计算机图形学作业二

21307077 凌国明

1. 三角形光栅化算法

1.1 DDA. DDA 算法用于栅格化线段，其核心原理是从线段的一端到另一端进行迭代，计算每一步的坐标值。算法首先计算线段在 X 和 Y 方向的差值 dx 和 dy，然后基于这两者中的较大者来确定总的迭代步数（steps）。在每一步中，算法会沿线段均匀地增加 X 和 Y 坐标，这些增量由 dx 和 dy 除以 steps 计算得出。这样，DDA 算法可以生成线段上均匀分布的点序列，从而绘制出近似于原线段的栅格化版本。

```
void MyGLWidget::DDA(FragmentAttr& start, FragmentAttr& end
    , int id) {
    int dx = end.x - start.x;
    int dy = end.y - start.y;
    int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy); //
        步骤数基于最大的dx或dy

    float xIncrement = dx / (float)steps;
    float yIncrement = dy / (float)steps;

    float x = start.x;
    float y = start.y;

    for (int i = 0; i <= steps; i++) {
        temp_render_buffer[int(x) * WindowSizeW + int(y)] =
            vec3(255, 255, 255); // 绘制像素
        temp_z_buffer[int(x) * WindowSizeW + int(y)] =
            start.z + (end.z - start.z) * (i / (float)
            steps); // 计算并更新深度信息
        x += xIncrement;
        y += yIncrement;
    }
}
```

Listing 1. DDA

1.2 bresenham. Bresenham 算法是一种有效的栅格化直线绘制方法，特别适用于计算资源受限的环境。这个算法的关键在于它仅使用整数运算，避免了浮点运算的复杂性。算法开始时计算两点间的水平和垂直距离（dx 和 dy），并根据这些值初始化一个误差变量（error）。在每一步迭代中，算法更新当前点的坐标，根据误差变量来判断下一步是仅在水平方向（X 轴）移动，还是同时在水平和垂直方向（X 轴和 Y 轴）移动。这种方法的优点在于，所有计算都是基于整数的，减少了计算量，同时保证了线段的连续性和均匀性。

```
void MyGLWidget::bresenham(FragmentAttr& start,
     FragmentAttr& end, int id) {
    int dx = abs(end.x - start.x);
    int dy = -abs(end.y - start.y);
    int sx = start.x < end.x ? 1 : -1;
    int sy = start.y < end.y ? 1 : -1;
    int error = dx + dy;  // error值是两倍的真实error
    int e2;  // error的临时变量

    FragmentAttr nowV = start; // 初始点

    // 计算总步数
    int totalSteps = max(abs(end.x - start.x), abs(end.y -
        start.y));
    int step = 0; // 当前步数

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

        // 将像素渲染到temp_render_buffer, 并更新
            temp_z_buffer
        temp_render_buffer[nowV.x * WindowSizeW + nowV.y] =
            vec3(255, 255, 255);  // 渲染像素颜色
        temp_z_buffer[nowV.x * WindowSizeW + nowV.y] =
            depth;  // 更新深度信息

        // 检查是否到达终点
        if (nowV.x == end.x && nowV.y == end.y) break;

        e2 = 2 * error;
        if (e2 >= dy) {  // e_xy + e_x > 0
            if (nowV.x == end.x) break;
            error += dy;
            nowV.x += sx;
        }
        if (e2 <= dx) {  // e_xy + e_y < 0
            if (nowV.y == end.y) break;
            error += dx;
            nowV.y += sy;
        }

        step++; // 增加步数
    }
}
```

Listing 2. bresenham

DDA 算法使用浮点运算，以性能为代价实现更精确的线段绘制。Bresenham 算法只使用整数运算，效率更高，尤其适合硬件实现。但它可能在处理非水平或垂直线段时，不如 DDA 算法平滑。

DDA 算法实现相对简单直接，但需要处理浮点数和四舍五入到最近的整数，可能在某些情况下产生累计误差。Bresenham 算法虽然避免了浮点运算，但其实现需要仔细处理误差累计和步进逻辑，尤其是在处理较长的线段时。

DDA 算法更适合需要高精度线段绘制的应用，如在高分辨率显示器上绘制复杂图形。Bresenham 算法更适用于性能受限的环境。

1.3 edge_walking. Edge_Walking 算法是一种用于三角形光栅化的技术，它通过遍历屏幕的每一行像素来确定三角形的边界。在每一行中，算法搜索从三角形外部到内部的过渡点（即左边界），然后再寻找从内部到外部的过渡点（即右边界）。这些边界点通过比较相邻像素的颜色来确定，一旦找到，算法便在这两个边界之间填充像素，渲染出三角形的一部分。此过程涉及到对每个像素的颜色和深度值进行适当的插值。

```
int MyGLWidget::edge_walking() {
    int firstchangeline = WindowSizeH;

    // 遍历每一行
    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;
              // 存储边界点的深度值

        // 遍历每一列
        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+1;
                foundstart = true;
                startDepth = temp_z_buffer[x * WindowSizeW
                      + y]; // 获取起始边界的深度值
            }
            // 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-1;
                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
            }
        }

        // 如果找到了三角形的边界
        if (foundstart) {
            firstchangeline = firstchangeline < x ?
                  firstchangeline : x;

            for (int y = start; y <= end; y++) {
                // 计算插值深度
                float depth= startDepth + (endDepth -
                      startDepth) * ((y - start) / (end -
                      start + 0.1));
                temp_z_buffer[x * WindowSizeW + y] = depth;
                // 填充颜色
                temp_render_buffer[x * WindowSizeW + y] =
                      vec3(0, 255, 0);
            }
        }
    }
    return firstchangeline;
}
```

Listing 3. edge_walking

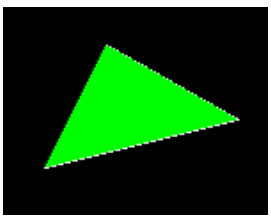

图 1. bresenham + ew

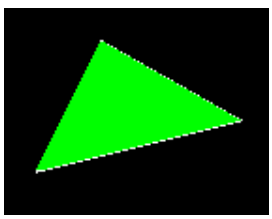

图 2. DDA + ew

2. 光照着色算法

2.1 Gouraud. 在 Bresenham 算法中结合 Gouraud 着色的应用涉及在确定线段每个像素点时同时进行颜色的插值。这里的关键是首先计算线段两端点的颜色，使用 GauraudShading 函数得到 start_color 和 end_color。然后，在 Bresenham 算法的迭代过程中，对于线段上的每个像素点，根据其在线段上的位置来插值计算这个点的颜色。这种颜色插值是基于线段起点和终点的颜色以及像素在线段上的相对位置进行的。通过这种方法，线段上的颜色变化可以平滑过渡，从而在视觉上实现更柔和的渲染效果。

```
void MyGLWidget::bresenham2(FragmentAttr& start,
      FragmentAttr& end, int id) {
    int dx = abs(end.x - start.x);
    int dy = -abs(end.y - start.y);
    int sx = start.x < end.x ? 1 : -1;
    int sy = start.y < end.y ? 1 : -1;
    int error = dx + dy;  // error值是两倍的真实error
    int e2;  // error的临时变量

    vec3 start_color = GauraudShading(start);
    vec3 end_color = GauraudShading(end);
    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;

    FragmentAttr nowV = start; // 初始点

    // 计算总步数
    int totalSteps = max(abs(end.x - start.x), abs(end.y -
          start.y));
    int step = 0; // 当前步数

    while (true) {
        // 计算插值深度
        float depth = start.z + (end.z - start.z) * (step /
              (float)totalSteps);
        vec3 color = start_color + (end_color - start_color
              ) * (step / (float)totalSteps);
        if (color == vec3(0, 0, 0)) {
            color = vec3(0, 1, 0);
        }

        // 将像素渲染到temp_render_buffer，并更新
              temp_z_buffer
        temp_render_buffer[nowV.x * WindowSizeW + nowV.y] =
              color;  // 渲染像素颜色
        temp_z_buffer[nowV.x * WindowSizeW + nowV.y] =
              depth;  // 更新深度信息

        // 检查是否到达终点
        if (nowV.x == end.x && nowV.y == end.y) break;

        e2 = 2 * error;
        if (e2 >= dy) {  // e_xy + e_x > 0
            if (nowV.x == end.x) break;
            error += dy;
            nowV.x += sx;
        }
        if (e2 <= dx) {  // e_xy + e_y < 0
            if (nowV.y == end.y) break;
            error += dx;
            nowV.y += sy;
        }

        step++; // 增加步数
    }
}
```

Listing 4. bresenham2

在 Edge Walking 算法中应用 Gouraud 着色的核心是对三角形内部每个像素点的颜色进行插值。该过程首先涉及对三角形每行的边界点颜色的确定，然后在这两点之间对每个像素点的颜色进行插值计

算。在算法中，对于每一行，一旦确定了三角形的左右边界以及这些边界点的颜色（startColor 和 endColor），就会对这两个颜色之间的每个像素点进行线性插值，以计算其颜色值。这样，三角形内的颜色渐变能够根据三角形边界的颜色变化自然过渡，从而在渲染时产生更加平滑和真实的视觉效果。

```
int MyGLWidget::edge_walking2() {
    int firstchangeline = WindowSizeH;

    // 遍历每一行
    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;
              // 存储边界点的深度值
        vec3 startColor, endColor;

        // 遍历每一列
        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]; // 获取起始边界的深度值
                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]; // 获取结束边界的深度值
                endColor = temp_render_buffer[x *
                      WindowSizeW + y]; // 获取结束边界的颜
                      色
                break; // 找到一行中的起始和结束边界
            }
        }

        // 如果找到了三角形的边界
        if (foundstart) {
            firstchangeline = firstchangeline < x ?
                  firstchangeline : x;

            for (int y = start; y <= end; y++) {
                // 计算插值颜色和深度
                float t = (float)(y - start) / (float)(end
                      - start + 0.01f);
                vec3 color = vec3(startColor.x + (endColor.
                      x - startColor.x) * t,
                                  startColor.y + (
                                        endColor.y -
                                        startColor.y) * t
                                        ,
                                  startColor.z + (
                                        endColor.z -
                                        startColor.z) * t
                                        );
                float depth = startDepth + (endDepth -
                      startDepth) * (y - start) / (end -
                      start + 0.01f);
                temp_z_buffer[x * WindowSizeW + y] = depth;
                // 填充颜色
                temp_render_buffer[x * WindowSizeW + y] =
                      color;
            }
        }
    }

    return firstchangeline;
}
```

Listing 5. edge_walking2

遇到一个问题，getTriangleByID 中，Triangle 的 triangleVertices 和 triangleNormals 都是指针，然后在 getTriangleByID 中指针指向局部变量，然后离开函数后，局部变量消亡，指针变为野指针。

修改 Triangle 中的指针为长度为 3 的数组

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

Listing 6. Triangle

修改 getTriangleByID，直接对 Triangle 赋值

```
Triangle getTriangleByID(int id) {
    assert(id < triangleCount);
    int* nowTirVerIDs = triangles[id];
    int* nowTriNormIDs = triangle_normals[id];

    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;
}
```

Listing 7. getTriangleByID

Gouraud 着色的原理基于在三维图形的顶点上计算光照效果，然后在图形的表面上插值这些光照值。对每个顶点，环境光和漫反射的强度首先被计算出来。环境光是一个常量，通常与材料颜色简单相乘（ambient = ambientColor × materialColor）。漫反射光则基于光源方向和顶点法线的点积计算（diffuse = max(dot(norm, lightDir),0.0) × lightColor × materialColor）。总光照强度是这两种光照的组合（intensity = ambient + diffuse）。在图形的其它部分，顶点的光照强度会根据顶点间的相对位置进行线性插值，从而在整个图形上产生平滑的颜色变化。这种方法的优点在于其计算效率较高，特别是在硬件加速的情况下，但可能会在高光效果的渲染上不够精确，尤其是在大面积的平面上。

```
vec3 MyGLWidget::GauraudShading(FragmentAttr&
      nowPixelResult) {
    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(nowPixelResult.normal);
    vec3 lightDir = normalize(lightPosition - vec3(
          nowPixelResult.pos_mv));
    float diff = dot(norm, lightDir);
    if (diff < 0) {
        diff = 0.0f;
    }
    vec3 diffuse = diff * lightColor * materialColor;

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

    return intensity;
}
```

Listing 8. GauraudShading

2.2 Phong. Phong 光照模型是一种广泛使用的模型，它考虑了环境光、漫反射和镜面反射三个组成部分。环境光部分由环境光颜色和材料颜色的乘积构成，表示了无方向的、普遍存在的光照（ambient = ambientColor × materialColor）。漫反射光则是光源方向和顶点法线的点积与光源颜色和材料颜色的乘积，反映了光源方向对光照强度的影响（diffuse = max(dot(norm, lightDir),0.0) × lightColor × materialColor）。镜面反射光是观察方向和反射方向的点积的高次幂与光源颜色和材料颜色的乘积，表示了光源在物体表面的镜面反射效果（specular = pow(max(dot($viewDir, reflectDir$),0.0),shininess)× lightColor × materialColor）。最终的光照强度是这三个部分的总和。

```
vec3 MyGLWidget::PhoneShading(FragmentAttr&
      nowPixelResult) {
    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(nowPixelResult.normal);
    vec3 lightDir = normalize(lightPosition - vec3(
        nowPixelResult.pos_mv));
    float diff = dot(norm, lightDir);
    if (diff < 0) {
        diff = 0.0f;
    }
    vec3 diffuse = diff * lightColor * materialColor;

    vec3 viewDir = normalize(-vec3(nowPixelResult.
        pos_mv));
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0f
        ), 64); // 32为镜面反射的光泽度
    vec3 specular = spec * lightColor * materialColor;

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

    return intensity;
}
```

Listing 9. PhoneShading

在 Edge Walking 算法中，Phong 光照模型被用于计算三角形内部每个像素点的颜色。这一过程首先包括对每个像素点的法向量和模型视图位置进行插值计算。之后，每个像素点都使用 Phong 光照模型独立计算光照效果，包括环境光、漫反射和镜面反射。这样，每个像素点的颜色都会根据其位置和方向相对于光源和观察者的变化而变化，从而在三角形内部实现平滑和逼真的光照效果。

```
for (int y = start; y <= end; y++) {
    // 计算插值颜色和深度
    FragmentAttr tmp = FragmentAttr(x, y, 0, 0);
    tmp.normal = InterpolateNormal(x, y,
        transformedVertices);
    tmp.pos_mv = InterpolatePosmv(x, y, transformedVertices
        );
    vec3 color = PhoneShading(tmp);
    float depth = startDepth + (endDepth - startDepth) * (y
        - start) / (end - start + 0.01f);
    temp_z_buffer[x * WindowSizeW + y] = depth;
    // 填充颜色
    temp_render_buffer[x * WindowSizeW + y] = color;
}
```

Listing 10. edge_walking

在三角形光栅化过程中，通过重心坐标来插值计算法向量是一种常用的技术。这个方法首先根据三角形三个顶点的屏幕坐标计算目标像素点的重心坐标。计算公式是基于向量的点乘和行列式来确定每个顶点对目标像素点的影响程度（bary = $\frac{d11\times d20-d01\times d21}{denom}, \frac{d00\times d21-d01\times d20}{denom}, 1.0 - bary.y - bary.z$）。接着，使用这些重心坐标值，可以通过线性插值的方式从三个顶点的法向量中计算出目标像素点的法向量（interpolatedNormal = normalize(bary.x × vertices[0].normal + bary.y × vertices[1].normal + bary.z × vertices[2].normal)）。这种方法能够确保三角形内部的法向量变化平滑，从而在渲染时提供更自然的光照效果。

```
vec3 InterpolateNormal(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);
    return interpolatedNormal;
}
```

Listing 11. InterpolateNormal

同样地，设置一个函数对像素点的 posmv 进行插值

```
vec3 InterpolatePosmv(int x, int y, FragmentAttr vertices
    [3])
    // 省略，逻辑与上面一样
```

Listing 12. InterpolatePosmv

2.3 Blinn-Phong. Blinn-Phong 着色模型是 Phong 模型的一种改进，它同样包括环境光、漫反射和镜面反射三个部分。环境光部分由环境光颜色和材料颜色相乘得到（ambient = ambientColor × materialColor），为场景提供基础照明。漫反射光根据光源方向与法线的点积计算，反映了表面对光源的散射（diffuse = max(dot($norm, lightDir$),0.0) × lightColor × materialColor）。镜面反射光的核心是半程向量（halfDir），它是光源方向和观察方向的平均向量，这一部分根据法线与半程向量的点积的高次幂以及光源颜色计算（specular = pow(max(dot($norm, halfDir$),0.0),shininess) × lightColor）。最终的光照强度是这三部分的组合，可以为物体表面提供更自然和平滑的光照效果。

```
vec3 MyGLWidget::BlinnPhongShading(FragmentAttr&
    nowPixelResult) {
    vec3 ambientColor = vec3(0.1, 0.1, 0.1);
    vec3 lightColor = vec3(1.0, 1.0, 1.0);
    vec3 materialColor = vec3(1.0, 1.0, 1.0);
    vec3 norm = normalize(nowPixelResult.normal);
    vec3 lightDir = normalize(lightPosition - vec3(
        nowPixelResult.pos_mv));
    vec3 viewDir = normalize(-vec3(nowPixelResult.pos_mv));
    vec3 halfDir = normalize(lightDir + viewDir);
    float diff = max(dot(norm, lightDir), 0.0f);
    float spec = pow(max(dot(norm, halfDir), 0.0f), 32);

    vec3 ambient = ambientColor * materialColor;
    vec3 diffuse = diff * lightColor * materialColor;
    vec3 specular = spec * lightColor;

    return ambient + diffuse + specular;
}
```

Listing 13. BlinnPhongShading

Blinn-Phong 模型与传统的 Phong 模型的主要区别在于镜面反射部分的处理。在 Phong 模型中，镜面反射是基于反射向量和观察向量的点积计算的，这要求计算反射向量，这在计算上可能较为复杂。相比之下，Blinn-Phong 模型使用半程向量，即光源方向和观察方向的平均值，以简化计算。这个变化使得 Blinn-Phong 模型在镜面反射的计算上更为高效和实际，尤其是在处理大量光源的情况下。此外，Blinn-Phong 模型通常能够提供更平滑和自然的视觉效果，特别是在处理硬面和高光反射的场景时。

然后在 Edge Walking 中修改每个像素点颜色的计算方法为 BlinnPhongShading 即可。

```
// 计算颜色和深度
FragmentAttr tmp = FragmentAttr(x, y, 0, 0);
tmp.normal = InterpolateNormal(x, y, transformedVertices);
tmp.pos_mv = InterpolatePosmv(x, y, transformedVertices);
vec3 color = BlinnPhongShading(tmp);
```

Listing 14. edge_walking

3. 效果展示

3.1 DDA vs bresenham. 第一题的 DDA 和 bresenham 的效果分别如下

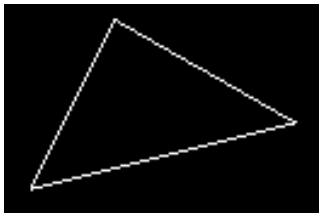

图 3. DDA

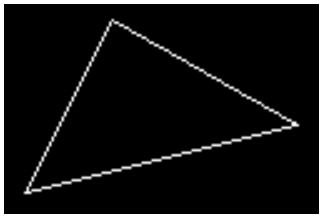

图 4. bresenham

3.2 DDA vs bresenham. 第一题的 DDA 和 bresenham 加上 Edge Walking 的效果分别如下

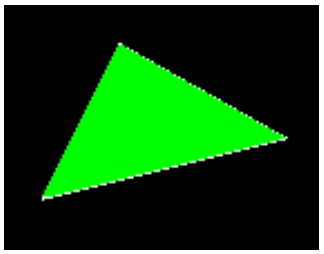

图 5. DDA + ew

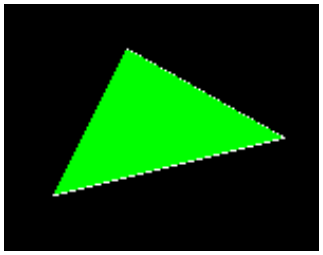

图 6. bresenham + ew

3.3 bresenham and ew. 第一题的 bresenham 加上 Edge Walking 的效果如下

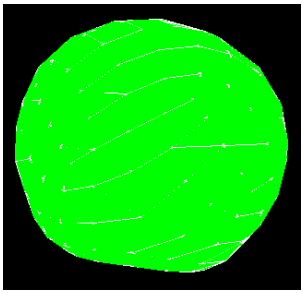

图 7. bresenham + ew

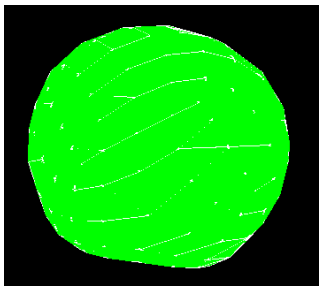

图 8. bresenham + ew

3.4 bresenham and Gauraud. 第二题的 bresenham 加上 GauraudShading 的效果如下

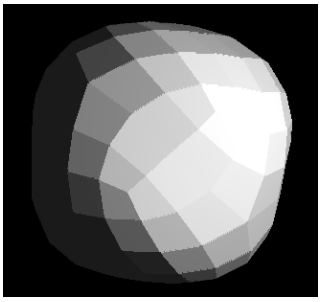

图 9. bresenham + ew

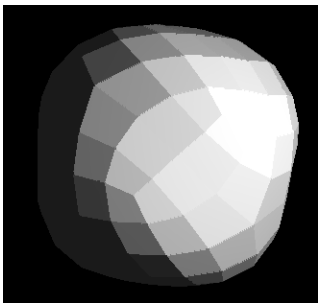

图 10. bresenham + ew

3.5 bresenham and Phong. 第二题的 bresenham 加上 PhongShading 的效果如下

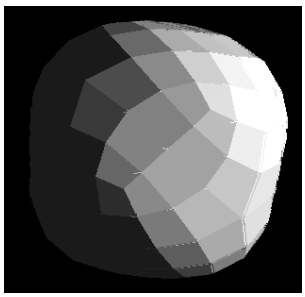

图 11. bresenham + Phong

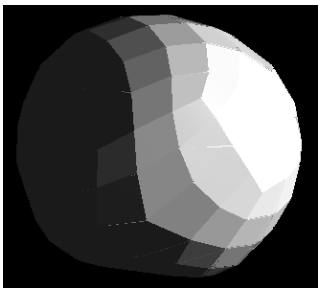

图 12. bresenham + Phong

3.6 bresenham and BlinnPhong. 第二题的 bresenham 加上 BlinnPhongShading 的效果如下

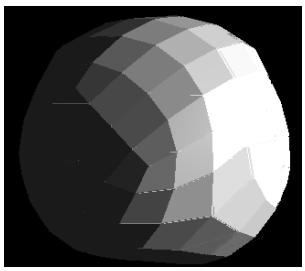

图 13. BlinnPhong

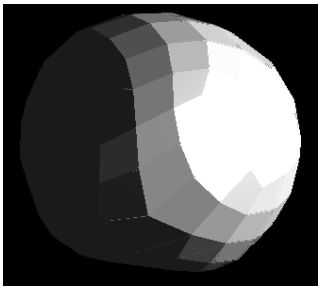

图 14. BlinnPhong

4. 总结

在深入理解和实现不同的光照模型，如 Gouraud 着色、Phong 着色和 Blinn-Phong 着色，以及它们在计算机图形学中的应用之后，我对 3D 渲染和光照处理有了更深入的了解。这些模型各有特点，有效地展示了如何通过计算机算法来模拟现实世界中的光照效果。

Gouraud 着色通过在三角形的顶点上计算光照，然后在三角形的表面上对这些光照值进行插值，提供了一种高效的着色方法。这种方法虽然计算简单，但在渲染大面积平面或低密度网格时可能出现颜色断层。

Phong 着色模型则在每个像素上独立计算光照，包括环境光、漫反射和镜面反射，从而提供了更精细和逼真的渲染效果。Phong 模型对于镜面高光的处理更为精确，尤其适合渲染光滑表面。

Blinn-Phong 着色模型是 Phong 模型的一个变种，它使用半程向量替代了传统 Phong 模型中的反射向量，使得镜面反射的计算更加高效。这一改进在提供与 Phong 模型相似的视觉效果的同时，减少了计算复杂性，特别适用于处理多光源的场景。

在这些着色方法中,** 深度测试（Z-buffering）** 发挥着至关重要的作用。深度测试是一种解决图形渲染中可见性问题的方法，它通过比较像素的深度值来决定哪个像素应该被绘制到屏幕上。这种方法确保了渲染的正确性，防止了较远物体覆盖较近物体的情况发生。深度测试在现代图形渲染中不可或缺，是实现逼真 3D 图像的关键技术之一。

总的来说，这些光照模型和深度测试技术是计算机图形学领域的基石，它们共同构成了 3D 渲染中真实感图像生成的核心。通过对这些技术的学习和实践，我对 3D 图形的渲染原理有了更全面的理解。