中山大學

本科生实验报告

实验课程	计算机图形学第二次作业
专业名称	计算机科学与技术
学生姓名	李世源
学生学号	22342043
提交时间	2024年12月27日

程序使用

```
cd code
make run
```

详情参考 `Makefile` 文件。

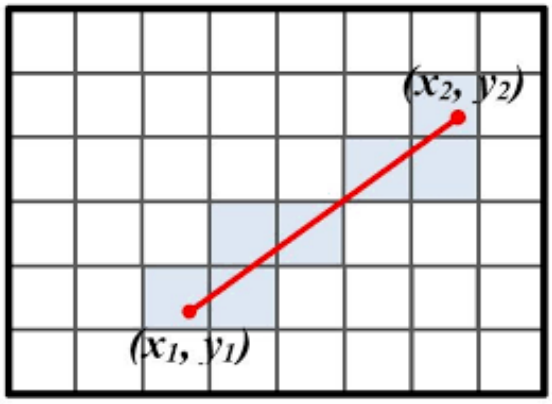
可以通过如下按键进行操作：

```
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_6: obj = (obj + 1) % 5; update(); break;
    case Qt::Key_7: line_mode = (line_mode + 1) % 2; update(); break;
    case Qt::Key_8: shading_mode = (shading_mode + 1) % 4; update(); break;
    case Qt::Key_9: degree += 5; update(); break;
  }
}
```

实验一：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（数字差分分析）算法是一种用于线性插值的图形绘制算法，主要用于计算和绘制直线段。其基本思想是利用两个端点的坐标，通过增量计算逐步生成直线上的所有像素点。

DDA算法的步骤：

1. **计算增量**：根据直线的起点和终点计算x和y方向的增量：
   - $\Delta x = x_2 - x_1$
   - $\Delta y = y_2 - y_1$
   - 选择增量较大的方向作为主要方向。
2. **计算步数**：确定绘制直线所需的步数：
   - (\text{steps} = \max(|\Delta x|, |\Delta y|))
3. **计算每一步的增量**：计算每一步x和y的增量：
   - (x_{\text{inc}} = \Delta x / \text{steps})
   - (y_{\text{inc}} = \Delta y / \text{steps})
4. **逐步绘制**：从起点开始，根据计算出的增量逐步更新x和y的值，并绘制每个计算得到的像素点。

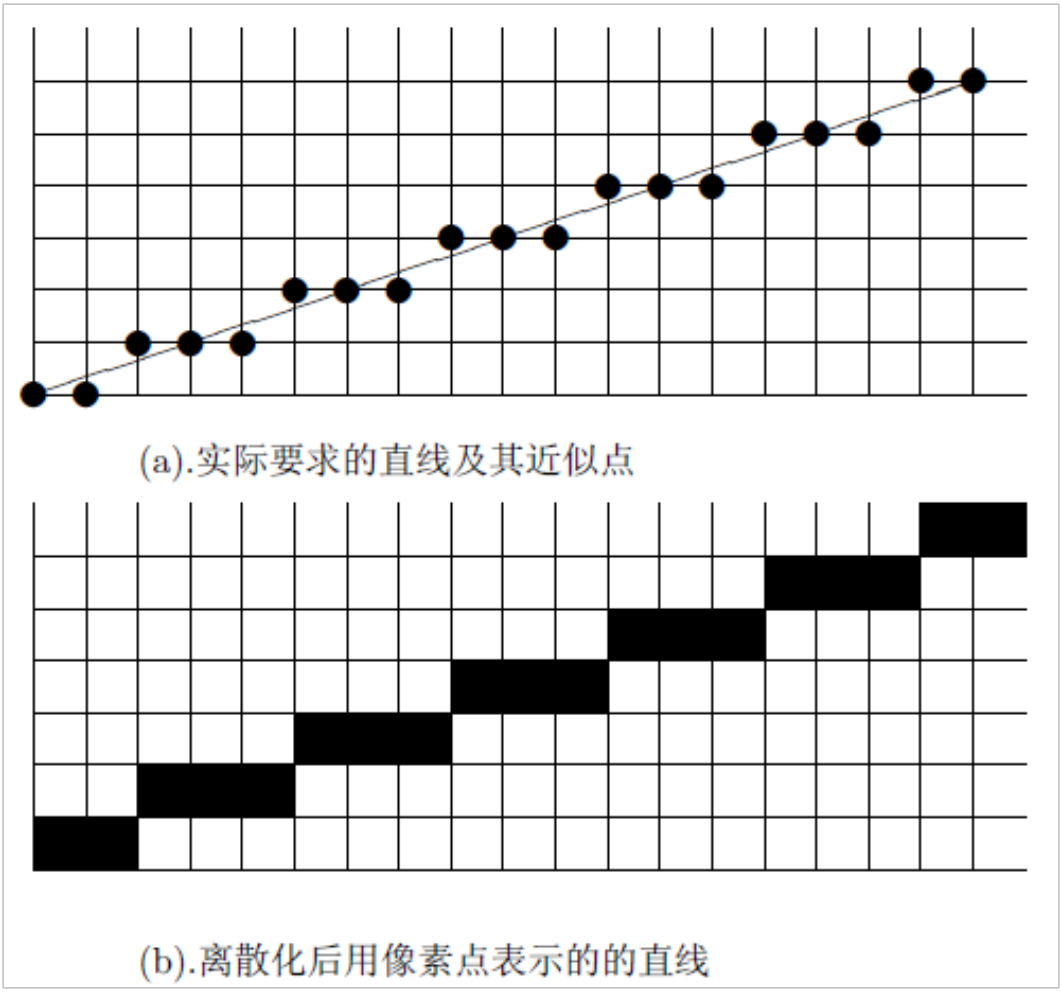
代码实现

```
void MyGLWidget::dda(FragmentAttr& start, FragmentAttr& end) {
  float x_start = start.x, y_start = start.y;
    float x_end = end.x, y_end = end.y;

    float dx = x_end - x_start;
    float dy = y_end - y_start;

    int step_count = static_cast<int>(std::max(abs(dx), abs(dy)));

    float x_step = dx / step_count;
    float y_step = dy / step_count;

    float z_start = start.z, z_end = end.z;
    vec3 color_start = start.color, colorEnd = end.color;

    float z_step = (z_end - z_start) / step_count;
    vec3 color_step = (colorEnd - color_start) / static_cast<float>(step_count);

    float x = x_start;
    float y = y_start;
    float z = z_start;
    vec3 color = color_start;

    for (int step = 0; step <= step_count; step++) {
      int pixel_x = static_cast<int>(round(x));
      int pixel_y = static_cast<int>(round(y));

      if (pixel_x >= 0 && pixel_x < WindowSizeW && pixel_y >= 0 && pixel_y < WindowSizeH)
{
        int index = pixel_y * WindowSizeW + pixel_x;
        if (temp_z_buffer[index] > z) {
          temp_z_buffer[index] = z;
```

```
            temp_render_buffer[index] = color;
          }
        }

        x += x_step;
        y += y_step;
        z += z_step;
        color += color_step;
      }
  }
```

1.2 用 bresenham 实现三角形边的绘制

算法原理



Bresenham算法是一种高效的整数算法，用于在计算机图形学中绘制直线。它通过只使用整数运算，避免了浮点运算，从而提高了绘制效率和精度。

Bresenham算法的步骤：

1. **确定起点和终点**：设定直线的起始点 ((x_1, y_1)) 和结束点 ((x_2, y_2))。
2. **计算增量**：计算x和y方向的增量：

- (\Delta x = x_2 - x_1)
- (\Delta y = y_2 - y_1)

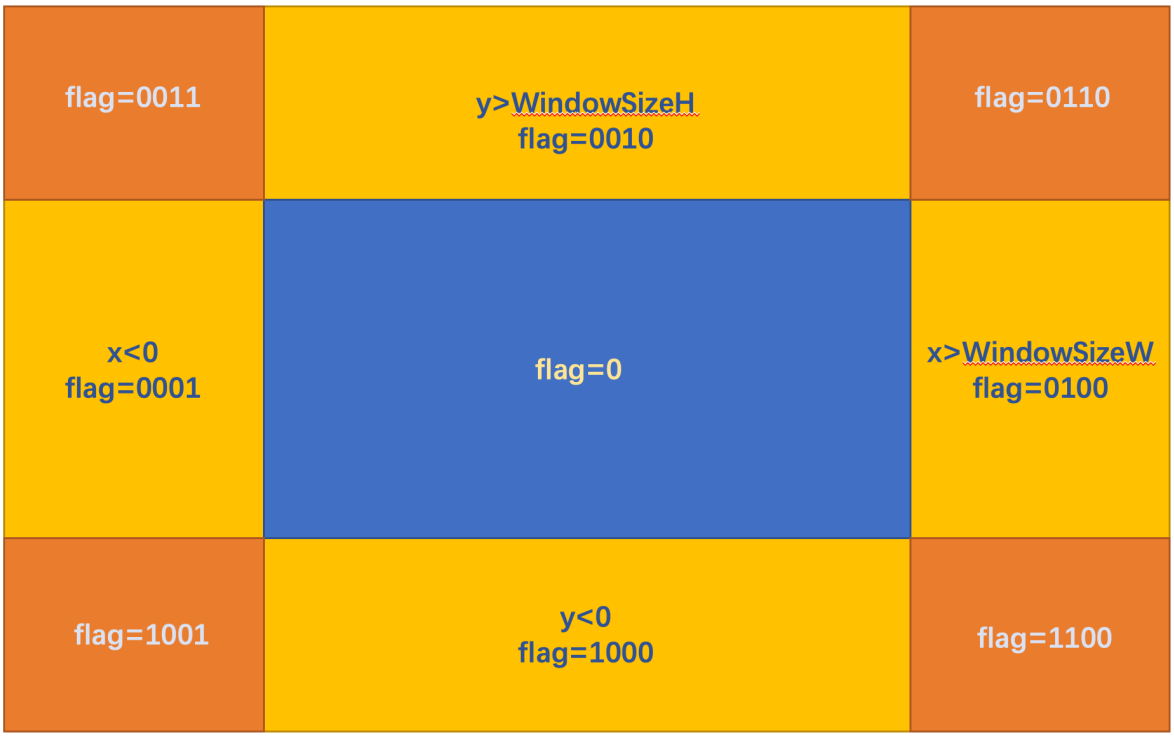
3. **决定主方向**：根据增量的大小确定主方向（x或y），并计算步数。
4. **计算误差**：使用一个误差变量来判断何时增加y的值，从而保持直线的斜率。
5. **逐步绘制**：从起点开始，按照主方向逐个绘制像素点，更新误差变量。

代码实现

```
void MyGLWidget::bresenham(FragmentAttr& start, FragmentAttr& end) {
  int x1 = start.x, y1 = start.y, x2 = end.x, y2 = end.y;
  int dx = abs(x2 - x1);
  int dy = abs(y2 - y1);
  int x_step = x1 < x2 ? 1 : -1;
  int y_step = y1 < y2 ? 1 : -1;
  int err = (dx > dy ? dx : -dy) / 2;

  float dz_half = (end.z - start.z) / 2;
  // z 在 x,y 轴方向上的分量的步长，分量就取一半
  float z_step_x = dz_half / dx;
  float z_step_y = dz_half / dy;
  float z = static_cast<float>(start.z);

  while (x1 != x2 || y1 != y2) {
    temp_render_buffer[y1 * WindowSizeW + x1] = vec3(0.0, 1.0, 0.0);
    temp_z_buffer[y1 * WindowSizeW + x1] = z;
    int err_ = err;
    if (err_ > -dx) { err -= dy; x1 += x_step; }
    if (err_ < dy) { err += dx; y1 += y_step; }
    z += dx > dy ? z_step_x : z_step_y;
  }
}
```

加分项：补充了像素在画布外的处理方式

为处理比较复杂的多种情况，我是用标志位分情况处理。具体划分如下：



详细代码如下：

```
void MyGLWidget::bresenham(FragmentAttr& start, FragmentAttr& end) {
  uint64_t start_pos_flag = 0, end_pos_flag = 0;
  if(start.x < 0) start_pos_flag |= 1;
  if(start.y >= WindowSizeH) start_pos_flag |= 2;
  if(start.x >= WindowSizeW) start_pos_flag |= 4;
  if(start.y < 0) start_pos_flag |= 8;
  if(end.x < 0) end_pos_flag |= 1;
  if(end.y >= WindowSizeH) end_pos_flag |= 2;
  if(end.x >= WindowSizeW) end_pos_flag |= 4;
  if(end.y < 0) end_pos_flag |= 8;
  FragmentAttr a, b;
  if((start_pos_flag & end_pos_flag) != 0) return;
  else if ((start_pos_flag | end_pos_flag) == 1 + 4) {
    a = getLinearInterpolationByX(start, end, 0);
    b = getLinearInterpolationByX(start, end, WindowSizeW);
  } else if ((start_pos_flag | end_pos_flag) == 2 + 8) {
    a = getLinearInterpolationByY(start, end, 0);
    b = getLinearInterpolationByY(start, end, WindowSizeH);
  } else if ((start_pos_flag | end_pos_flag) == 1 + 2) {
    a = getLinearInterpolationByX(start, end, 0);
    b = getLinearInterpolationByY(start, end, WindowSizeH);
    if(a.y > WindowSizeH || b.x < 0) return;
  } else if ((start_pos_flag | end_pos_flag) == 2 + 4) {
    a = getLinearInterpolationByX(start, end, WindowSizeW);
    b = getLinearInterpolationByY(start, end, WindowSizeH);
    if(a.y > WindowSizeH || b.x > WindowSizeW) return;
  } else if ((start_pos_flag | end_pos_flag) == 4 + 8) {
    a = getLinearInterpolationByX(start, end, WindowSizeW);
    b = getLinearInterpolationByY(start, end, 0);
```

```
    if(a.y < 0 || b.x > WindowSizeW) return;
  } else if ((start_pos_flag | end_pos_flag) == 8 + 1) {
    a = getLinearInterpolationByX(start, end, 0);
    b = getLinearInterpolationByY(start, end, 0);
    if(a.y < 0 || b.x < 0) return;
  } else {
    auto f = [&](FragmentAttr a, uint64_t flag){
      if(flag & 1) return getLinearInterpolationByX(start, end, 0);
      if(flag & 2) return getLinearInterpolationByY(start, end, WindowSizeH);
      if(flag & 4) return getLinearInterpolationByX(start, end, WindowSizeW);
      if(flag & 8) return getLinearInterpolationByY(start, end, 0);
      else return a;
    };
    a = f(start, start_pos_flag);
    b = f(end, end_pos_flag);
  }
  int x1 = a.x, y1 = a.y, x2 = b.x, y2 = b.y;
  // bresenham 算法 ...
}
```

尝试实现粗细厚度

详细代码如下：

```
void MyGLWidget::bresenhamThicker(FragmentAttr a, FragmentAttr b, int thickness) {
  bresenham(a, b);
  int dy = a.y - b.y;
  int dx = a.x - b.x;
  int x1 = a.x, y1 = a.y, x2 = b.x, y2 = b.y;
  // 四个象限
  int dir1[4][2][2]={
    {
      {-1,-1},
      {1,1}
    },
    {
      {-1,1},
      {1,-1}
    },
    {
      {1,1},
      {-1,-1}
    },
    {
      {-1,1},
      {1,-1}
    }
  };
  if(dx>0 && dy<0) {
    for(int i=1; i<=(thickness+1)/2; i++) {
      FragmentAttr a_ = a, b_ = b;
```

```
      a_.x=x1+i*dir1[0][0][0]+(thickness+1)/2;
      a_.y=y1+i*dir1[0][0][1]+(thickness+1)/2;
      b_.x=x2+i*dir1[0][0][0]+(thickness+1)/2;
      b_.y=y2+i*dir1[0][0][1]+(thickness+1)/2;
      bresenham(a_, b_ );
    }
    for(int i=1; i<=(thickness)/2; i++) {
      FragmentAttr a_ = a, b_ = b;
      a_.x=x1+i*dir1[0][1][0]+(thickness)/2;
      a_.y=y1+i*dir1[0][1][1]+(thickness)/2;
      b_.x=x2+i*dir1[0][1][0]+(thickness)/2;
      b_.y=y2+i*dir1[0][1][1]+(thickness)/2;
      bresenham(a_, b_ );
    }
  } else if(dx<0 && dy<0){
    for(int i=1; i<=(thickness+1)/2; i++) {
      FragmentAttr a_ = a, b_ = b;
      a_.x=x1+i*dir1[1][0][0]+(thickness+1)/2;
      a_.y=y1+i*dir1[1][0][1]+(thickness+1)/2;
      b_.x=x2+i*dir1[1][0][0]+(thickness+1)/2;
      b_.y=y2+i*dir1[1][0][1]+(thickness+1)/2;
      bresenham(a_, b_ );
    }
    for(int i=1; i<=(thickness)/2; i++) {
      FragmentAttr a_ = a, b_ = b;
      a_.x=x1+i*dir1[1][1][0]+(thickness)/2;
      a_.y=y1+i*dir1[1][1][1]+(thickness)/2;
      b_.x=x2+i*dir1[1][1][0]+(thickness)/2;
      b_.y=y2+i*dir1[1][1][1]+(thickness)/2;
      bresenham(a_, b_ );
    }
  }
  else if(dx<0&&dy>0){
    for(int i=1; i<=(thickness+1)/2; i++) {
      FragmentAttr a_ = a, b_ = b;
      a_.x=x1+i*dir1[2][0][0]+(thickness+1)/2;
      a_.y=y1+i*dir1[2][0][1]+(thickness+1)/2;
      b_.x=x2+i*dir1[2][0][0]+(thickness+1)/2;
      b_.y=y2+i*dir1[2][0][1]+(thickness+1)/2;
      bresenham(a_, b_ );
    }
    for(int i=1; i<=(thickness)/2; i++) {
      FragmentAttr a_ = a, b_ = b;
      a_.x=x1+i*dir1[2][1][0]+(thickness)/2;
      a_.y=y1+i*dir1[2][1][1]+(thickness)/2;
      b_.x=x2+i*dir1[2][1][0]+(thickness)/2;
      b_.y=y2+i*dir1[2][1][1]+(thickness)/2;
      bresenham(a_, b_ );
    }
  }
  else if(dx>0&&dy>0){
    for(int i=1; i<=(thickness+1)/2; i++) {
```

```
      FragmentAttr a_ = a, b_ = b;
      a_.x=x1+i*dir1[3][0][0];
      a_.y=y1+i*dir1[3][0][1];
      b_.x=x2+i*dir1[3][0][0];
      b_.y=y2+i*dir1[3][0][1];
      bresenham(a_, b_ );
    }
    for(int i=1; i<=(thickness)/2; i++) {
      FragmentAttr a_ = a, b_ = b;
      a_.x=x1+i*dir1[3][1][0];
      a_.y=y1+i*dir1[3][1][1];
      b_.x=x2+i*dir1[3][1][0];
      b_.y=y2+i*dir1[3][1][1];
      bresenham(a_, b_ );
    }
  }
}
```

1.3 用 edge-walking 填充三角形内部颜色

算法原理

Edge Walking算法是一种用于多边形填充的图形算法，主要用于扫描线填充法。该算法通过跟踪多边形的边界，确定哪些像素应该被填充。

Edge Walking算法的步骤：

1. **边界列表创建**：首先，为多边形的每条边创建一个边界列表，包含每条边的起始和结束点、斜率等信息。
2. **扫描线遍历**：从多边形的最底部开始，逐行扫描，每次处理一条水平扫描线。
3. **边界交点计算**：在每条扫描线上，计算与多边形边界的交点。
4. **填充区域**：根据交点的坐标，将扫描线之间的像素填充颜色。通常是从左交点到右交点填充。
5. **更新边界**：在处理完当前扫描线后，更新边界信息，准备处理下一条扫描线。

代码实现

```
int MyGLWidget::edge_walking(TransformedTriangle const & tri) {
  std::vector<int> edge_starts(WindowSizeH, -1);
  std::vector<int> edge_ends(WindowSizeH, -1);
  int firstChangeLine = -1;

  // 构建边缘表
  for (int y = 0; y < WindowSizeH; ++y) {
    for (int x = 0; x < WindowSizeW; ++x) {
      int index = y * WindowSizeW + x;
      if (temp_render_buffer[index] != vec3(0.0f, 0.0f, 0.0f)) {
        if (edge_starts[y] == -1) {
          edge_starts[y] = x; // 记录第一个边缘
```

```
        }
        edge_ends[y] = x; // 更新最后一个边缘
      }
    }

    if (firstChangeLine == -1 && edge_starts[y] != -1 && edge_ends[y] != -1 &&
edge_starts[y] != edge_ends[y]) {
      firstChangeLine = y;
    }
  }

  // 填充三角形内部区域
  for (int y = 0; y < WindowSizeH; ++y) {
    if (edge_starts[y] != -1 && edge_ends[y] != -1) {
      int x_start = edge_starts[y];
      int x_end = edge_ends[y];
      float z_start = temp_z_buffer[y * WindowSizeW + x_start];
      float z_end = temp_z_buffer[y * WindowSizeW + x_end];

      for (int x = x_start + 1; x < x_end; ++x) {
        // 选择光照计算方法
        switch (shading_mode) {
          case 0: gouraud(x, y, tri); break;
          case 1: phong(x, y, tri); break;
          case 2: blinn_phong(x, y, tri); break;
          default:
            temp_render_buffer[y * WindowSizeW + x] = vec3(1.0f, 1.0f, 1.0f);
        }

        // 插值计算深度值
        float t = static_cast<float>(x - x_start) / (x_end - x_start);
        temp_z_buffer[y * WindowSizeW + x] = (1 - t) * z_start + t * z_end;
      }
    }
  }

  return firstChangeLine != -1 ? firstChangeLine : 0;
}
```

这里在实现光照着色前，默认使用纯色着色。

1.4 讨论

DDA算法：

- **优点**：算法简单，易于实现，适用于计算机图形学中的直线绘制。
- **缺点**：由于浮点运算，可能会导致绘制结果不够精确，尤其在较长的直线段上。

Bresenham算法：

- **优点**：算法效率高，避免了浮点运算，精度较好，适用于直线和其他图形的绘制。
- **缺点**：实现较为复杂，需要处理不同的象限和斜率。

Edge Walking算法：

- **优点**：算法简单、直观，适用于非凸多边形的填充。
- **缺点**：对于复杂多边形，边界管理和交点计算可能会变得复杂。

实验二：实现光照、着色

2.0 光照

漫反射光

漫反射希望某个片段能接收从其他方向的光，而且一般这束光的位置越趋近于法方向，照亮的强度就应该越大。这一个特征使用向量的内积来做反映，内积越大，则说明两个向量夹角越小，并使用其余弦值来进行强度的调整。

环境光

环境光可以认为是实体的“**底色**”，这里没有给出环境光照的比例系数，直接给其设为强度为**1**的值，即得到片段颜色乘上光颜色。

镜面高光

镜面光的展现跟观察者(摄像机)的位置有关，因此我们需要记录从摄像机到片段的方向向量。

2.1 用 Gouraud 实现三角形内部的着色

算法原理

Gouraud算法是一种用于光照计算的平滑着色算法，常用于计算机图形学中的三维模型渲染。它通过插值顶点颜色来实现表面光滑过渡，减少了由于多边形边缘造成的“锯齿”效果。

Gouraud算法的步骤：

1. **计算顶点颜色**：根据光照模型（如Phong反射模型），为每个顶点计算颜色值。这通常考虑了光源的位置、材料属性和视角。
2. **插值颜色**：在渲染多边形时，使用顶点颜色进行线性插值，计算多边形内每个像素的颜色值。
3. **绘制多边形**：将插值得到的颜色应用于多边形的每个像素，形成平滑的光照效果。

代码实现

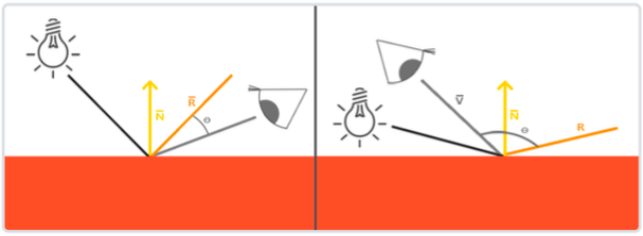
```
#include "myglwidget.h"

void MyGLWidget::gouraud(int x, int y, TransformedTriangle const & tri) {
  auto l = tri.lines;
  float alpha = (
    (
      static_cast<float>(y - l[1].y) * (l[2].x - l[1].x) - static_cast<float>(x - l[1].x)
* (l[2].y - l[1].y)
    ) / (
      static_cast<float>(l[0].y - l[1].y) * (l[2].x - l[1].x) - static_cast<float>(l[0].x
- l[1].x) * (l[2].y - l[1].y)
    )
  );
  float beta  = (
    static_cast<float>(y - l[2].y) * (l[0].x - l[2].x) - static_cast<float>(x - l[2].x) *
(l[0].y - l[2].y)
  ) / (
    static_cast<float>(l[1].y - l[2].y) * (l[0].x - l[2].x) - static_cast<float>(l[1].x -
l[2].x) * (l[0].y - l[2].y)
  );
  float gamma = 1 - alpha - beta;
  vec3 color = alpha * l[0].color + beta * l[1].color + gamma * l[2].color;
  temp_render_buffer[y * WindowSizeW + x] = color;
}
```

2.2 用 Phong 模型实现三角形内部的着色

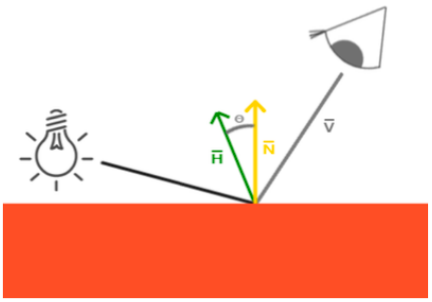
算法原理

Phong算法是一种用于计算机图形学中的光照模型，旨在模拟表面光照与反射效果。它通过考虑环境光、漫反射和镜面反射来计算物体表面的颜色，从而实现更真实的视觉效果。

Phong 模型观察漫反射与镜面反射的区别：



我们希望右图能在物体的反光度非常小时不再忽略它们对镜面光分量的贡献。于是引入半程分量：



利用半程分量与法线向量的夹角余弦来量化观察到的镜面光照分量：

$$获取半程分量：H = \frac{L+V}{||L+V||}$$

Phong算法的主要组成部分：

1. **环境光**：表示物体在环境光照下的基本亮度，通常为常量。
2. **漫反射**：根据光源与表面法线之间的角度计算出漫反射光的强度。使用Lambertian反射模型，亮度与入射光的角度成正比。
3. **镜面反射**：模拟高光效果，依赖于观察者视角与反射光线之间的角度。采用焦散（Cosine）模型，通常使用一个指数系数来控制高光的大小和强度。

Phong算法的步骤：

1. **计算光照分量**：为每个像素计算环境光、漫反射和镜面反射分量。
2. **组合光照**：将三个光照分量相加，得到最终颜色值。
3. **应用于渲染**：将计算出的颜色值应用于物体的表面，实现光照效果。

代码实现

```
void MyGLWidget::phong(int x, int y, TransformedTriangle const & tri){
  auto l = tri.lines;

  // 计算重心
  float alpha = (
      (static_cast<float>(y - l[1].y) * (l[2].x - l[1].x) - static_cast<float>(x - l[1].x)
* (l[2].y - l[1].y)
    ) / (
      static_cast<float>(l[0].y - l[1].y) * (l[2].x - l[1].x) - static_cast<float>(l[0].x
- l[1].x) * (l[2].y - l[1].y)
    )
  );
  float beta = (
```

```
    static_cast<float>(y - l[2].y) * (l[0].x - l[2].x) - static_cast<float>(x - l[2].x) *
(l[0].y - l[2].y)
  ) / (
    static_cast<float>(l[1].y - l[2].y) * (l[0].x - l[2].x) - static_cast<float>(l[1].x -
l[2].x) * (l[0].y - l[2].y)
  );
  float gamma = 1.0f - alpha - beta;

  // 插值法向量和位置
  vec3 interpolatedNormal = glm::normalize(alpha * l[0].normal + beta * l[1].normal +
gamma * l[2].normal);
  vec3 interpolatedPosMV = alpha * vec3(l[0].pos_mv) + beta * vec3(l[1].pos_mv) + gamma *
vec3(l[2].pos_mv);

  FragmentAttr nowPixelResult(x, y, 0.0f, 0);
  nowPixelResult.normal = interpolatedNormal;
  nowPixelResult.pos_mv = interpolatedPosMV;

  // 光照计算
  vec3 lightDir = glm::normalize(lightPosition - nowPixelResult.pos_mv);
  vec3 viewDir = glm::normalize(camPosition - nowPixelResult.pos_mv);
  vec3 reflectDir = glm::reflect(-lightDir, nowPixelResult.normal);

  // 颜色计算
  vec3 ambient = ambientStrength * lightColor;
  float diff = glm::max(glm::dot(nowPixelResult.normal, lightDir), 0.0f);
  vec3 diffuse = diff * lightColor;
  float spec = glm::pow(glm::max(glm::dot(viewDir, reflectDir), 0.0f), 16);
  vec3 specular = specularStrength * spec * lightColor;

  vec3 color = (ambient + diffuse + specular) * objectColor;
  temp_render_buffer[y * WindowSizeW + x] = color;
}
```

2.3 用 Blinn-Phong 实现三角形内部的着色

算法原理

Blinn-Phong算法是对Phong光照模型的改进，旨在提高计算效率并改善高光效果的表现。它在计算镜面反射时采用了不同的方法，适用于计算机图形学中的3D渲染。

Blinn-Phong算法的主要组成部分：

1. **环境光**：与Phong模型相同，表示物体在环境光照下的基本亮度。
2. **漫反射**：与Phong模型相似，使用Lambertian反射模型计算光源与表面法线之间的角度。
3. **镜面反射**：Blinn-Phong算法通过计算半向量（即光源向量与视线向量的中间向量）来代替直接的反射向量。这样可以减少计算复杂度，并在某些情况下改善高光效果。

Blinn-Phong算法的步骤：

1. **计算光照分量**：为每个像素计算环境光、漫反射和镜面反射分量。

2. **使用半向量**：在计算镜面反射时，通过半向量与法线的点积来确定高光强度。
3. **组合光照**：将三种光照分量相加，得到最终颜色值。

代码实现

```
void MyGLWidget::blinn_phong(int x, int y, TransformedTriangle const & tri) {
  auto l = tri.lines;
  float alpha = (
    (
      static_cast<float>(y - l[1].y) * (l[2].x - l[1].x) - static_cast<float>(x - l[1].x)
* (l[2].y - l[1].y)
    ) / (
      static_cast<float>(l[0].y - l[1].y) * (l[2].x - l[1].x) - static_cast<float>(l[0].x
- l[1].x) * (l[2].y - l[1].y)
    )
  );
  float beta = (
    static_cast<float>(y - l[2].y) * (l[0].x - l[2].x) - static_cast<float>(x - l[2].x) *
(l[0].y - l[2].y)
  ) / (
    static_cast<float>(l[1].y - l[2].y) * (l[0].x - l[2].x) - static_cast<float>(l[1].x -
l[2].x) * (l[0].y - l[2].y)
  );
  float gamma = 1 - alpha - beta;

  // 插值法计算法线和位置
  vec3 normalInterpolated = alpha * l[0].normal + beta * l[1].normal + gamma *
l[2].normal;
  vec3 positionMVInterpolated = alpha * vec3(l[0].pos_mv) + beta * vec3(l[1].pos_mv) +
gamma * vec3(l[2].pos_mv);

  FragmentAttr currentPixelResult(x, y, 0.0f, 0);
  currentPixelResult.normal = normalize(normalInterpolated);
  currentPixelResult.pos_mv = positionMVInterpolated;

  vec3 normalizedNormal = normalize(currentPixelResult.normal);

  // 光照计算
  vec3 lightDirection = normalize(lightPosition - currentPixelResult.pos_mv);
  vec3 viewDirection = normalize(camPosition - currentPixelResult.pos_mv);
  vec3 halfVector = normalize(lightDirection + viewDirection);

  vec3 ambientLight = ambientStrength * lightColor;
  float diffuseFactor = max(dot(normalizedNormal, lightDirection), 0.0f);
  vec3 diffuseLight = diffuseFactor * lightColor;

  float specularFactor = pow(max(dot(normalizedNormal, halfVector), 0.0f), 16);
  vec3 specularLight = specularStrength * specularFactor * lightColor;
```

```
    vec3 color = (ambientLight + diffuseLight + specularLight) * objectColor;

    temp_render_buffer[y * WindowSizeW + x] = color;
}
```

2.4 讨论

Gouraud 算法：

- **优点**：算法简单、计算效率高，适合实时渲染，能够有效减少色彩不连续现象。
- **缺点**：在高光或细节丰富的区域可能出现光照失真，因为算法只在顶点计算光照，未考虑表面细节。

Phong 算法：

- **优点**：能生成非常真实的光照效果，适用于静态和动态场景。
- **缺点**：计算复杂度较高，尤其在处理复杂场景时，可能影响渲染性能。

Blinn-Phong 算法：

- **优点**：计算效率高于传统Phong模型，特别是在处理动态场景时表现更好，同时能够产生更自然的高光效果。
- **缺点**：仍然可能在非常细腻的高光表现上有所不足，尤其是在明亮的光源下。