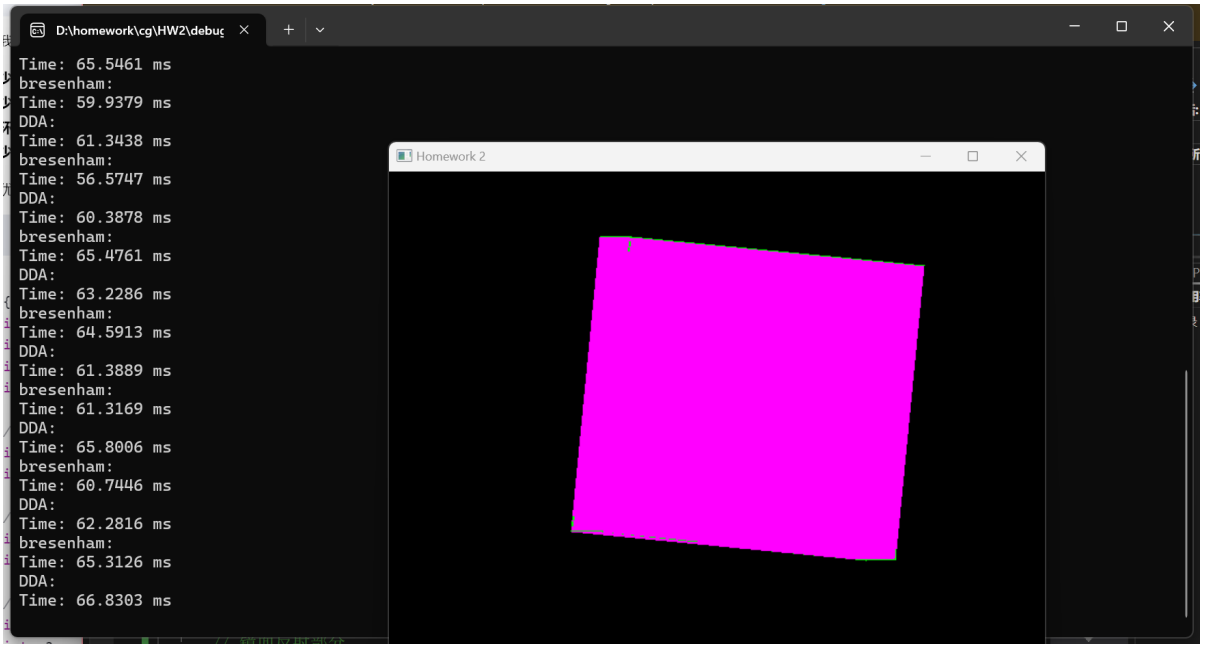
```
void MyGLWidget::scene_0()
{
    // 选择要加载的model
    objModel.loadModel("./objs/singleTriangle.obj");

    // 自主设置变换矩阵
    camPosition = vec3(100 * sin(degree * 3.14 / 180.0) +
objModel.centralPoint.y, 100 * cos(degree * 3.14 / 180.0) +
objModel.centralPoint.x, 10 + objModel.centralPoint.z);
    camLookAt = objModel.centralPoint;     // 例如，看向物体中心
    camUp = vec3(0, 1, 0);          // 上方向向量
    projMatrix = glm::perspective(radians(20.0f), 1.0f, 0.1f, 2000.0f);

    // 单一点光源，可以改为数组实现多光源
    lightPosition = objModel.centralPoint + vec3(0, 100, 100);
    clearBuffer(render_buffer);
    clearZBuffer(z_buffer);

    auto start_time = std::chrono::high_resolution_clock::now();// 开始时间

    for (int i = 0; i < objModel.triangleCount; i++) {
        Triangle nowTriangle = objModel.getTriangleByID(i);
        drawTriangle(nowTriangle);
    }

    auto end_time = std::chrono::high_resolution_clock::now(); // 结束时间
    // 计算并输出算法的执行时间
    std::chrono::duration<double, std::milli> ddaTime = end_time - start_time;
    std::cout << "Time: " << ddaTime.count() << " ms" << std::endl;

    glClear(GL_COLOR_BUFFER_BIT);
    renderWithTexture(render_buffer, WindowSizeH, WindowSizeW);
}
```

多次切换，运行不同模型，结果如下：

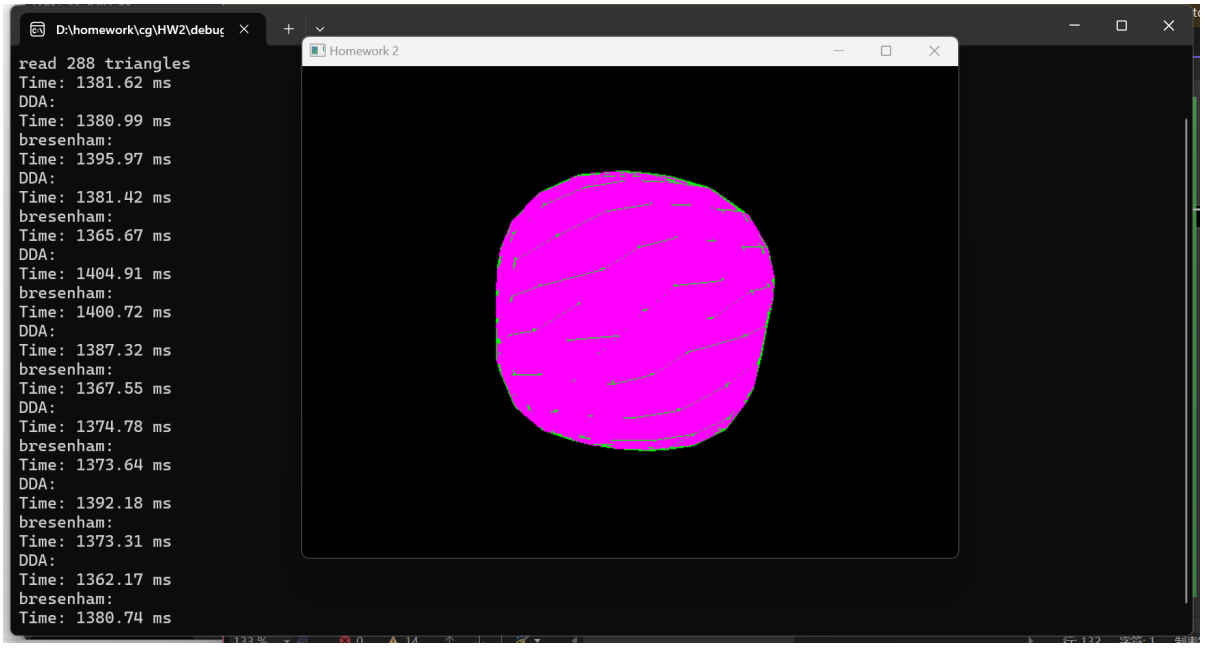
单面三角形：



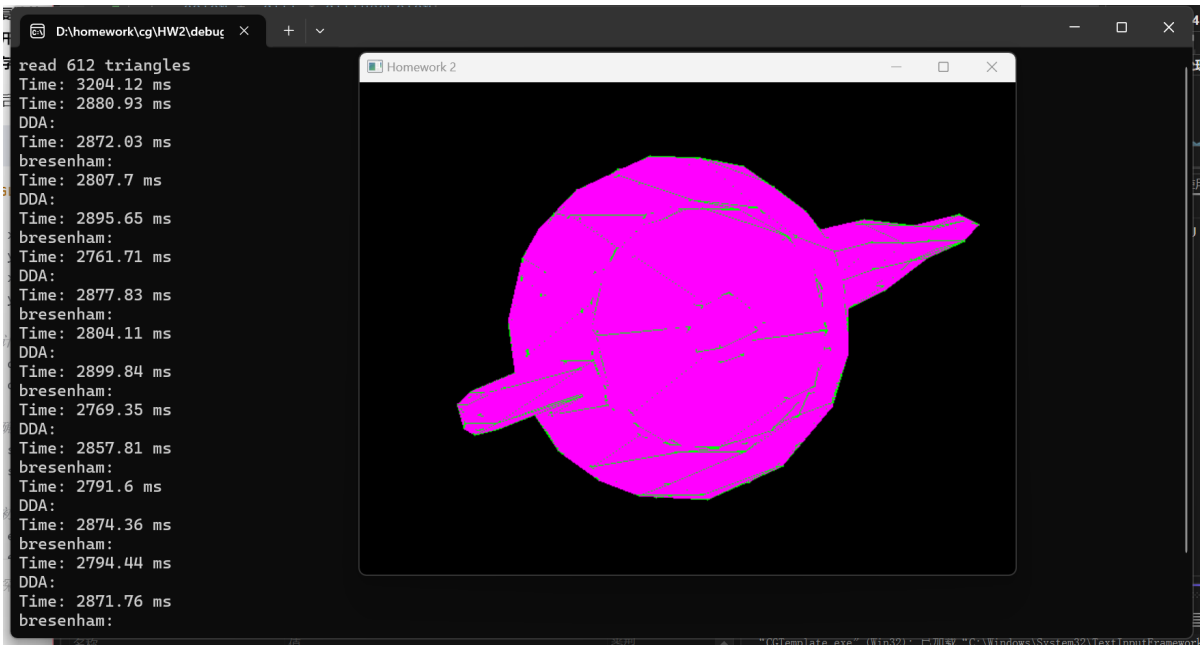
12面方块：



300面石块：



600面茶壶：



从图上我们可以看到，从平均时间上说，DDA的绘制时间普遍比bresenham长。尤其当面数越多时，差距越明显。

从理论上来讲，**DDA算法需要进行浮点数的加法和乘法运算**，这在某些硬件上比整数运算慢，而且它需要计算每一步的x和y坐标，以及深度值的插值。而**bresenham是一种整数运算**，这通常比浮点数运算快，且它不许要计算每一步的x和y坐标，而是通过决策变量来确定下一个像素的位置。所以从结论上说Bresenham算法在时间效率上通常优于DDA算法，尤其是在需要高精度和快速执行的图形应用中。

因此我们可以推出对于不同面数的模型，bresenham 的绘制效率比DDA高。面数越高的模型，bresenham绘制效率越比DDA高。

二、实现光照、着色

2.1：用 Gouraud 实现三角形内部的着色

Gouraud 着色是一种在每个顶点计算光照的着色技术，然后通过对顶点颜色进行插值来确定像素的颜色。这种方法假设每个顶点的法线是平面的，并且整个多边形表面上的光照是均匀的。

原理：

在每个顶点处计算光照，包括环境光、漫反射和高光。
使用顶点颜色在顶点之间进行线性插值来确定多边形上每个像素的颜色。
由于颜色插值是在顶点之间进行的，所以表面看起来比较平滑，但实际上整个多边形的颜色变化在顶点处是不连续的。

```
vec3 MyGLWidget::GouraudShading(FragmentAttr& fragment) {
    vec3 ambientColor = vec3(0.1, 0.1, 0.1);  // 环境光颜色
    vec3 lightColor = vec3(1.0, 1.0, 1.0);    // 光源颜色
    vec3 materialColor = vec3(1.0, 1.0, 1.0); // 材料颜色

    // 环境光部分
    vec3 ambient = ambientColor * materialColor;

    // 漫反射部分
    vec3 norm = normalize(fragment.normal);
    vec3 lightDir = normalize(lightPosition - vec3(fragment.pos_mv));
```

```
    float diff = max(dot(norm, lightDir), 0.0f);
    vec3 diffuse = diff * lightColor * materialColor;

    // 总光照强度
    vec3 intensity = ambient + diffuse;

    return intensity;
}
```

每次添加一个新的函数记得要将该函数添加到头文件中，全部改动如下，后面不再赘述：

```
class MyGLWidget : public QOpenGLWidget{
    // ...
private:
    // ...
     int edge_walking(FragmentAttr transformedVertices[3]);
     void DDA(FragmentAttr& start, FragmentAttr& end, int id);
     void bresenham(FragmentAttr& start, FragmentAttr& end, int id);
     void bresenham_light(FragmentAttr& start, FragmentAttr& end, int id);
     //...
     vec3 GouraudShading(FragmentAttr& nowPixelResult);
     vec3 PhongShading(FragmentAttr& nowPixelResult);
     vec3 BlinnPhongShading(FragmentAttr& nowPixelResult);

     int draw_id;
    // ...
}
```

bresenham:

bresenham 函数的主要任务是边缘绘制，它不关心具体的光照计算。光照计算（如法线插值、光照模型计算）通常在其他部分进行。所以这三种方法的调用是一样的。而因为 bresenham_light 函数被设计为处理光照效果，特别是我加入了绘制线段，在绘制线段时需要考虑线段两端点的光照颜色。

我把它写为bresenham_light，后面将不重复写：

```
void MyGLWidget::bresenham_light(FragmentAttr& start, FragmentAttr& end, int id)
{
    //初始化与bresenham一样

    vec3 start_color, end_color, color; //加入两个顶点的颜色

    switch (draw_id) {
    case 4:
        start_color = GouraudShading(start);
        end_color = GouraudShading(end);
        break;
    case 5:
        start_color = PhongShading(start);
        end_color = PhongShading(end);
        break;
    case 6:
        start_color = BlinnPhongShading(start);
        end_color = BlinnPhongShading(end);
        break;
```

```
    }
    start.color = start_color;
    end.color = end_color;
    temp_render_buffer[start.x * WindowSizeW + start.y] = start_color;
    temp_render_buffer[end.x * WindowSizeW + end.y] = end_color;

    int e2;

    int totalSteps = max(abs(end.x - start.x), abs(end.y - start.y));
    float step = 0;

    while (true) {
        // 计算插值深度
        float depth = start.z + (end.z - start.z) * (step / (float)totalSteps);

        // 计算当前步的颜色
        vec3 color = start_color + (end_color - start_color) * (step /
(float)totalSteps);

        // 更新边的颜色和深度缓冲区
        temp_render_buffer[x0 * WindowSizeW + y0] = color;
        temp_z_buffer[x0 * WindowSizeW + y0] = depth;

        if (x0 == x1 && y0 == y1) break;

        e2 = err;
        if (e2 > -dx) {
            err -= dy;
            x0 += sx;
        }
        if (e2 < dy) {
            err += dx;
            y0 += sy;
        }

        step++;
    }
}
```

而在图形渲染中，edge_walking 用于不同的绘制目的，每种方法对像素的处理方式不同，因此需要相应的变化来适应不同的光照模型。

edge_walking:

在 Gouraud 着色中，edge_walking 函数需要计算每个顶点的颜色，并在三角形内部进行插值。因此，edge_walking 函数需要访问顶点的颜色信息，并根据顶点颜色和位置信息计算三角形内部的像素颜色。

```
int MyGLWidget::edge_walking(FragmentAttr transformedVertices[3]) {
    int firstChangeLine = WindowSizeH;

    for (int x = 0; x < WindowSizeH; x++) {
        bool inside = false;
        int start = 0;
        int end = 0;
```

```
        bool foundStart = false;
        float startDepth = 99999.0f, endDepth = 99999.0f;
        vec3 startColor, endColor;

        for (int y = 1; y < WindowSizeW; y++) {
            // 检测一行中，三角形左边的边界
            if (!inside && temp_render_buffer[x * WindowSizeW + y] != vec3(0, 0,
0) && temp_render_buffer[x * WindowSizeW + y - 1] == vec3(0, 0, 0)) {
                inside = true;
                start = y;
                foundStart = true;
                startDepth = temp_z_buffer[x * WindowSizeW + y];
                startColor = temp_render_buffer[x * WindowSizeW + y]; // 获取起始边
界的颜色
            }
            // 检测一行中，三角形右边的边界
            else if (inside && temp_render_buffer[x * WindowSizeW + y] != vec3(0,
0, 0) && temp_render_buffer[x * WindowSizeW + y - 1] == vec3(0, 0, 0)) {
                end = y;
                endDepth = temp_z_buffer[x * WindowSizeW + y];
                endColor = temp_render_buffer[x * WindowSizeW + y]; // 获取结束边界
的颜色
                break;
            }
        }

        if (foundStart) {
            firstChangeLine = std::min(firstChangeLine, x);

            for (int y = start; y <= end; y++) {
                float t = (float)(y - start) / (float)(end - start + 0.01f);
                float depth = startDepth + (endDepth - startDepth) * t;

                vec3 color = vec3(0.0f, 0.0f, 0.0f); // 声明color变量
                switch (draw_id) {
                //...别的着色方法，在这里为方便阅读不显示
                case 4:{
                    //计算每个顶点的颜色
                    color = vec3(startColor.x + (endColor.x - startColor.x) * t,
                                 startColor.y + (endColor.y - startColor.y) * t,
                                 startColor.z + (endColor.z - startColor.z) * t);
                    break;
                }
                //...
                default:
                    break;
                }

                temp_render_buffer[x * WindowSizeW + y] = color; // 填充颜色
                temp_z_buffer[x * WindowSizeW + y] = depth;
            }
        }
    }

    return firstChangeLine;
}
```

运行结果将在2.4集中显示。

2.2：用 Phong 模型实现三角形内部的着色

Phong 着色是一种更为复杂的光照模型，它在每个像素上计算光照，可以产生更平滑和真实的高光效果。

原理：

计算每个顶点的法线向量，然后对法线向量进行插值，得到每个像素点的法线向量。
在每个像素处计算光照，包括环境光、漫反射和镜面反射。
漫反射计算考虑了光源方向和表面法线之间的关系。
镜面反射（高光）计算考虑了观察方向、光源方向和表面法线之间的关系，以及高光的锐利度（由光泽度指数控制）。

```
vec3 MyGLWidget::PhongShading(FragmentAttr& fragment) {
    vec3 ambientColor = vec3(0.1, 0.1, 0.1);  // 环境光颜色
    vec3 diffuseColor = vec3(0.9, 0.9, 0.9); // 漫反射颜色
    vec3 specularColor = vec3(0.2, 0.2, 0.2); // 镜面反射颜色
    vec3 lightColor = vec3(1.0, 1.0, 1.0);    // 光源颜色

    // 环境光分量
    vec3 color = ambientColor;

    // 漫反射分量
    vec3 lightDir = normalize(lightPosition - vec3(fragment.pos_mv)); // 光源方向
    vec3 norm = normalize(fragment.normal); // 法线
    float diff = max(dot(norm, lightDir), 0.0f); // 漫反射强度
    color += diff * diffuseColor;

    // 镜面反射分量
    vec3 viewDir = normalize(camPosition - vec3(fragment.pos_mv)); // 观察方向
    vec3 reflectDir = reflect(-lightDir, norm); // 反射方向
    float spec = pow(max(dot(viewDir, reflectDir), 0.0f), 32); // 镜面反射强度
    color += spec * specularColor;

    return color * lightColor; // 乘以光源颜色
}
```

edge_walking:

在 Phong 着色中，edge_walking 函数需要计算每个像素的法线向量，并在每个像素处计算光照。因此，edge_walking 函数需要访问顶点的法线向量，并在三角形内部进行插值，以计算每个像素的法线向量和相应的光照颜色。

```
int MyGLWidget::edge_walking(FragmentAttr transformedVertices[3]) {
    // 与第一题的edge_walking一致，不需要startcolor和endcolor
        if (foundStart) {
            firstChangeLine = std::min(firstChangeLine, x);

            for (int y = start; y <= end; y++) {
                //...
```

```
                case 5:{
                    FragmentAttr tmp = FragmentAttr(x, y, 0, 0);
                    std::pair<vec3, vec3> result = Interpolate(x, y,
transformedVertices);
                    tmp.normal = result.first;
                    tmp.pos_mv = result.second;
                    color = PhongShading(tmp);
                    break;
                }
                //...
                default:
                    break;
                }

                temp_render_buffer[x * WindowSizeW + y] = color; // 填充颜色
                temp_z_buffer[x * WindowSizeW + y] = depth;
            }
        }
    }

    return firstChangeLine;
}
```

使用 Interpolate函数计算每个像素的法线和位置，然后根据这些值计算光照颜色：

```
std::pair<vec3, vec3> Interpolate(int x, int y, FragmentAttr vertices[3]) {
    vec3 p = vec3(x, y, 0); // 将输入的x和y坐标转换为vec3

    // 计算重心坐标
    vec3 v0 = vec3(vertices[1].x - vertices[0].x, vertices[1].y - vertices[0].y,
vertices[1].z - vertices[0].z);
    vec3 v1 = vec3(vertices[2].x - vertices[0].x, vertices[2].y - vertices[0].y,
vertices[2].z - vertices[0].z);
    vec3 v2 = vec3(p.x - vertices[0].x, p.y - vertices[0].y, 0); // 假设z值为0，因
为x和y是屏幕坐标

    float d00 = dot(v0, v0);
    float d01 = dot(v0, v1);
    float d11 = dot(v1, v1);
    float d20 = dot(v2, v0);
    float d21 = dot(v2, v1);
    float denom = d00 * d11 - d01 * d01;

    vec3 bary;
    bary.y = (d11 * d20 - d01 * d21) / denom;
    bary.z = (d00 * d21 - d01 * d20) / denom;
    bary.x = 1.0f - bary.y - bary.z;

    // 通过重心坐标插值法向量
    vec3 interpolatedNormal = normalize(bary.x * vertices[0].normal + bary.y *
vertices[1].normal + bary.z * vertices[2].normal);
    vec3 interpolatedPosmv = normalize(bary.x * vertices[0].pos_mv + bary.y *
vertices[1].pos_mv + bary.z * vertices[2].pos_mv);
    return std::make_pair(interpolatedNormal, interpolatedPosmv);
}
```

运行结果将在2.4集中显示。

2.3：用 Blinn-Phong 实现三角形内部的着色

Blinn-Phong 着色是 Phong 着色的改进版，它简化了镜面反射的计算，使用半角向量（即光源方向和观察方向的中间向量）来计算高光。

原理：

与 Phong 着色类似，Blinn-Phong 着色也在每个像素处计算光照。
漫反射部分与 Phong 着色相同。
镜面反射使用半角向量来计算，这简化了计算并且可以更好地模拟金属等材质的高光效果。
半角向量的方向取决于光源和观察者的位置，其大小取决于表面法线的光泽度。

```
vec3 MyGLWidget::BlinnPhongShading(FragmentAttr& fragment) {
    vec3 ambientColor = vec3(0.1, 0.1, 0.1);  // 环境光颜色
    vec3 lightColor = vec3(1.0, 1.0, 1.0);    // 光源颜色
    vec3 materialColor = vec3(1.0, 1.0, 1.0); // 材料颜色

    // 环境光部分
    vec3 ambient = ambientColor * materialColor;

    // 漫反射部分
    vec3 norm = normalize(fragment.normal);
    vec3 lightDir = normalize(lightPosition - vec3(fragment.pos_mv));
    float diff = max(dot(norm, lightDir), 0.0f);
    vec3 diffuse = diff * lightColor * materialColor;

    // 镜面反射部分
    vec3 viewDir = normalize(-vec3(fragment.pos_mv)); // 观察方向
    vec3 halfDir = normalize(lightDir + viewDir); // 半角向量
    float spec = pow(max(dot(norm, halfDir), 0.0f), 32); // 镜面反射强度
    vec3 specular = spec * lightColor;

    // 总光照强度
    vec3 intensity = ambient + diffuse + specular;

    return intensity;
}
```

edge_walking：

与 Phong 着色类似，Blinn-Phong 着色中的 edge_walking 函数需要计算每个像素的法线向量，并在每个像素处计算光照。因此，edge_walking 函数需要访问顶点的法线向量，并在三角形内部进行插值，以计算每个像素的法线向量和相应的光照颜色。

```
            case 6:{
                FragmentAttr tmp = FragmentAttr(x, y, 0, 0);
                std::pair<vec3, vec3> result = Interpolate(x, y,
transformedVertices);
                tmp.normal = result.first;
                tmp.pos_mv = result.second;
                color = BlinnPhongShading(tmp);
                break;
            }
            //别的与Phong方法中的edge_walking相同
```

运行结果将在2.4集中显示。

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。

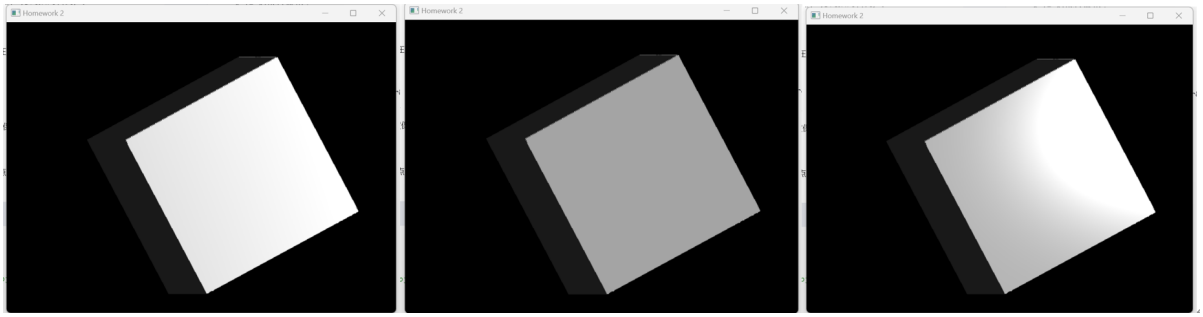
按键4、5、6分别表示Gouraud 、Phong、Blinn-Phong：

```
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_2: draw_id = 2; std::cout << "DDA:" << std::endl; update();
break;
    case Qt::Key_3: draw_id = 3; std::cout << "bresenham:" << std::endl;
update(); break;
    case Qt::Key_4: draw_id = 4; std::cout << "Gouraud:" << std::endl; update();
break;
    case Qt::Key_5: draw_id = 5; std::cout << "Phong:" << std::endl; update();
break;
    case Qt::Key_6: draw_id = 6; std::cout << "Blinn-Phong:" << std::endl;
update(); break;
    case Qt::Key_9: degree += 35; update(); break;
    }
}
```

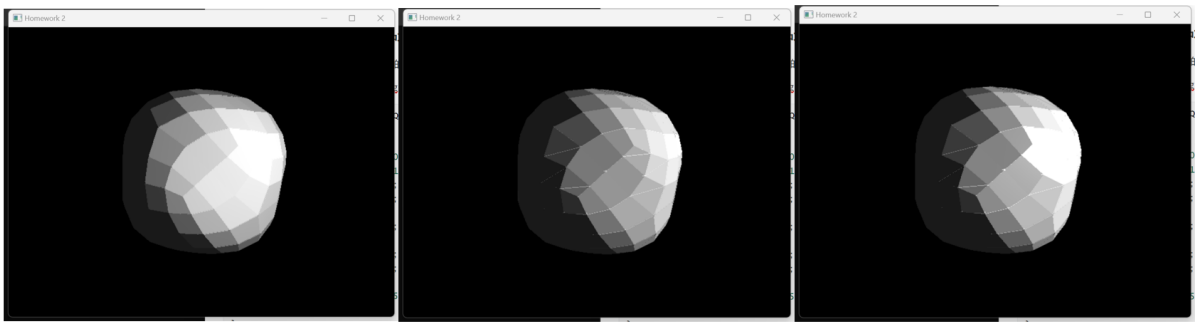
效果图：

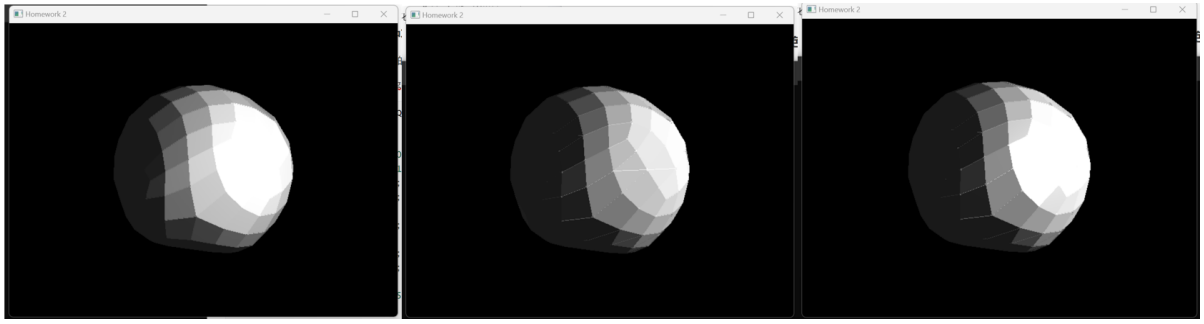
调整光照参数

1. 三张的从左到右分别为Gouraud 、Phong、Blinn-Phong。



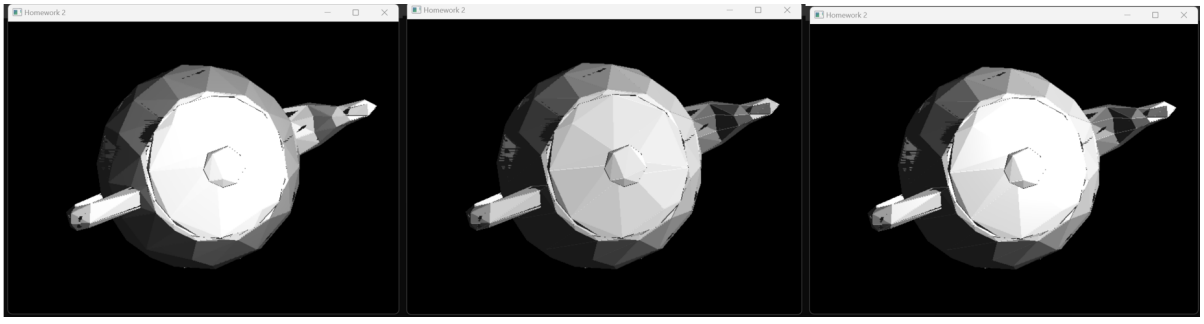
2. 更改相机位置：

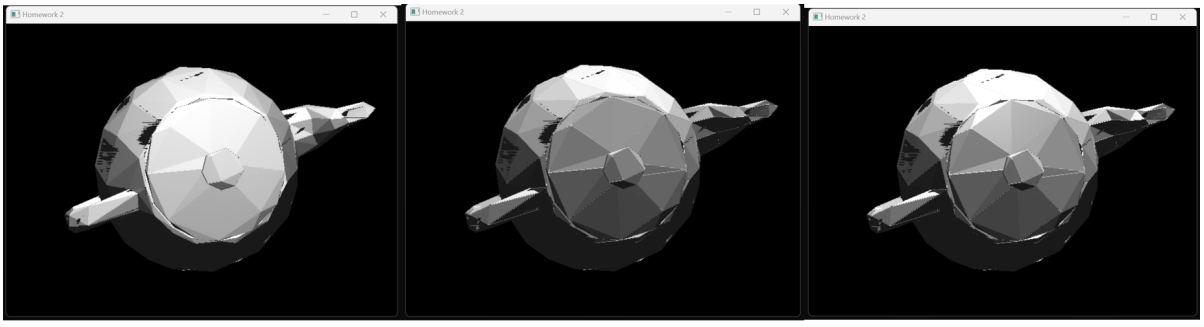




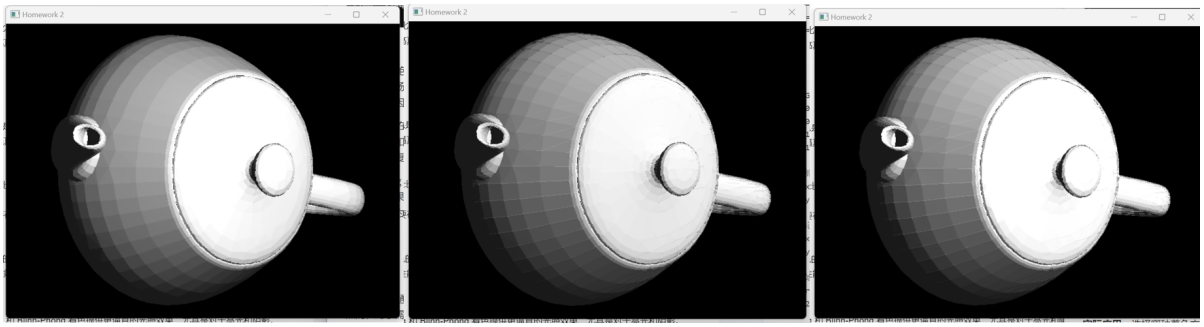
3. 更改光照为：

```
lightPosition = objModel.centralPoint + vec3(100, 0, 20);
```

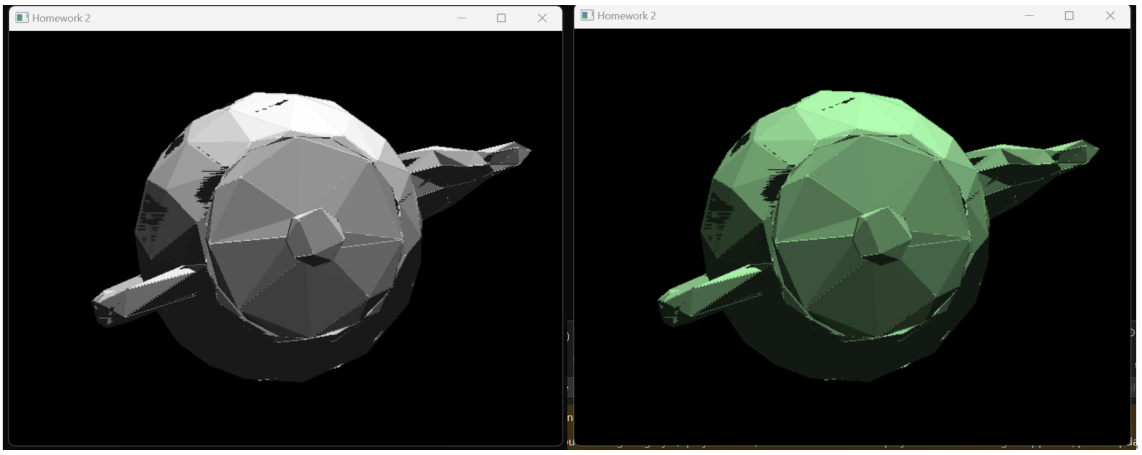




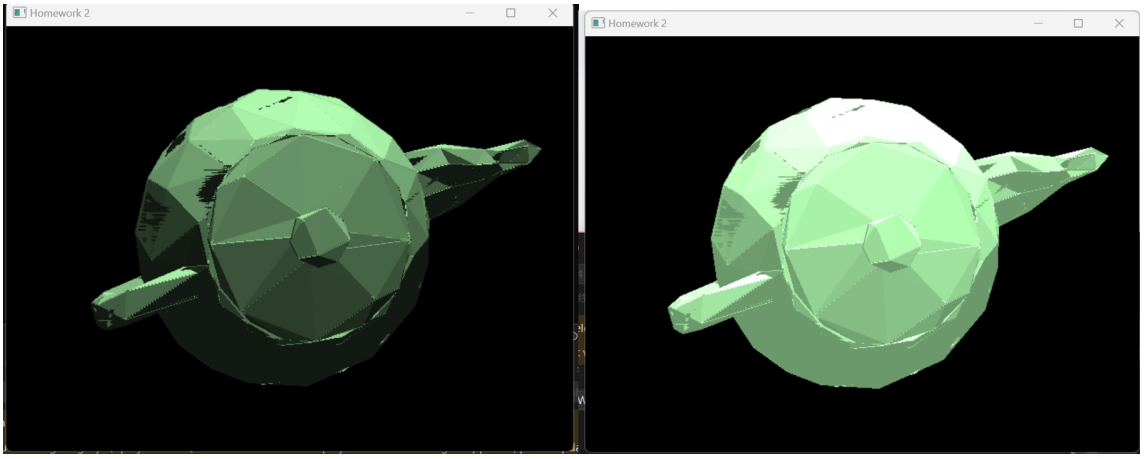
4.



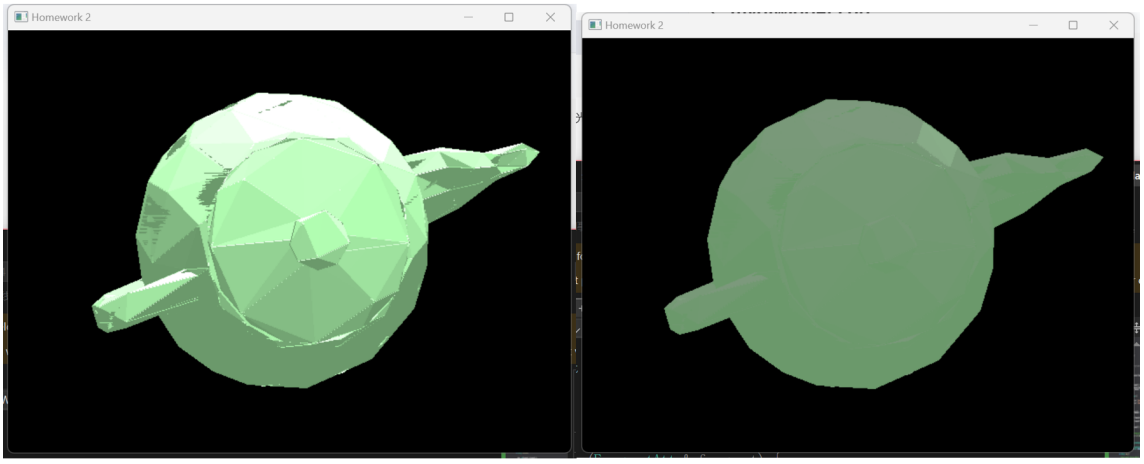
5. Phong更改光源颜色



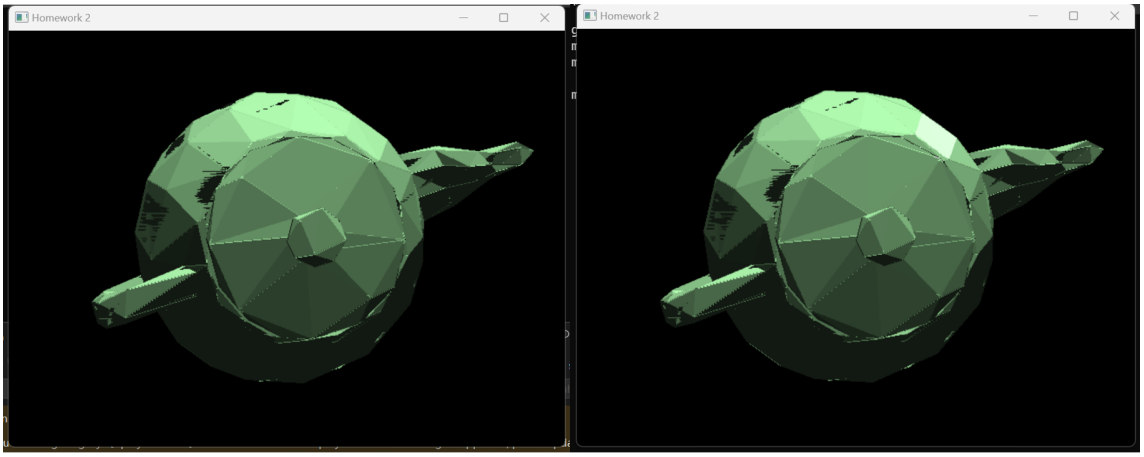
6. Phong更改环境光（0.1，0.1，0.1）为（0.6，0.6，0.6）



7. Phong更改漫反射（0.9，0.9，0.9）为（0.1，0，0.1）

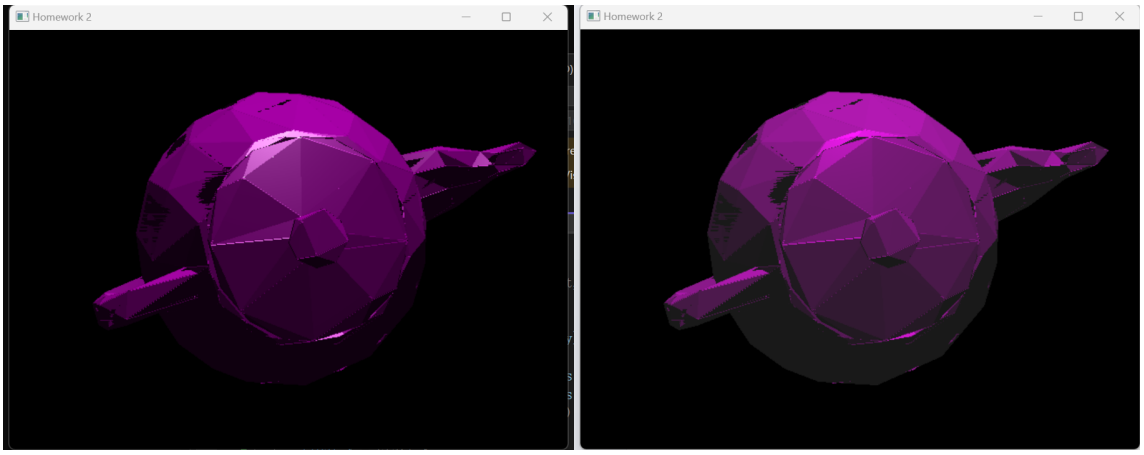


8. Phong更改镜面反射颜色（0.2，0.2，0.2）为（1.0，1.0，1.0）



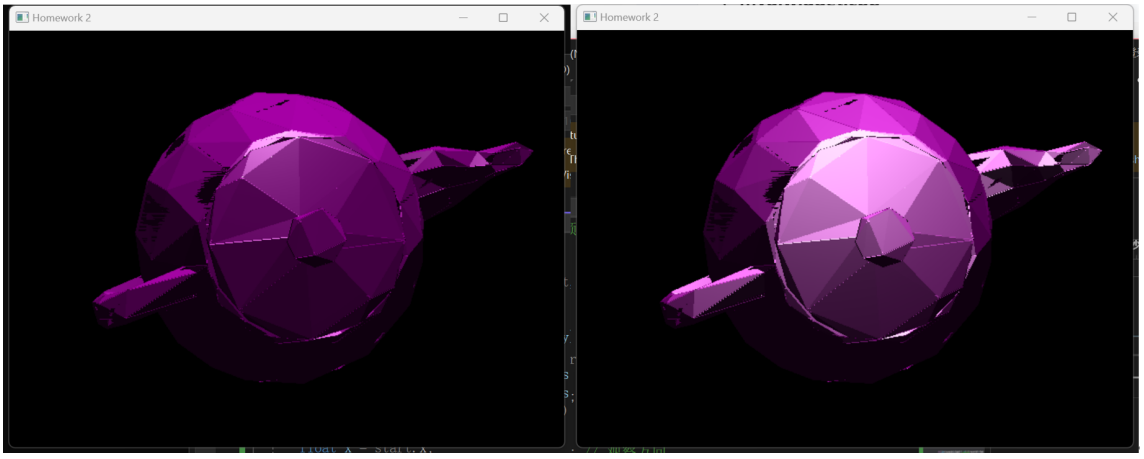
9. 第一张改变BlinnPhong材料颜色 （0.6，0，0.6），光源颜色为（1.0，1.0，1.0）

   第二张改变光源颜色为（0.6，0，0.6），材料颜色为（1.0，1.0，1.0）



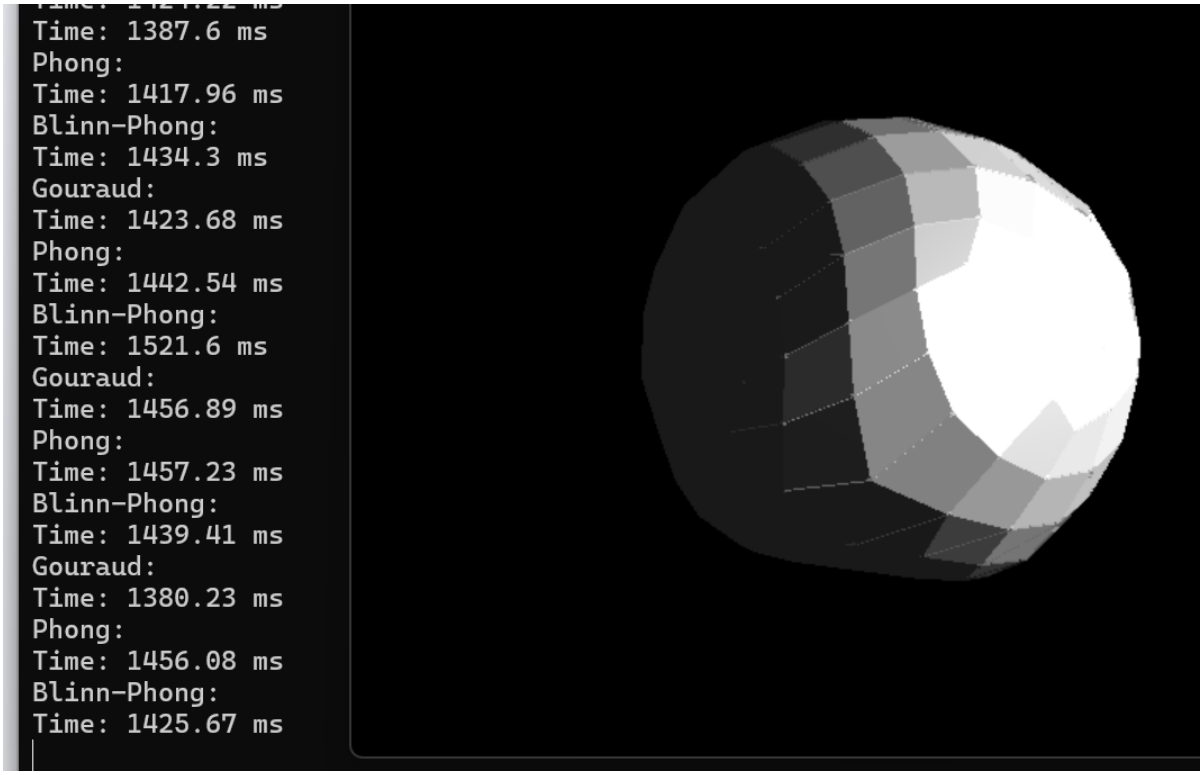
10. 改变镜面反射强度32为6：

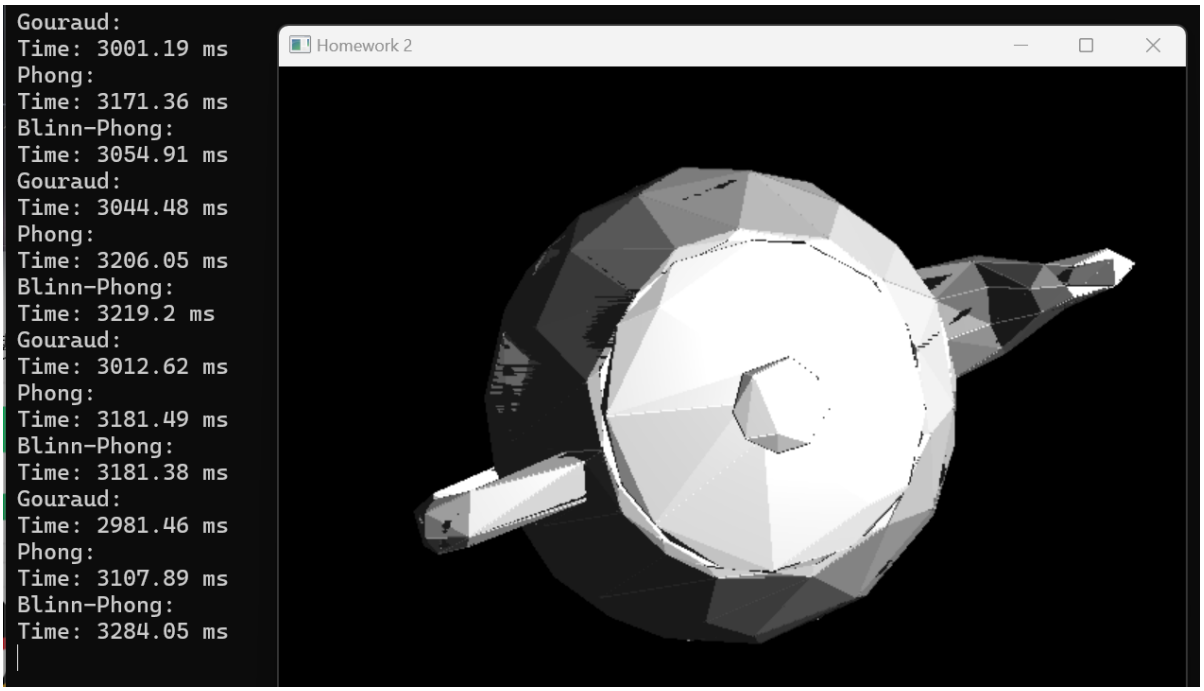
```
float spec = pow(max(dot(norm, halfDir), 0.0f), 6); // 镜面反射强度
```



着色效率：

我们使用300面的石块和600面的茶壶进行对比。实践对比如下：





Gouraud 着色由于只需要在顶点处计算光照，然后进行插值，所以计算量较小，适合实时渲染，适用于需要快速渲染的场景，并且可以产生相对平滑的渐变效果，特别是在光照变化不大的情况下。所以 Gouraud 着色相对高效，因为它避免了在每个像素上进行复杂的光照计算。

Phong 着色在每个像素上计算光照，包括环境光、漫反射和镜面反射，可以产生非常逼真的光照效果，特别是对于高光和阴影的渲染。因此Phong 着色比 Gouraud 着色更计算密集，这导致计算量显著增加，尤其是在高分辨率和复杂场景中。

Blinn-Phong 着色是 Phong 着色的改进版，使用半角向量来计算镜面反射，简化了计算。它同样可以产生逼真的光照效果，特别是在模拟金属或光泽表面时。与 Phong 着色相比，Blinn-Phong 着色在计算镜面反射时更高效，因为它避免了反射向量的计算。然而，它仍然需要在每个像素上进行光照计算，因此计算量仍然较大。

总结

视觉效果：Phong 和 Blinn-Phong 着色提供更逼真的光照效果，尤其是对于高光和阴影。Gouraud 着色则提供了平滑的颜色过渡。
渲染效率：Gouraud 着色最高效，适合实时渲染和大型场景。Phong 和 Blinn-Phong 着色虽然视觉效果更好，但计算成本更高。
实际应用：选择哪种着色方法取决于具体需求。对于需要快速渲染的应用，Gouraud 着色可能是最佳选择。对于需要高真实感视觉效果的应用，Phong 或 Blinn-Phong 着色可能更合适。