计算机图形学课程作业

Assignment2

Rasterization and Z-buffering

姓 名: 李 昊 伟
学 号: 20337056
班 号: 20 级计科一班

计算机图形学
(秋季, 2022)

中山大学
计算机学院
SYSUCSE

2022 年 10 月 26 日

目 录

1 概述

本次作业报告主要是对 7 个 task 的完成过程的介绍以及知识点的解析。作业程序代码可访问本人 github 代码仓库：https://github.com/distancemay5/learn_cg。实现效果以视频的形式保存在作业提交的压缩包。

2 作业任务解答

2.1 线性插值和三角形重心插值

所谓插值，就是通过直线、三角形等图形在有限个点处的取值状况，比如在端点处的取值状况，估算出图形其他位置、颜色的近似值。

2.1.1 线性插值

已知线段两个端点的位置、颜色：对于一条线段上其他点的位置，我们可以直接表示为两个端点坐标的加权和。根据相似三角形原理，两个端点的权重分别为 1-frac 和 frac，其中 $frac = \frac{i}{dx}$，i 代表从左边端点数的第 i 个点，而 dx 代表线段在 x 轴方向的投影长度。当然，$frac = \frac{i}{dy}$ 也是一样的道理。我们假设线段上点的颜色是渐变的，所以每个点的 RGB 值也可以由这个加权比例求加权和实现。

```
  TRShaderPipeline::VertexData TRShaderPipeline::VertexData::lerp(
  const TRShaderPipeline::VertexData& v0,
  const TRShaderPipeline::VertexData& v1,
  float frac)
{
  //Linear interpolation
  VertexData result;
  result.pos = (1.0f - frac) * v0.pos + frac * v1.pos;
  result.col = (1.0f - frac) * v0.col + frac * v1.col;
  result.nor = (1.0f - frac) * v0.nor + frac * v1.nor;
  result.tex = (1.0f - frac) * v0.tex + frac * v1.tex;
  result.cpos = (1.0f - frac) * v0.cpos + frac * v1.cpos;
  result.spos.x = (1.0f - frac) * v0.spos.x + frac * v1.spos.x;
  result.spos.y = (1.0f - frac) * v0.spos.y + frac * v1.spos.y;

  return result;
}
...
VertexData tmp = VertexData::lerp(from, to, static_cast<double>(i) / dx);
```

2.1.2 三角形重心插值

三角形重心插值是把三角形内部任意一点表示为三角形三个顶点的加权和。对于三角形内任意一个点 P，如下图:

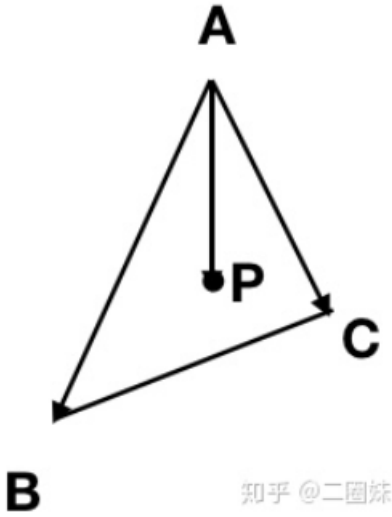

图 1: 三角形内任意一点的表示

显然，$\vec{AP} = u\vec{AB} + v\vec{AC}$。这个式子展开得到，$\vec{OP} - \vec{OA} = u(\vec{OB} - \vec{OA}) + v(\vec{OC} - \vec{OA})$，整理得:

$$P = (1 - u - v)A + uB + vC$$

图 2: P 点表示

只要求出 u、v 我们就可以用 (1-u-v)、u 、v 来作为比例来进行插值了。那么，如何求这两个参数呢?

由 $\vec{AP} = u\vec{AB} + v\vec{AC}$ 得，$u\vec{AB} + v\vec{AC} + \vec{PA} = \vec{0}$，也就是两对向量垂直:

$$\begin{bmatrix} u & v & 1 \end{bmatrix} \begin{bmatrix} \overrightarrow{AB}_x \\ \overrightarrow{AC}_x \\ \overrightarrow{PA}_x \end{bmatrix} = 0$$

$$\begin{bmatrix} u & v & 1 \end{bmatrix} \begin{bmatrix} \overrightarrow{AB}_y \\ \overrightarrow{AC}_y \\ \overrightarrow{PA}_y \end{bmatrix} = 0$$

图 3: P 点表示规律

可见向量 $(u, v, 1)$ 是 $(\vec{AB}_x, \vec{AC}_x, \vec{PA}_x)$ 和 $(\vec{AB}_y, \vec{AC}_y, \vec{PA}_y)$ 的公垂线，也就是二者的叉乘。

```
glm::vec2 pot = glm::vec2(i, j);
glm::vec2 l1 = v1.spos - v0.spos;
glm::vec2 l2 = v2.spos - v0.spos;
glm::vec2 l3 = glm::vec2(v0.spos) - pot;
glm::vec3 xvector = glm::vec3(l1.x, l2.x, l3.x);
glm::vec3 yvector = glm::vec3(l1.y, l2.y, l3.y);
glm::vec3 u = glm::cross(xvector, yvector);
glm::vec3 w = glm::vec3(1.0f - (u.x + u.y) / u.z, u.x / u.z, u.y
    / u.z);
```

如上代码，则 w 向量为上面所说的 $(u, v, 1)$，只要注意最后一步的除法，使得 u.z = 1 即可。

2.2 Task1 实现 Bresenham 直线光栅化算法

2.2.1 问题分析

如图，在直线光栅化过程中，给定直线方程和起始点后，程序不断地从起始点向前推进，也就是将 x 加上 1，试图找到新的 x 对应的点的像素点位置。我们讨论直线斜率小于四十五度的情况。大于四十五度其实是一样的，只需要把 x 和 y 颠倒就行了。

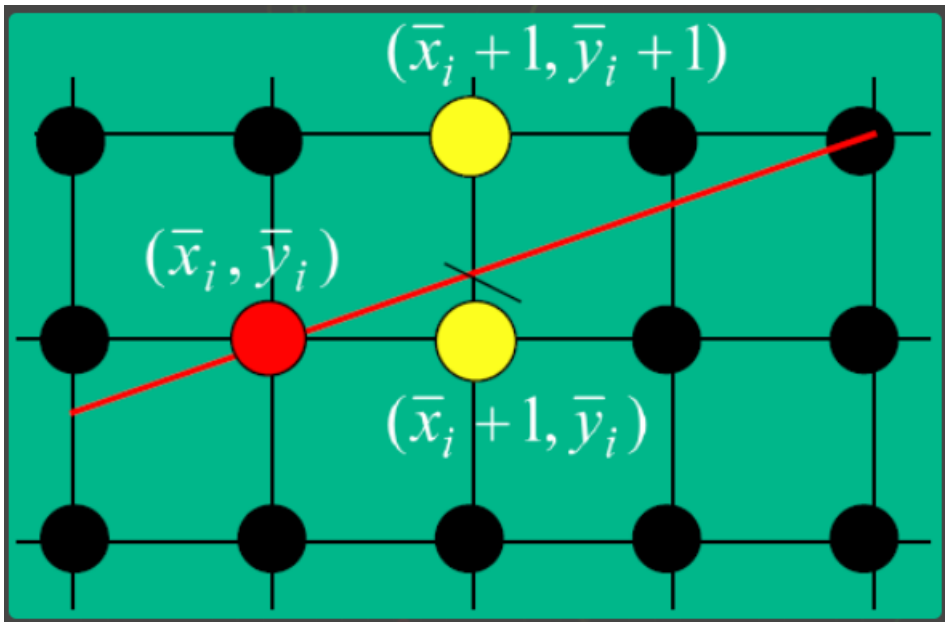

图 4: 直线光栅化问题描述

如果斜率小于四十五度，那么对于 x_{i+1} 相对于 x_i, y_{i+1} 有两种可能，y_i 或者 y_i+1，注意这两个数是整数。只需要将 x_{i+1} 带入直线方程计算出浮点数 y，看看 y 和哪个 y_{i+1} 更加接近即可。

假设直线方程为 $y=mx+B$，那么对于 x_i 和 $x_{i+1}=x_i+1$，$y_{i+1}=mx_i+m+B$。其中，$m=\frac{\triangle y}{\triangle x}$ 如果用 $\bar{y}$ 表示像素点实际的纵坐标 (这个值不再是浮点数而是一个整数)，则与两个可能的像素点的距离分别是：

$$d_{upper}=\bar{y}_i+1-\bar{y}_{i+1}=\bar{y}_i+1-m\bar{x}_{i+1}-B$$
$$d_{lower}=y_{i+1}-\bar{y}_i=mx_{i+1}+B-\bar{y}_i$$

图 5: 距离的表示

二者做差，就得到像素点选择判别式:

$$\begin{aligned} d_{lower}-d_{upper} &= m(x_i+1)+B-\bar{y}_i-(\bar{y}_i+1-m(x_i+1)-B) \\ &= 2m(x_i+1)-2\bar{y}_i+2B-1 \end{aligned}$$

图 6: 像素点选择判别式

若这个值大于 0，则应该选择右上方的点；若小于 0，则应该选择右方的点。Bresenham 算法的优点在于使用增量计算，使得只需要检查误差项的符号，就可以确定需要绘制的点。以下是该算法的思路：

首先认为直接进行浮点数除法运算计算 m 的过程是昂贵的，所以两边同乘 $\triangle x$ 消除浮点数除法计算。此时，判别式 p_i 变成如图 7 所示:

$$
\begin{aligned}
p_i &= \Delta x \cdot (d_{lower} - d_{upper}) = 2\Delta y \cdot (x_i + 1) - 2\Delta x \cdot \overline{y}_i + (2B - 1)\Delta x \\
&= 2\Delta y \cdot x_i - 2\Delta x \cdot \overline{y}_i + c \\
&where\ \ c = (2B - 1)\Delta x + 2\Delta y
\end{aligned}
$$

图 7: 新像素点选择判别式

该式应该可以进一步优化，因为我们计算 p_{i+1} 时忽略了之前计算的成果，没有利用到上次选择的信息，也就是 p_i 的符号信息。如下图，我们将 p_{i+1} 与 p_i 做差:

$$
\begin{aligned}
p_{i+1} - p_i &= (2\Delta y \cdot x_{i+1} - 2\Delta x \cdot \overline{y}_{i+1} + c) - (2\Delta y \cdot x_i - 2\Delta x \cdot \overline{y}_i + c) \\
&= 2\Delta y - 2\Delta x(\overline{y}_{i+1} - \overline{y}_i)
\end{aligned}
$$

图 8: p_{i+1} 与 p_i 做差

发现 p_{i+1} 可以表示为 p_i 与其增量的和。该增量也取决于 p_i。如果 $p_i > 0$，$\overline{y_{i+1}} = \overline{y_i} + 1$ ，不然 $\overline{y_{i+1}} = \overline{y_i}$。于是，我们得到该算法最终的利用增量计算判别式的公式。

若 $p_i \leq 0$，那么选择右边的点，此时 $\overline{y}_{i+1} = \overline{y}_i$，那么有:

$$p_{i+1} = p_i + 2\Delta y$$

若 $p_i > 0$，那么选择右上角的点，此时 $\overline{y}_{i+1} = \overline{y}_i + 1$，那么有:

$$p_{i+1} = p_i + 2\Delta y - 2\Delta x$$

图 9: Bresenham 判别式计算

2.2.2 代码实现

```
void TRShaderPipeline::rasterize_wire_aux(
  const VertexData& from,
  const VertexData& to,
  const unsigned int& screen_width,
  const unsigned int& screen_height,
  std::vector<VertexData>& rasterized_points)
{
  int dx = to.spos.x - from.spos.x;
  int dy = to.spos.y - from.spos.y;
  int stepX = 1, stepY = 1;

  // 判断from点和to点的相对位置
  if (dx < 0)
  {
    stepX = -1;
    dx = -dx;
  }
  if (dy < 0)
  {
    stepY = -1;
    dy = -dy;
  }

  // 计算d2y - d2x
  int d2x = 2 * dx, d2y = 2 * dy;
  int d2y_minus_d2x = d2y - d2x;
  int sx = from.spos.x;
  int sy = from.spos.y;

  // slope < 1.
  if (dy <= dx)
  {
    // p0
    int p = d2y - dx;
    for (int i = 0; i <= dx; ++i)
    {
      // linear interpolation
      VertexData tmp = VertexData::lerp(from, to, static_cast<double>(i)
          / dx);
      sx += stepX;
      if (p <= 0)
        p += d2y;
      else
      {
        sy += stepY;
        p += d2y_minus_d2x;
      }
      tmp.spos.x = sx;
```

```
      tmp.spos.y = sy;
      if (tmp.spos.x < screen_width && tmp.spos.y < screen_height) {
        rasterized_points.push_back(tmp);
      }
    }
  }

  // slope > 1.
  else
  {
    int p = d2x - dy;
    for (int i = 0; i <= dy; ++i)
    {
      // linear interpolation
      VertexData tmp = VertexData::lerp(from, to, static_cast<double>(i)
          / dy);
      sy += stepY;
      if (p <= 0)
        p += d2x;
      else
      {
        sx += stepX;
        p -= d2y_minus_d2x;
      }
      tmp.spos.x = sx;
      tmp.spos.y = sy;
      if (tmp.spos.x < screen_width && tmp.spos.y < screen_height) {
        rasterized_points.push_back(tmp);
      }
    }
  }



  //Task1: Implement Bresenham line rasterization
  // Note: You shold use VertexData::lerp(from, to, weight) for
      interpolation,
  //       interpolated points should be pushed back to rasterized_points.
  //       Interpolated points shold be discarded if they are outside the
      window.

  //       from.spos and to.spos are the screen space vertices.

  //For instance:
  rasterized_points.push_back(from);
  rasterized_points.push_back(to);

}
```

2.3 Task2 + Task7 简单的齐次空间裁剪以及 Sutherland-Hodgeman 裁剪算法

2.3.1 问题分析

在第一次作业当中，我们实现了投影矩阵的计算，把摄像机空间的点投影到了裁剪空间。在计算过程中，我们假设了点都是在平截头体内部的。但是事实上，在平截头体之外，比如在远平面和近平面外面，还有一些点没有被去除。在经过变换后，他们的坐标值不在 $[-1,1]$ 之间。事实上，这些点的坐标属于无用值，需要把他们去掉，我们在渲染结果当中也不需要这些点，相当于在我们视野当中根本看不到这些点，因为他们不是太远了，就是太近了，或者根本不在视角当中。

如果不裁剪，就会带来麻烦。比如一些点的 w 值是 0，就会带来除 0 错误。比如一些点处于观察者身后，其观察坐标 z 会大于 0，那么透视投影之后 w 会小于 0，进行透视除法会导致顶点的 x_{ndc}, y_{ndc} 符号翻转。

想要除去这些不合理的点，最直观的想法是按照一个一个三角形整体处理，使得只要有一个顶点在可见的范围之内，那么就返回三角形的全部顶点；否则三角形的三个顶点全部不在可见范围之内，此时应该把他裁剪掉。这就是 task2 的🄴容，实际上就是判断标准化后的点是否在标准化坐标空间的立方体内，也就是坐标大小是不是在 [-1,1] 范围之内。而且只要有一个点在这个范围之内，那么就保留整个三角形。

首先我们要定义上下左右前后六个平面的位置。三维平面的一般形式方程为 $Ax + BY + Cz + D = 0$，其中 $n = (A, B, C)$ 为平面法线。对于任意一点 P(x,y,z)，其到平面的距离为 $Ax + By + Cz + D$，若 $d = 0$ 则点在平面上，$d > 0$ 说明点在平面法线所指向的一侧，$d < 0$ 说明在另一侧。以下是六个包围平面：

- 左 $A = 1, B = 0, C = 0, D = w$
- 右 $A = -1, B = 0, C = 0, D = w$
- 上 $A = 0, B = -1, C = 0, D = w$
- 下 $A = 0, B = 1, C = 0, D = w$
- 近 $A = 0, B = 0, C = 1, D = w$
- 远 $A = 0, B = 0, C = -1, D = w$

图 10: 六个包围平面

接下来只要实现判断点在平面哪一侧的算法，进而实现点是否位于包围平面构成立方体内部的算法，就可以完成 task2。

Sutherland-Hodgeman 算法或者叫逐边裁剪算法，在二维平面下的原理是遍历每条裁剪边，生成顶点并作为下一条边的输入。也就是说三角形会被部分裁剪，并且生成一个多边形，再重新划分为三角形。如图：

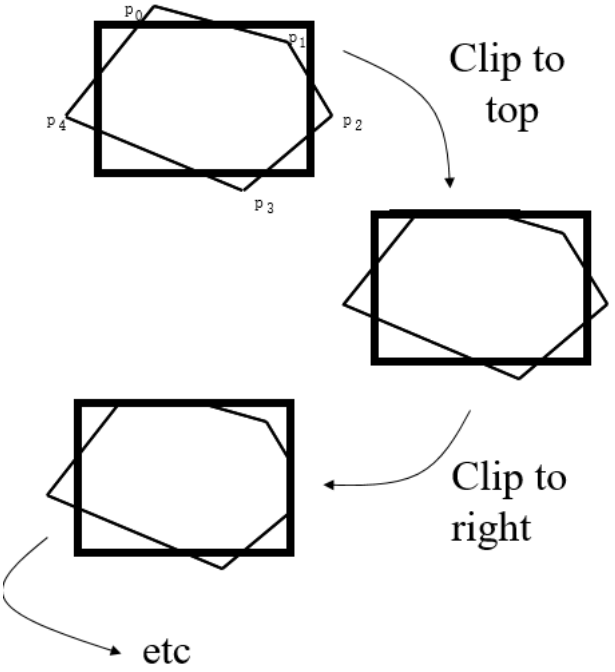

图 11: Sutherland-Hodgeman 算法图示

其实现思路如下：如果三角形有一条边的两个顶点位于平面的两侧，那么将三角形这条边与平面的交点作为新的顶点而舍去在平面外的点。对于六个平面，逐个处理，直到生成合格的多边形。如果有两点 $P_1 = (x_1, y_1, z_1)$ 和 $P_2 = (x_2, y_2, z_2)$ 分别位于平面两侧，则可以通过权重 $\lambda = \frac{d_1}{d_1 - d_2}$，插值得到它们连线与平面的交点 $P = \lambda P_2 + (1 - \lambda) P_1$。

2.3.2 代码实现

```
bool TRRenderer::Inside(float x, float y, float z, float w , const
     glm::vec4& p) const {
  return x * p.x + y * p.y + z * p.z + w * p.w >= 0;
}

bool TRRenderer::PointInside(const std::vector<glm::vec4> lines , const
     glm::vec4& p) const {
  bool flag = true;
  for (auto &line : lines) {
    flag = flag && Inside(line.x , line.y , line.z , line.w , p);
```

```
  }
  return flag;
}
TRShaderPipeline::VertexData Intersect(const
    TRShaderPipeline::VertexData& v1, const
    TRShaderPipeline::VertexData& v2, const glm::vec4& line) {
  float da = v1.cpos.x * line.x + v1.cpos.y * line.y + v1.cpos.z * line.z
      + line.w * v1.cpos.w;
  float db = v2.cpos.x * line.x + v2.cpos.y * line.y + v2.cpos.z * line.z
      + line.w * v2.cpos.w;

  float weight = da / (da - db);
  return TRShaderPipeline::VertexData::lerp(v1, v2, weight);
}
std::vector<TRShaderPipeline::VertexData> TRRenderer::cliping(
  const TRShaderPipeline::VertexData &v0,
  const TRShaderPipeline::VertexData &v1,
  const TRShaderPipeline::VertexData &v2) const
{
  //Clipping in the homogeneous clipping space

  //Task2: Implement simple vertex clipping
  // Note: If one of the vertices is inside the [-w,w]^3 space (and w
      should be in [near, far]),
  //       just return {v0, v1, v2}. Otherwise, return {}

  //       Please Use v0.cpos, v1.cpos, v2.cpos
  //       m_frustum_near_far.x -> near plane
  //       m_frustum_near_far.y -> far plane
  // simple clipping

  const std::vector<glm::vec4> ViewLines = {
    //near
    glm::vec4(0,0,1,1),
    //far
    glm::vec4(0,0,-1,1),
    //left
    glm::vec4(1,0,0,1),
    //right
    glm::vec4(-1,0,0,1),
    //top
    glm::vec4(0,-1,0,1),
    //bottom
    glm::vec4(0,1,0,1)
  };
  /*bool flag = false;
  flag = flag || PointInside(ViewLines, v0.cpos) ||
      PointInside(ViewLines, v1.cpos) || PointInside(ViewLines, v2.cpos);
  // flag = flag || (PointInside(ViewLines, v0.cpos) && v0.cpos.w > 0) ||
      (PointInside(ViewLines, v1.cpos) && v1.cpos.w > 0) ||
      (PointInside(ViewLines, v2.cpos) && v2.cpos.w > 0);
```

```
  if(flag)
    return { v0, v1, v2 };
  return {};*/


  std::vector<TRShaderPipeline::VertexData> output = {v0,v1,v2};
  if ( PointInside(ViewLines , v0.cpos) && PointInside(ViewLines,
       v1.cpos) && PointInside(ViewLines, v2.cpos)) {
    return output;
  }
  else {
    for (int i = 0; i < ViewLines.size(); i++) {
      std::vector<TRShaderPipeline::VertexData> input(output);
      output.clear();
      for (int j = 0; j < input.size(); j++) {
        TRShaderPipeline::VertexData current = input[j];
        TRShaderPipeline::VertexData last = input[(j + input.size() - 1)
            % input.size()];
        if (Inside(ViewLines[i].x , ViewLines[i].y , ViewLines[i].z ,
            ViewLines[i].w , current.cpos)) {
          if (!Inside(ViewLines[i].x, ViewLines[i].y, ViewLines[i].z,
              ViewLines[i].w, last.cpos)) {
            TRShaderPipeline::VertexData intersecting = Intersect(last,
                current, ViewLines[i]);
            output.push_back(intersecting);
          }
          output.push_back(current);
        }
        else if (Inside(ViewLines[i].x, ViewLines[i].y, ViewLines[i].z,
            ViewLines[i].w, last.cpos)) {
          TRShaderPipeline::VertexData intersecting = Intersect(last,
              current, ViewLines[i]);
          output.push_back(intersecting);
        }
      }
    }
    return output;
  }
}
```

2.4 Task3 实现三角形的背向面剔除

2.4.1 问题分析

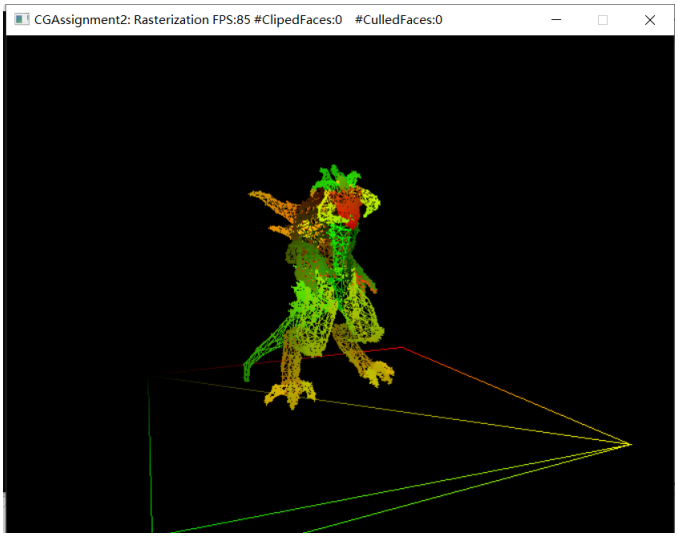

图 12: 未进行背向面剔除的效果

如图，实现一二任务之后，渲染出的图像似乎怪怪的，似乎有一些重叠和臃肿，其原因是没有做背向面剔除。当我们看向一个物体时，永远只有朝向我们的那一面可以被我们看到。从三角形的角度来讲，永远只有法线朝向我们的眼睛的三角形可以被我们看到。所以，我们需要进行背面剔除，删去很大一部分三角形。

我们可以通过叉乘得到三角形的法线朝向，然后与视线方向进行点乘，根据点乘结果大于 0 还是小于 0 来判断三角形此时是否是正面朝向还是背面朝向，如果背面朝向，则应该直接剔除，不进行光栅化等后续的处理，这就是背向面剔除的基本原理。只要进行了背向面剔除，会减少很多三角形，我们的怪兽看起来也更简洁合理了。

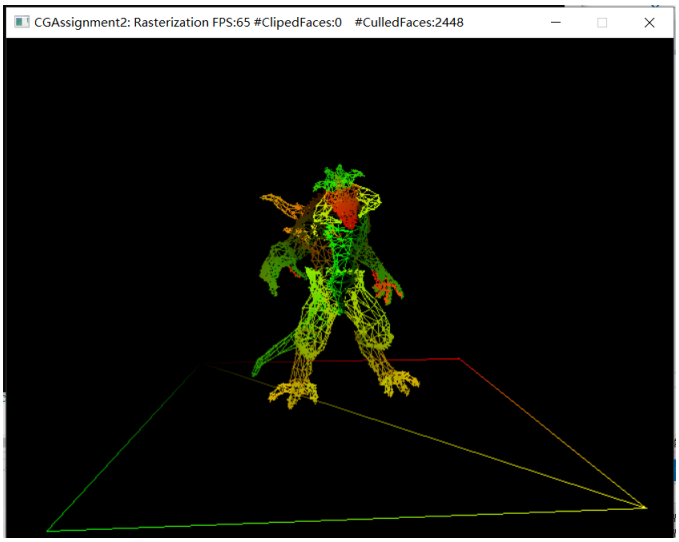

图 13: 进行背向面剔除的效果

2.4.2 代码实现

```
bool TRRenderer::isTowardBackFace(const glm::vec4 &v0, const glm::vec4
    &v1, const glm::vec4 &v2) const
{
  //Back face culling in the ndc space

  // Task3: Implement the back face culling
  //  Note: Return true if it's a back-face, otherwise return false.
  glm::vec3 e1 = glm::vec3((v1 - v0));
  glm::vec3 e2 = glm::vec3((v2 - v1));
  glm::vec3 normal_vector = glm::cross(e1 , e2);
  glm::vec3 watch = glm::vec3(0.0f, 0.0f, -1.0f);
  float dot_result = glm::dot(normal_vector , watch);
  return dot_result > 0;
}
```

2.5 Task4 实现基于 Edge-function 的三角形填充算法

2.5.1 问题分析

基于 Edge-function 的三角形填充算法首先计算三角形的包围盒，然后遍历包围盒内的像素点，判断该像素点是否在三角形内部，这就是它的基本原理。判断完毕是否在三角形内部以后，进行三角形重心插值的算法，已经在 2.1.2 中详细介绍。现在只介绍如何判断点在三角形内部的算法。

对于三角形内部的点 P，我们可以知道，P 点位于 $\vec{AB}$、$\vec{BC}$ 、$\vec{CA}$ 的

同一侧。也就是说，我们只需要判断三个向量的起始点到 P 的向量与对应向量的叉乘，也就是 $\vec{AP}$ 与 $\vec{AB}$、$\vec{BP}$ 与 $\vec{BC}$、$\vec{CP}$ 与 $\vec{CA}$ 的叉乘，是否指向同一方向即可。

在本任务中，我还特别实现了抗锯齿算法，对于一个像素点 (x, y)，我分别判断 $(x-0.25, y-0.25)$、$(x-0.25+y+0.25)$、$(x+0.25, y-0.25)$、$(x+0.25, y+0.25)$ 是否在三角形内，全在赋予全色，否则颜色按照在三角形内的点和全部点的比例打折。如果没有点在三角形内，就不点亮该像素。实现效果:

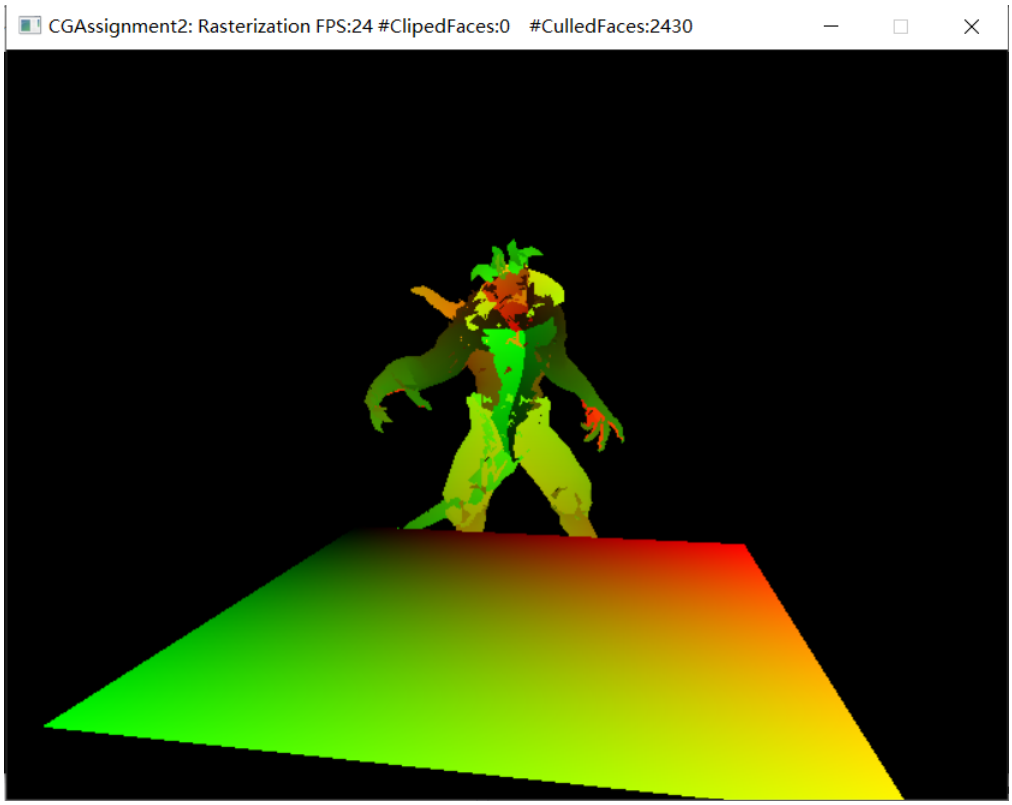

图 14: Task4 的效果

2.5.2 代码实现

```
bool TRShaderPipeline::InsideTriangle(const glm::ivec2& v0, const
    glm::ivec2& v1, const glm::ivec2& v2, const glm::vec2& p) {
  glm::vec3 e0 = glm::vec3(v1 - v0, 1.0f);
  glm::vec3 e1 = glm::vec3(v2 - v1, 1.0f);
  glm::vec3 e2 = glm::vec3(v0 - v2, 1.0f);
  glm::vec3 point = glm::vec3(p, 1.0f);
```

```
  bool f0 = (glm::cross((point - glm::vec3(v0, 1.0f)), e0).z >= 0);
  bool f1 = glm::cross((point - glm::vec3(v1, 1.0f)), e1).z >= 0;
  bool f2 = glm::cross((point - glm::vec3(v2, 1.0f)), e2).z >= 0;
  return (f0 == f1) && (f1 == f2);
}

void TRShaderPipeline::rasterize_fill_edge_function(
  const VertexData& v0,
  const VertexData& v1,
  const VertexData& v2,
  const unsigned int& screen_width,
  const unsigned int& screene_height,
  std::vector<VertexData>& rasterized_points)
{
  int minX, maxX, minY, maxY;
  minX = maxX = v0.spos.x;
  minY = maxY = v0.spos.y;

  minX = std::min(minX, v1.spos.x);
  minX = std::min(minX, v2.spos.x);
  maxX = std::max(maxX, v1.spos.x);
  maxX = std::max(maxX, v2.spos.x);
  minY = std::min(minY, v1.spos.y);
  minY = std::min(minY, v2.spos.y);
  maxY = std::max(maxY, v1.spos.y);
  maxY = std::max(maxY, v2.spos.y);
  for (int i = minX; i <= maxX; i++) {
    for (int j = minY; j <= maxY; j++) {
      glm::vec3 color = { 0.0f , 0.0f , 0.0f };
      glm::vec2 tmp0 = { i - 0.25 , j + 0.25 };
      glm::vec2 tmp1 = { i - 0.25 , j - 0.25 };
      glm::vec2 tmp2 = { i + 0.25 , j + 0.25 };
      glm::vec2 tmp3 = { i + 0.25 , j - 0.25 };
      std::vector<glm::vec2> tmp_points = { tmp0 , tmp1 , tmp2 , tmp3 };
      int num = 0;
      for (auto tmp_p : tmp_points) {
        if (InsideTriangle(v0.spos, v1.spos, v2.spos, tmp_p)) {
          num += 1;
        }
      }
      if (num > 0) {
        glm::vec2 pot = glm::vec2(i, j);
        glm::vec2 l1 = v1.spos - v0.spos;
        glm::vec2 l2 = v2.spos - v0.spos;
        glm::vec2 l3 = glm::vec2(v0.spos) - pot;
        glm::vec3 xvector = glm::vec3(l1.x, l2.x, l3.x);
        glm::vec3 yvector = glm::vec3(l1.y, l2.y, l3.y);
        glm::vec3 u = glm::cross(xvector, yvector);
        glm::vec3 w = glm::vec3(1.0f - (u.x + u.y) / u.z, u.x / u.z, u.y
            / u.z);
        VertexData tmp = VertexData::barycentricLerp(v0, v1, v2, w);
```

```
            tmp.spos.x = i;
            tmp.spos.y = j;
            tmp.col = glm::vec3{ tmp.col.x / num , tmp.col.y / num ,
                tmp.col.z / num };
            rasterized_points.push_back(tmp);
          }
        }
      }
      //Edge-function rasterization algorithm

      //Task4: Implement edge-function triangle rassterization algorithm
      // Note: You should use VertexData::barycentricLerp(v0, v1, v2, w) for
          interpolation,
      //       interpolated points should be pushed back to rasterized_points.
      //       Interpolated points shold be discarded if they are outside the
          window.

      //       v0.spos, v1.spos and v2.spos are the screen space vertices.

      //For instance:

  }
```

2.6 Task5 实现深度测试

2.6.1 问题分析

为了体现正确的三维前后遮挡关系，我们实现的帧缓冲包含了一个深度缓冲，用于存储当前场景中最近物体的深度值，前后的。三角形的三个顶点经过一系列变换之后，其 z 存储了深度信息，取值为 [-1,1] ，越大则越远。思路上，在处理每一个点时，查阅深度缓冲当中对应于该点坐标的最浅深度。如果该点的深度更要浅，那就渲染这个点，并且更新最浅深度。否则，该点已被遮挡，不需要渲染。

实现效果:

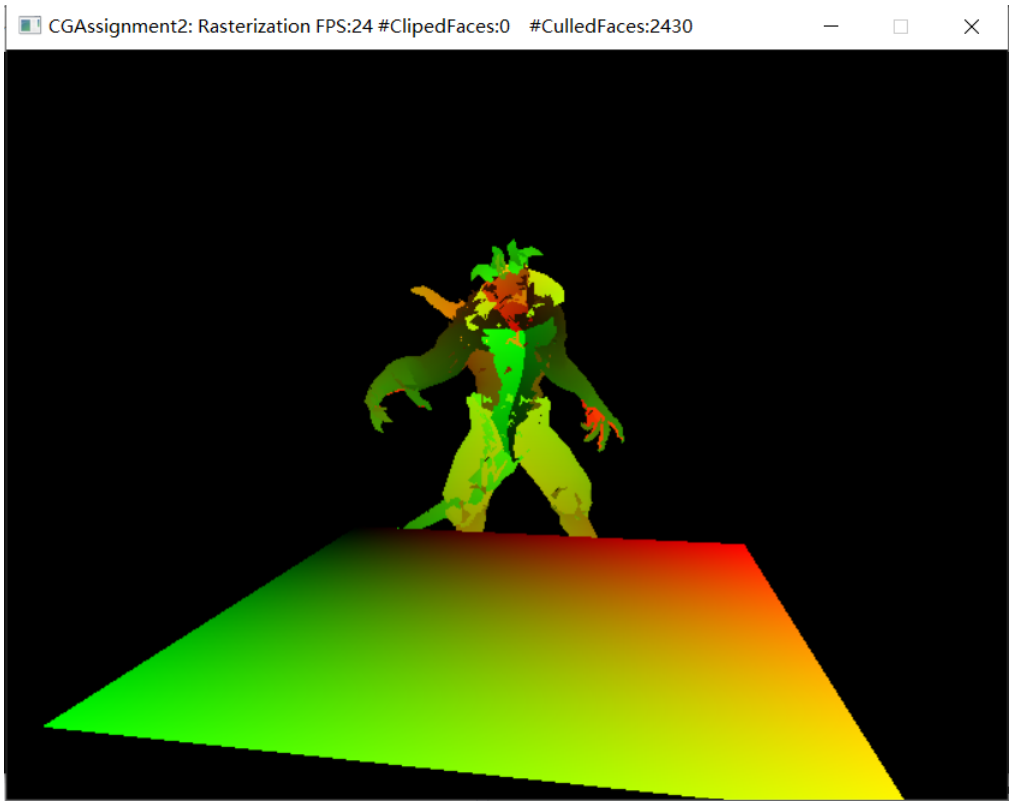

图 15: Task5 的效果

2.6.2 代码实现

```
if(points.cpos.z < m_backBuffer->readDepth(points.spos.x ,
    points.spos.y))
{
  //Perspective correction after rasterization
  TRShaderPipeline::VertexData::aftPrespCorrection(points);
  glm::vec4 fragColor;
  m_shader_handler->fragmentShader(points, fragColor);
  m_backBuffer->writeColor(points.spos.x, points.spos.y,
      fragColor);
  m_backBuffer->writeDepth(points.spos.x, points.spos.y,
      points.cpos.z);
}
```

2.7 Task6 遇到的问题、困难和体会

2.7.1 问题一：渲染的图像出现裂痕

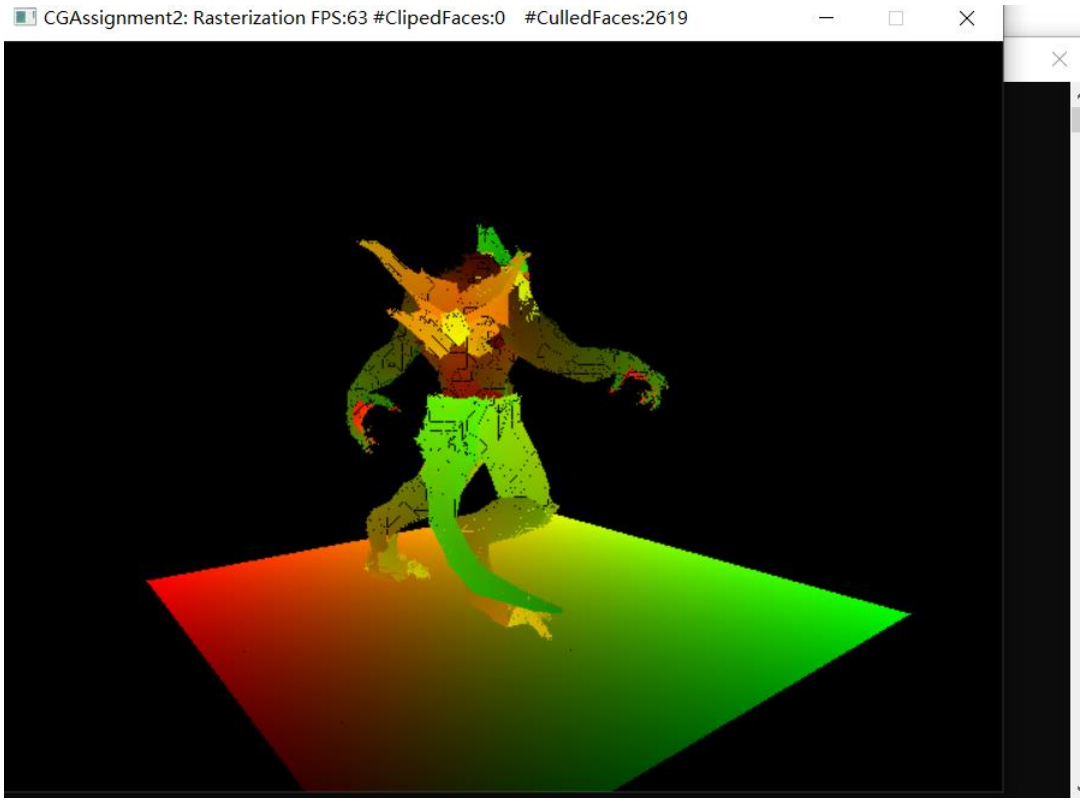

图 16: 渲染的图像出现裂痕

定位问题：没有考虑边界情况，也就是叉乘等于 0 的情况。将大于号改为大于等于号后，问题解决。

```
bool f0 = (glm::cross((point - glm::vec3(v0, 1.0f)), e0).z >= 0);
bool f1 = glm::cross((point - glm::vec3(v1, 1.0f)), e1).z >= 0;
bool f2 = glm::cross((point - glm::vec3(v2, 1.0f)), e2).z >= 0;
```

图 17: 渲染的图像出现裂痕

2.7.2 问题二：渲染的帧率低，只有 2fps

定位问题：忘记开启 release 模式，只使用了 debug 模式。开启 release 模式问题解决。

2.7.3 体会

在本次作业中，我参考资料，独立地编写代码，解决问题，最终达到了理想的作业效果，甚至还帮助周围的同学解决了不少问题。图形学的作业永远都是给人一种逐步探索的感觉，就一步步地做完了在没有选这门课的同学眼里很酷的工作，让我感到很有成就感。

在作业的过程中，我遇到的以上两个问题，都是对细节的关注程度不够造成的。细节决定成败，之后我要更加关注细节。特别是第二个问题，本来非常蠢，但是却纠结了我一中午的时间，真是非常可惜。