Assignment 2：Rasterization & Z-buffering

学号：17343124

姓名：伍斌

Task 1

实现Bresenham直线光栅化算法（你应该要用到线性插值函数 VertexData::lerp ）。

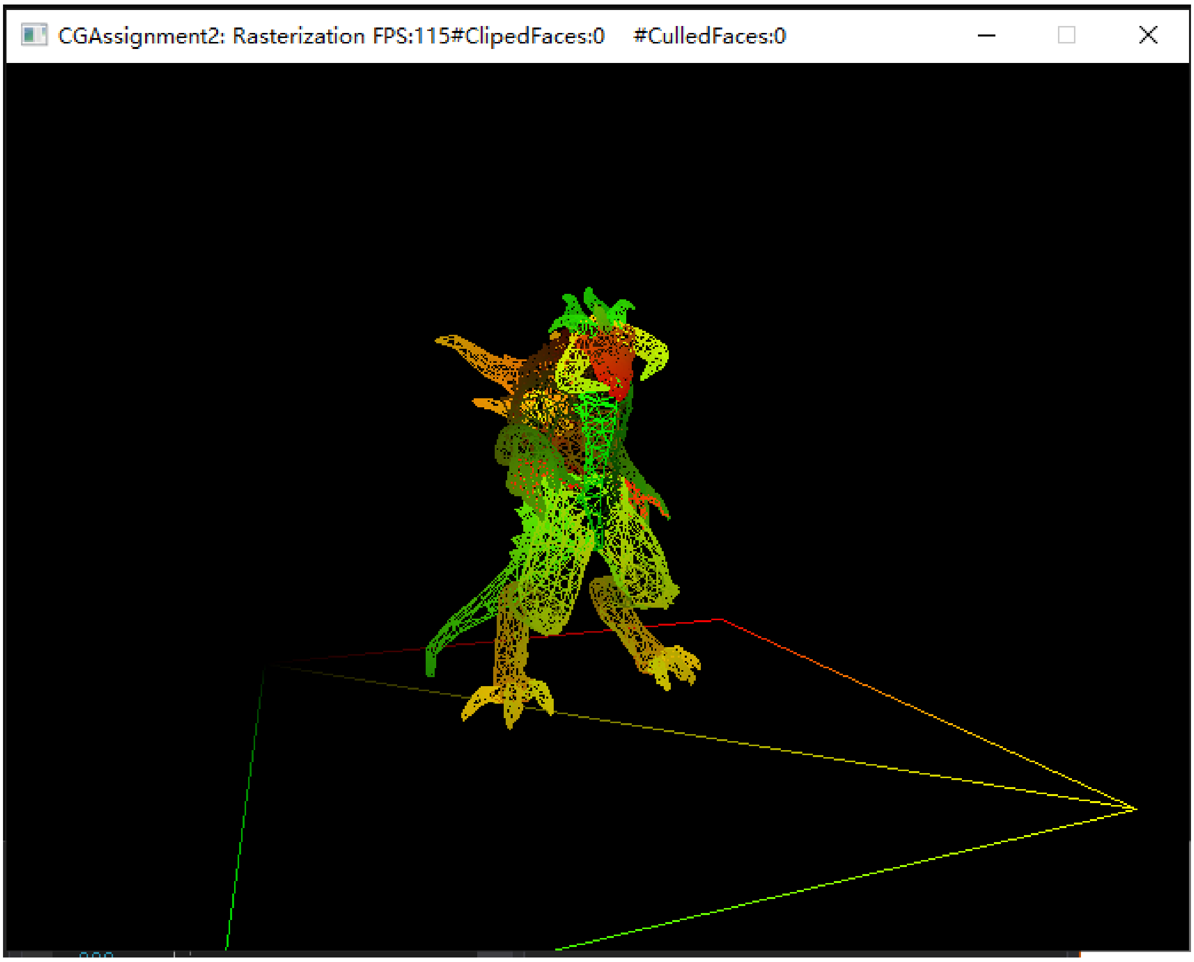
代码

```
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)
{

        //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.
    int dx = to.spos.x - from.spos.x;
    int dy = to.spos.y - from.spos.y;
    int stepX = 1, stepY = 1;

    // judge the sign
    if (dx < 0)
    {
        stepX = -1;
        dx = -dx;
    }
    if (dy < 0)
    {
        stepY = -1;
        dy = -dy;
    }
```

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

    if (dy <= dx)
    {
        int flag = d2y - dx;
        for (int i = 0; i <= dx; ++i)
        {
            auto mid = VertexData::lerp(from, to, static_cast<float>(i) / dx);
            mid.spos = glm::vec4(sx, sy, 0.0f, 1.0f);
            if (mid.spos.x >= 0 && mid.spos.x < screen_width && mid.spos.y >= 0
&& mid.spos.y < screen_height)
            {
                rasterized_points.push_back(mid);
            }
            sx += stepX;
            if (flag <= 0)
            {
                flag += d2y;
            }
            else
            {
                sy += stepY;
                flag += d2y_minus_d2x;
            }
        }
    }
    // slope > 1.
    else
    {
        int flag = d2x - dy;
        for (int i = 0; i <= dy; ++i)
        {
            auto mid = VertexData::lerp(from, to, static_cast<float>(i) / dy);
            mid.spos = glm::vec4(sx, sy, 0.0f, 1.0f);
            if (mid.spos.x >= 0 && mid.spos.x < screen_width && mid.spos.y >= 0
&& mid.spos.y < screen_height)
            {
                rasterized_points.push_back(mid);
            }
            sy += stepY;
            if (flag <= 0)
            {
                flag += d2x;
            }
            else
            {
                sx += stepX;
                flag -= d2y_minus_d2x;
            }
        }
    }
    //For instance:
    rasterized_points.push_back(from);
    rasterized_points.push_back(to);
```

```
}
```

实验结果



实现思路

设线段方程:

$$ax + by + c = 0(x1 < x < x2, y1 < y < y2)$$

令

$$dx = x2 - x1, dy = y2 - y1$$

则斜率为

$$-a/b = dy/dx$$

从第一个点开始，我们有

$$F(x, 1, y1) = a * x1 + b * y1 + c = 0$$

下面求线段ax+by+c=0与x=x1+1的交点:
由

$$a * (x1 + 1) + b * y + c = 0,$$

求出交点坐标

$$y = (-c - a(x1 + 1))/b$$

所以交点与M的y坐标差值

$$Sub1 = (-c - a(x1+1))/b - (y1+0.5) = -a/b - 0.5$$

，即Sub1的处始值为-a/b-0.5。

则可得条件当 Sub1 = -a/b-0.5>0时候,即下个点为U.反之,下个点为B.
代入a/b,则

$$Sub1 = dy/dx - 0.5$$

因为是个循环中都要判断Sub,所以得求出循环下的Sub表达式,我们可以求出Sub的差值的表达式.下面求x=x1+2时的Sub，即Sub2：

1.如果下下个点是下个点的右上邻接点，则

$$Sub2 = (-c - a(x1+2))/b - (y1+1.5) = -2a/b - 1.5$$

故Sub差值

$$Dsub = Sub2 - Sub1 = -2a/b - 1.5 - (-a/b - 0.5) = -a/b - 1$$

.代入a/b得

$$Dsub = dy/dx - 1$$

2.如果下下个点是下个点的右邻接点,

$$Sub2 = (-c - a(x1+2))/b - (y1+0.5) = -2a/b - 0.5$$

故Sub差值

$$Dsub = Sub2 - Sub1 = -2a/b - 0.5 - (-a/b - 0.5) = -a/b$$

. 代入a/b得

$$Dsub = dy/dx$$

于是，我们有了Sub的处始值

$$Sub1 = -a/b - 0.5 = dy/dx - 0.5$$

，及Sub的差值的表达式

$$Dsub = dy/dx - 1\text{（当}Sub1 > 0\text{）或}dy/dx\text{（当}Sub1 < 0\text{）}$$

伪代码如下：

```
x=x1;
y=y1;
dx = x2-x1;
dy = y2-y1;
Sub = dy/dx-0.5; // 赋初值，下个要画的点与中点的差值
DrawPixel(x, y); // 画起点

while(x<x2)
{
  x++;
  if(Sub > 0) // 下个要画的点为当前点的右上邻接点
  {
    Sub += dy/dx - 1; //下下个要画的点与中点的差值
```

```
      y++; // 右上邻接点y需增1
    }
    else// 下个要画的点为当前点的右邻接点
    {
      Sub += dy/dx;
    }
    // 画下个点
    DrawPixel(x,y);
  }
```

Task2

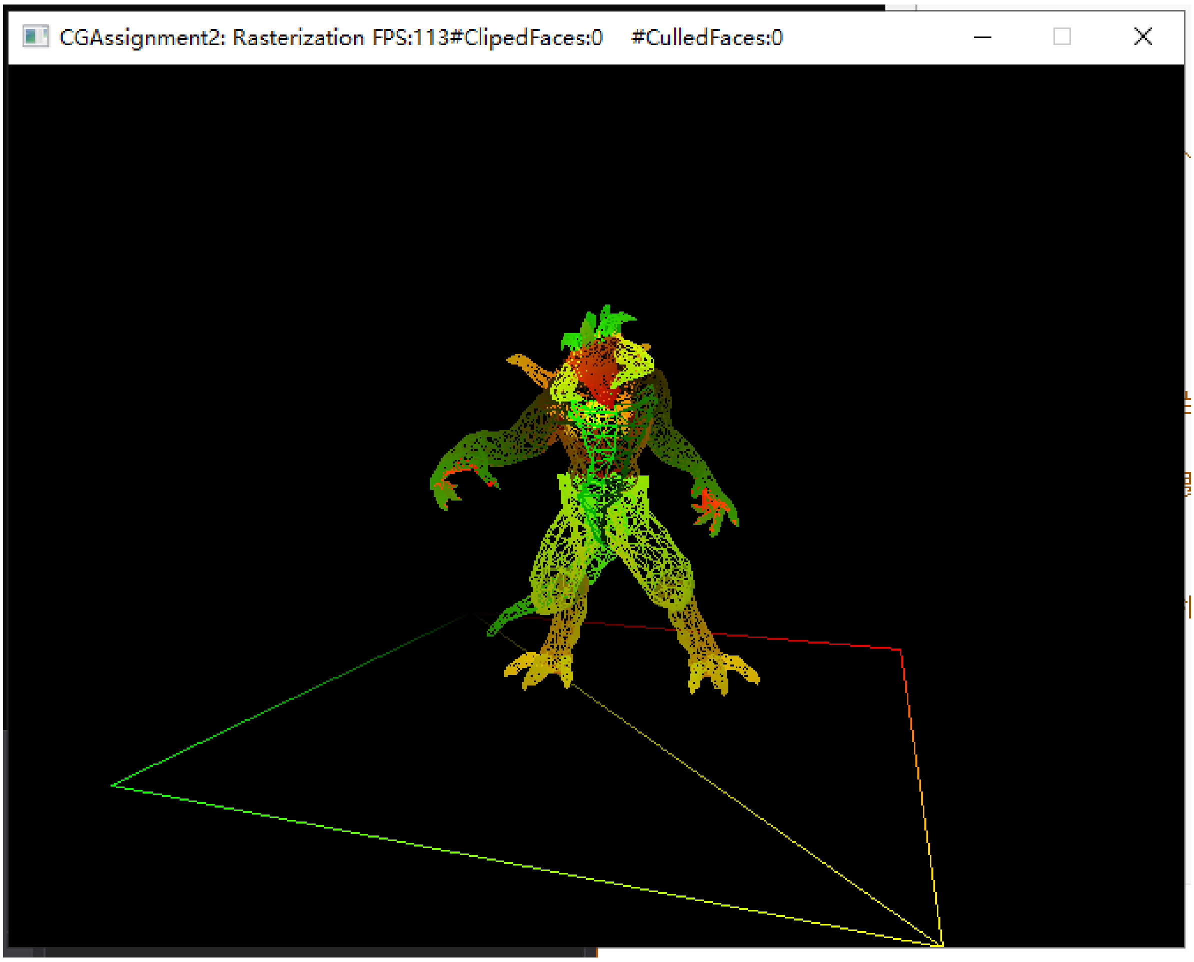
实现简单的齐次空间裁剪，简述你是怎么做的。

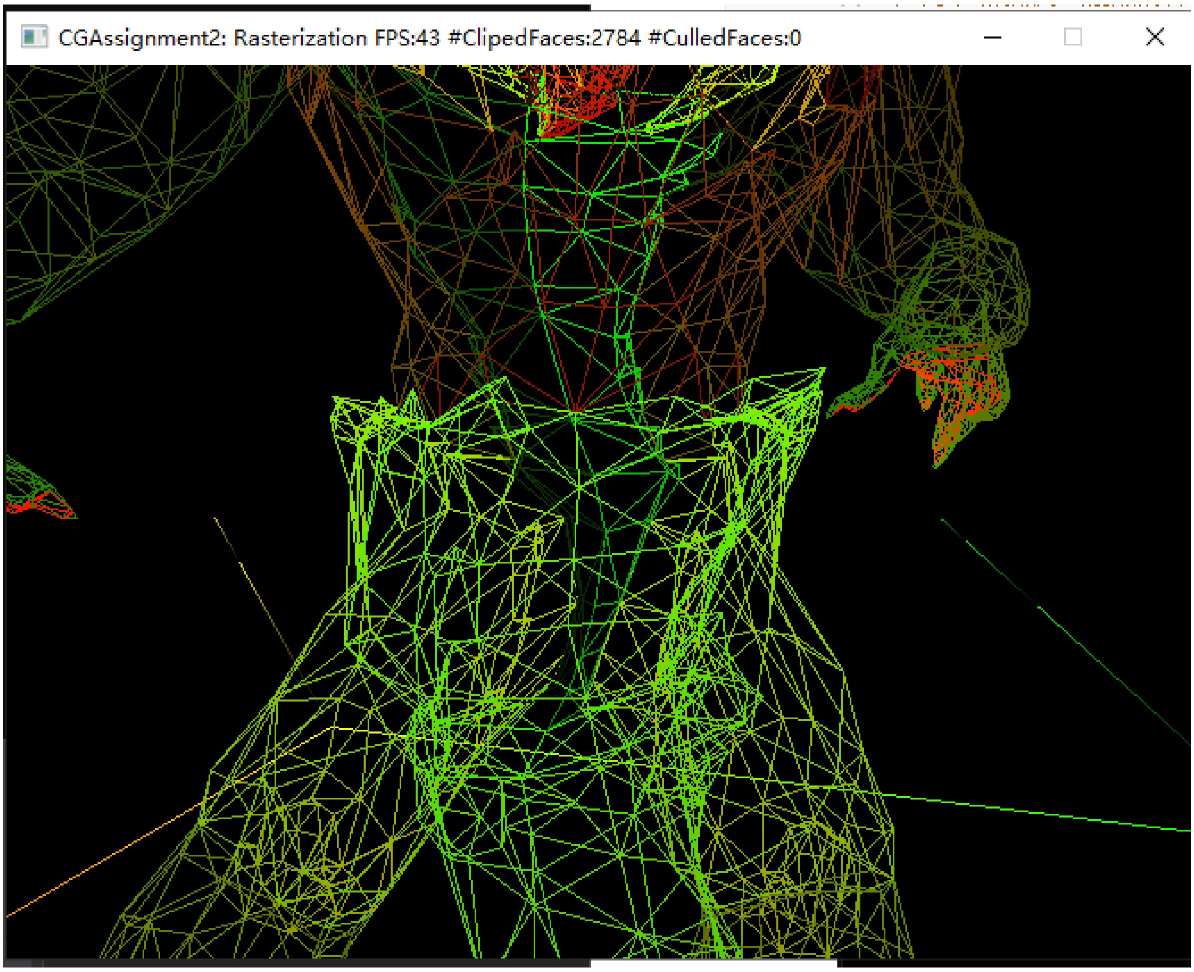
代码

```
    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

        if (v0.cpos.w < m_frustum_near_far.x && v1.cpos.w < m_frustum_near_far.x && v2.cpos.w < m_frustum_near_far.x)
            return {};
        if (v0.cpos.w > m_frustum_near_far.y && v1.cpos.w > m_frustum_near_far.y && v2.cpos.w > m_frustum_near_far.y)
            return {};
        if (v0.cpos.x > v0.cpos.w && v1.cpos.x > v1.cpos.w && v2.cpos.x > v2.cpos.w)
            return {};
        if (v0.cpos.x < -v0.cpos.w && v1.cpos.x < -v1.cpos.w && v2.cpos.x < -v2.cpos.w)
            return {};
        if (v0.cpos.y > v0.cpos.w && v1.cpos.y > v1.cpos.w && v2.cpos.y > v2.cpos.w)
            return {};
        if (v0.cpos.y < -v0.cpos.w && v1.cpos.y < -v1.cpos.w && v2.cpos.y < -v2.cpos.w)
            return {};
        if (v0.cpos.z > v0.cpos.w && v1.cpos.z > v1.cpos.w && v2.cpos.z > v2.cpos.w)
            return {};
```

```
        if (v0.cpos.z < -v0.cpos.w && v1.cpos.z < -v1.cpos.w && v2.cpos.z < -
v2.cpos.w)
            return {};

        return { v0, v1, v2 };
    }
```

实现效果





　可以看见，在滑动鼠标滚轮将镜头拉近，将一些三角形推出屏幕之外后，窗口标题的 #ClipedFaces 变大。

实现思路

　　透视投影之后透视除法之前的坐标空间被称为裁剪空间，也叫齐次（裁剪）空间，它实质上是一个四维空间，变换到齐次空间的顶点之间仍然是线性相关的（可以直接使用线性插值而不是透视插值）。视锥体中的点，都满足如下条件：

$$-w \leq x, y, z \leq w \text{和} near \leq w \leq far$$

　　如果不满足这个条件的点，理论上就应该剔除或者裁剪。

　　由上式可以得到六个裁剪平面：

- 左 x+w >= 0
- 右 -x+w >= 0
- 上 y+w >= 0
- 下 -y+w >=0
- 前 z+w >=0
- 后 -z+w >=0

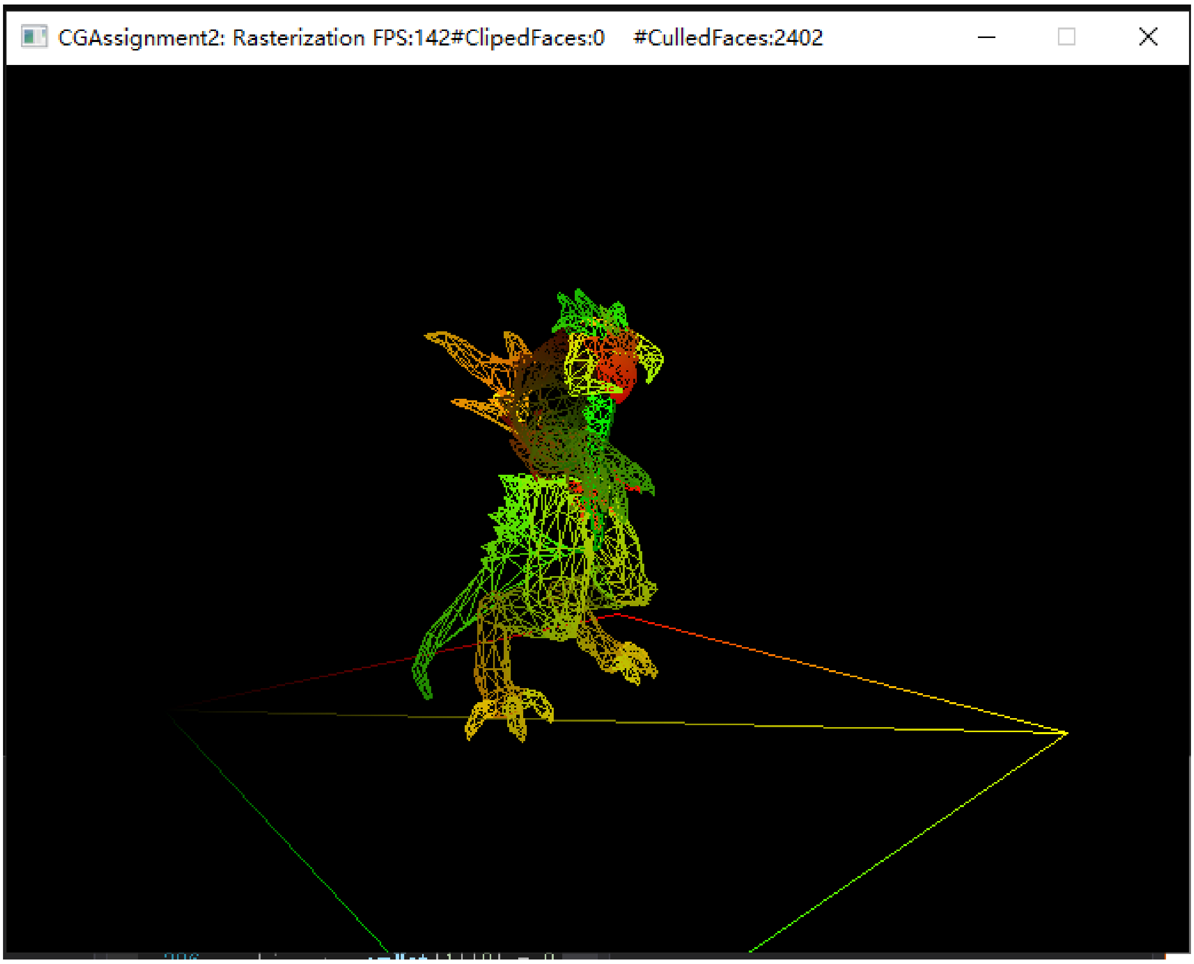
Task3

实现三角形的背向面剔除，简述你是怎么做的。

代码

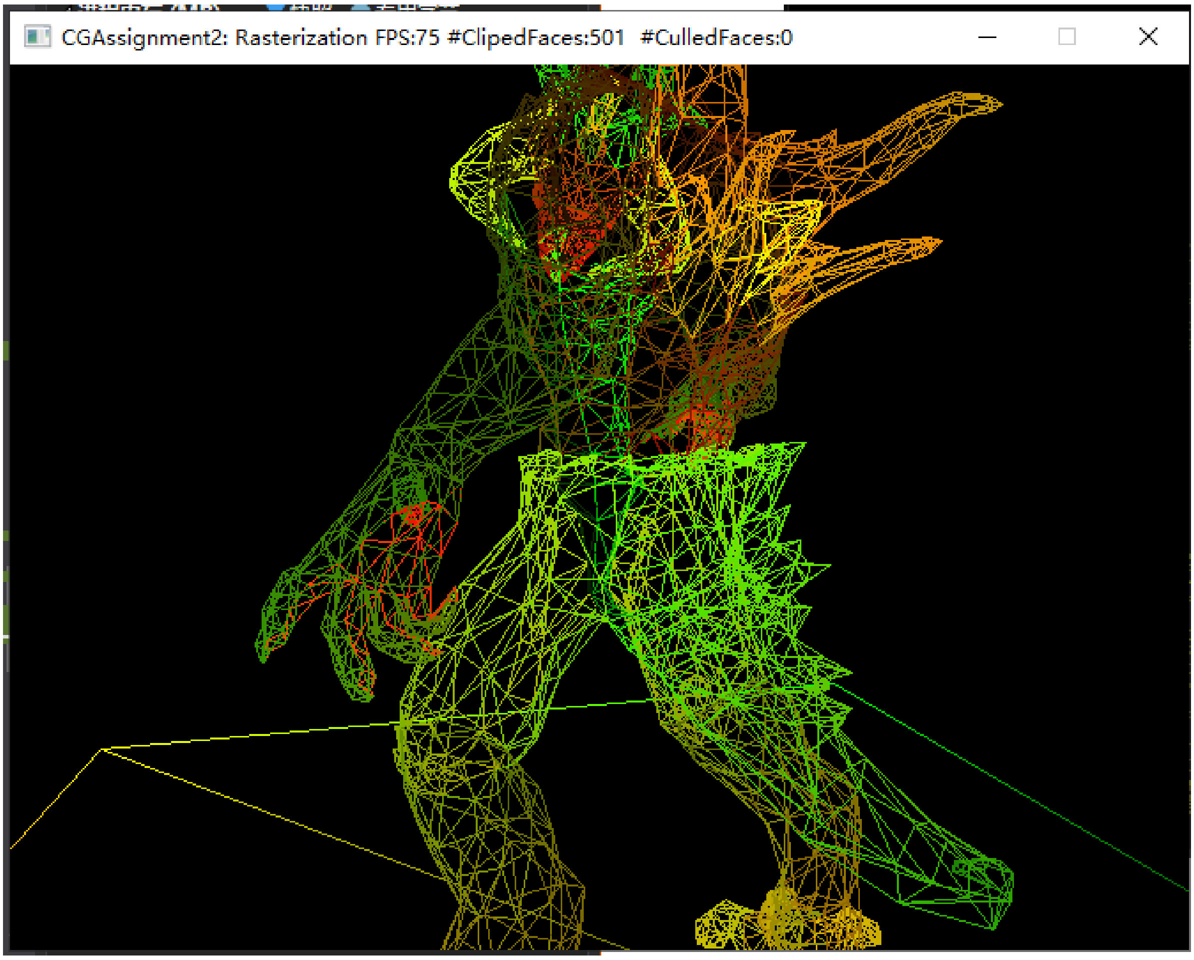
```
    bool TRRenderer::isTowardBackFace(const glm::vec4 &v0, const glm::vec4 &v1,
const glm::vec4 &v2) const
    {
        //Back face culling in the ndc space

        // Task3: Implement the back face culling
        //  Note: Return true if it's a back-face, otherwise return false.

        glm::vec3 tmp1 = glm::vec3(v1.x - v0.x, v1.y - v0.y, v1.z - v0.z);
        glm::vec3 tmp2 = glm::vec3(v2.x - v0.x, v2.y - v0.y, v2.z - v0.z);

        //叉乘得到法向量
        glm::vec3 normal = glm::normalize(glm::cross(tmp1, tmp2));
        //glm::vec3 view = glm::normalize(glm::vec3(v1.x - camera->Position.x,
v1.y - camera->Position.y, v1.z - camera->Position.z));
        //NDC中观察方向指向+z
        glm::vec3 view = glm::vec3(0, 0, -1);
        if (glm::dot(normal, view) > 0)
            return true;
        else
            return false;
    }
```

实现效果

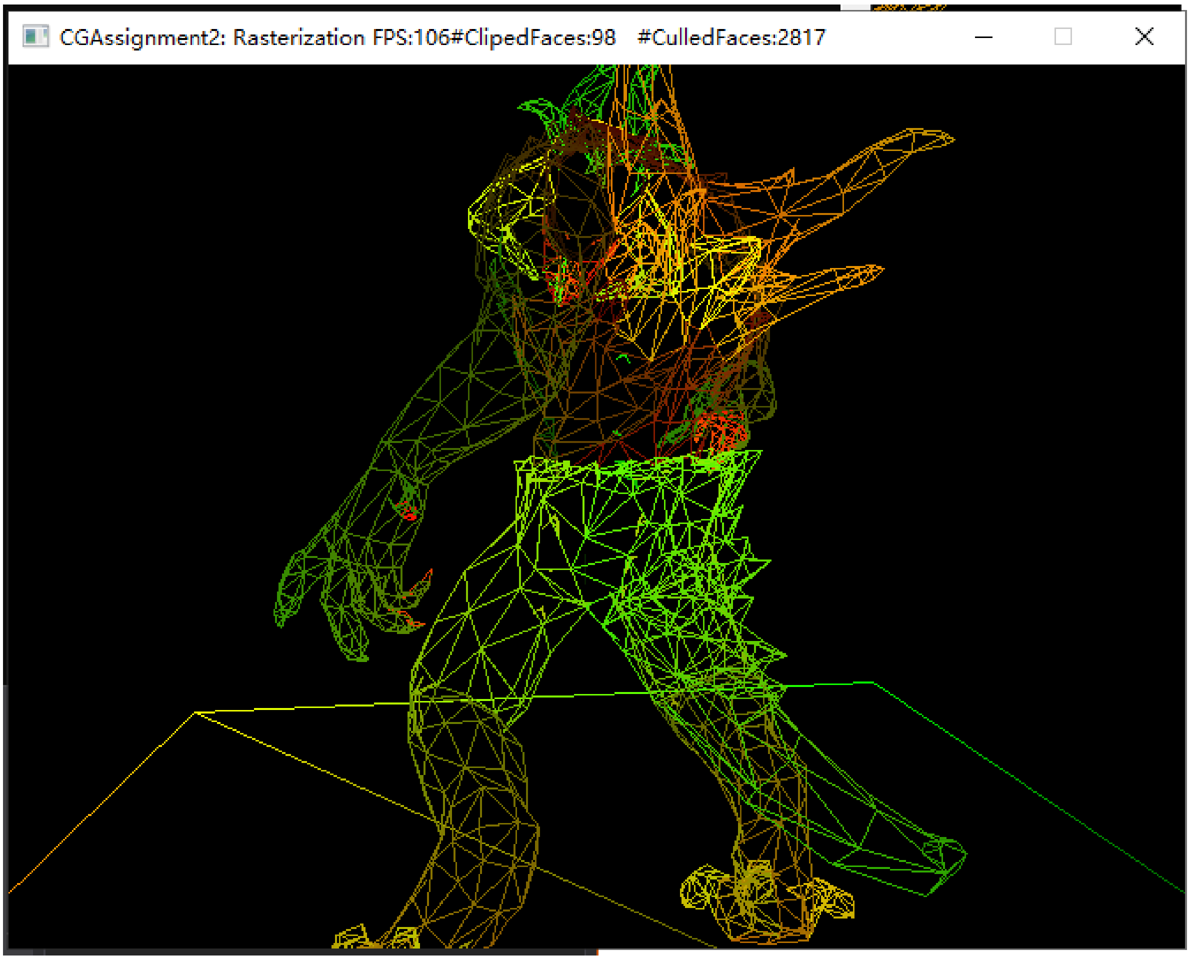


与之前的实验图对比可以看出窗口标题的 #CulledFaces 值增大。

剔除前：



剔除后：



显然，可以看出面剔除后背面的连线不再显示。

实现思路

我们将在ndc空间做三角形的背向面剔除，以逆时针环绕顺序为正面。在ndc空间，三角形的顶点坐标的 x 、 y 和 z 取值在 [-1,1] 之间，摄像机在坐标原点 (0,0,0) 处，朝向 (0,0,-1) 方向（右手坐标系）。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，

Task4

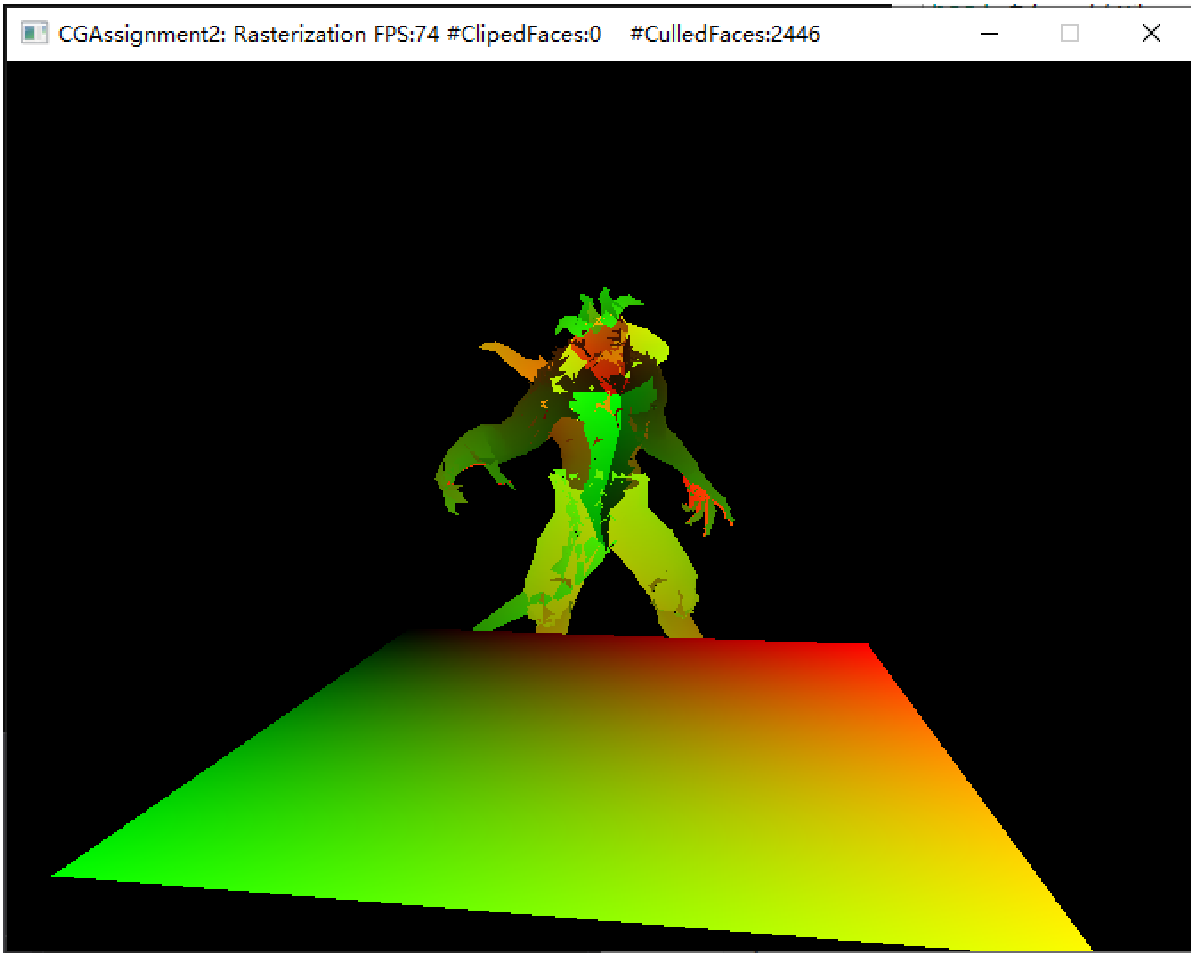
实现基于Edge-function的三角形填充算法。

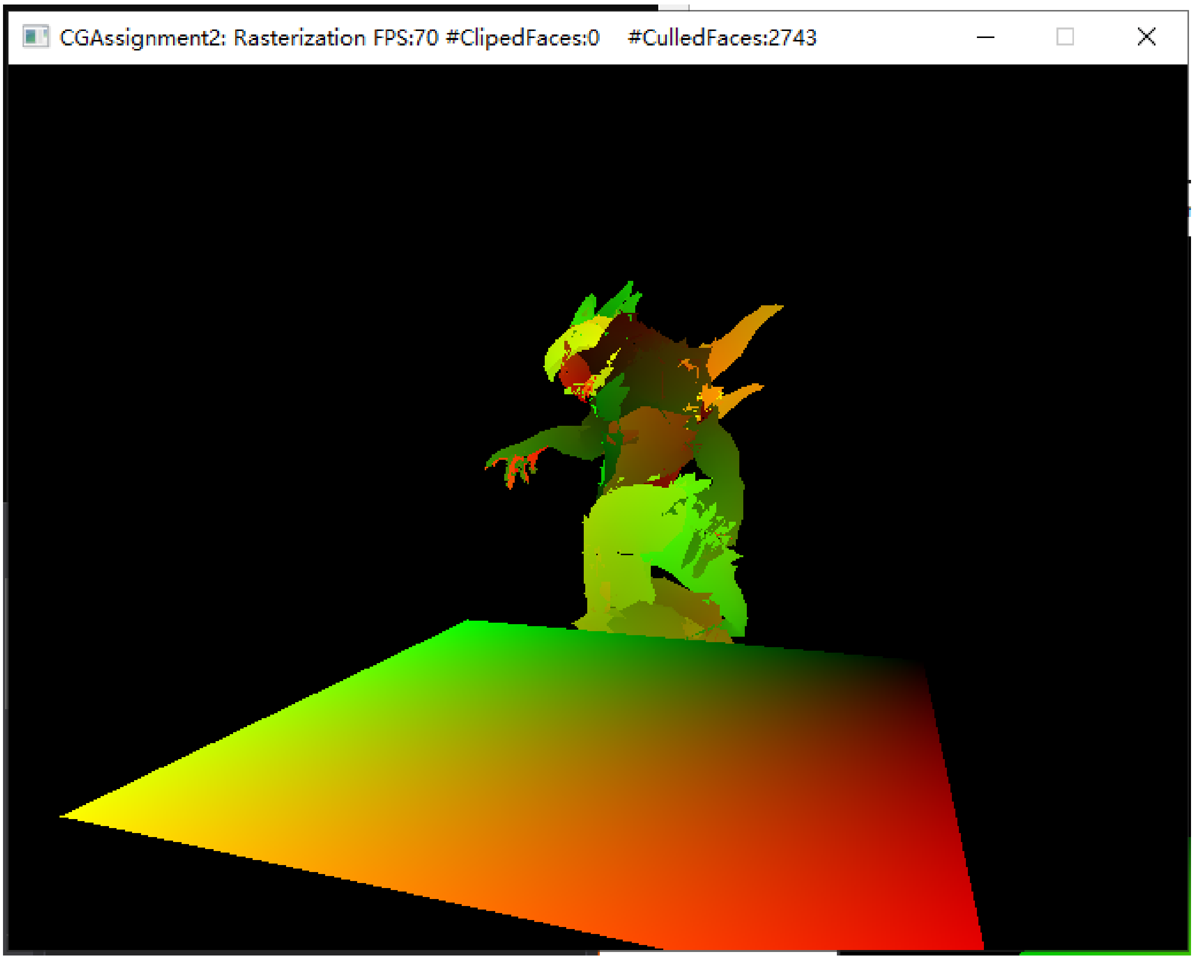
代码

```
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)
{
    //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.

    //求出三角形包围盒
    VertexData v[3] = {v0, v1, v2};
    double minX, maxX, minY, maxY;
    minX = maxX = v0.spos.x;
    minY = maxY = v0.spos.y;
    for (int i = 0; i < 3; i++) {
        minX = std::min(minX, (double)v[i].spos.x);
        maxX = std::max(maxX, (double)v[i].spos.x);

        minY = std::min(minY, (double)v[i].spos.y);
        maxY = std::max(maxY, (double)v[i].spos.y);
    }
    //遍历包围盒中所有点
    for (int ix = minX; ix <= maxX; ix++) {
        for (int iy = minY; iy <= maxY; iy++) {
            //普通光栅化
            float x = ix + 1 / 2;
            float y = iy + 1 / 2;
            //计算三角形重心
            glm::vec3 s1 = glm::vec3(v1.spos.x - v0.spos.x, v2.spos.x -
v0.spos.x, v0.spos.x - x);
```

```
            glm::vec3 s2 = glm::vec3(v1.spos.y - v0.spos.y, v2.spos.y - 
v0.spos.y, v0.spos.y - y);
            glm::vec3 u = glm::cross(s1, s2);
            //判断点是否在三角形内
            float x1 = v0.spos.x;
            float x2 = v1.spos.x;
            float x3 = v2.spos.x;
            float y1 = v0.spos.y;
            float y2 = v1.spos.y;
            float y3 = v2.spos.y;
            bool f1 = ((y - y1) * (x2 - x1) - (y2 - y1) * (x - x1) > 0);
            bool f2 = ((x3 - x2) * (y - y2) - (y3 - y2) * (x - x2) > 0);
            bool f3 = ((x1 - x3) * (y - y3) - (y1 - y3) * (x - x3) > 0);
            //三角形内：判断三个叉积的结果是否同号
            if (f1 == f2 && f2 == f3) {
                //插值计算,插值权重(1-u-v,u,v)即(1-(u.x+u.y)/u.z, u.x/u.z, u.y/u.z)
                glm::vec3 k;
                k.x = 1 - (u.x + u.y) / u.z;
                k.y = u.x / u.z;
                k.z = u.y / u.z;
                auto mid = VertexData::barycentricLerp(v0, v1, v2, k);
                mid.spos = glm::ivec2(ix, iy);//这是浮点数误差导致的孔洞
                //判断点是否在屏幕范围内
                if (mid.spos.x >= 0 && mid.spos.x < screen_width && mid.spos.y 
>= 0 && mid.spos.y < screene_height)
                {
                    rasterized_points.push_back(mid);
                }
            }
        }
    }
    //For instance:
    rasterized_points.push_back(v0);
    rasterized_points.push_back(v1);
    rasterized_points.push_back(v2);
}
```

实现效果


CGAssignment2: Rasterization FPS:74 #ClipedFaces:0 #CulledFaces:2446


CGAssignment2: Rasterization FPS:70 #ClipedFaces:0 #CulledFaces:2743

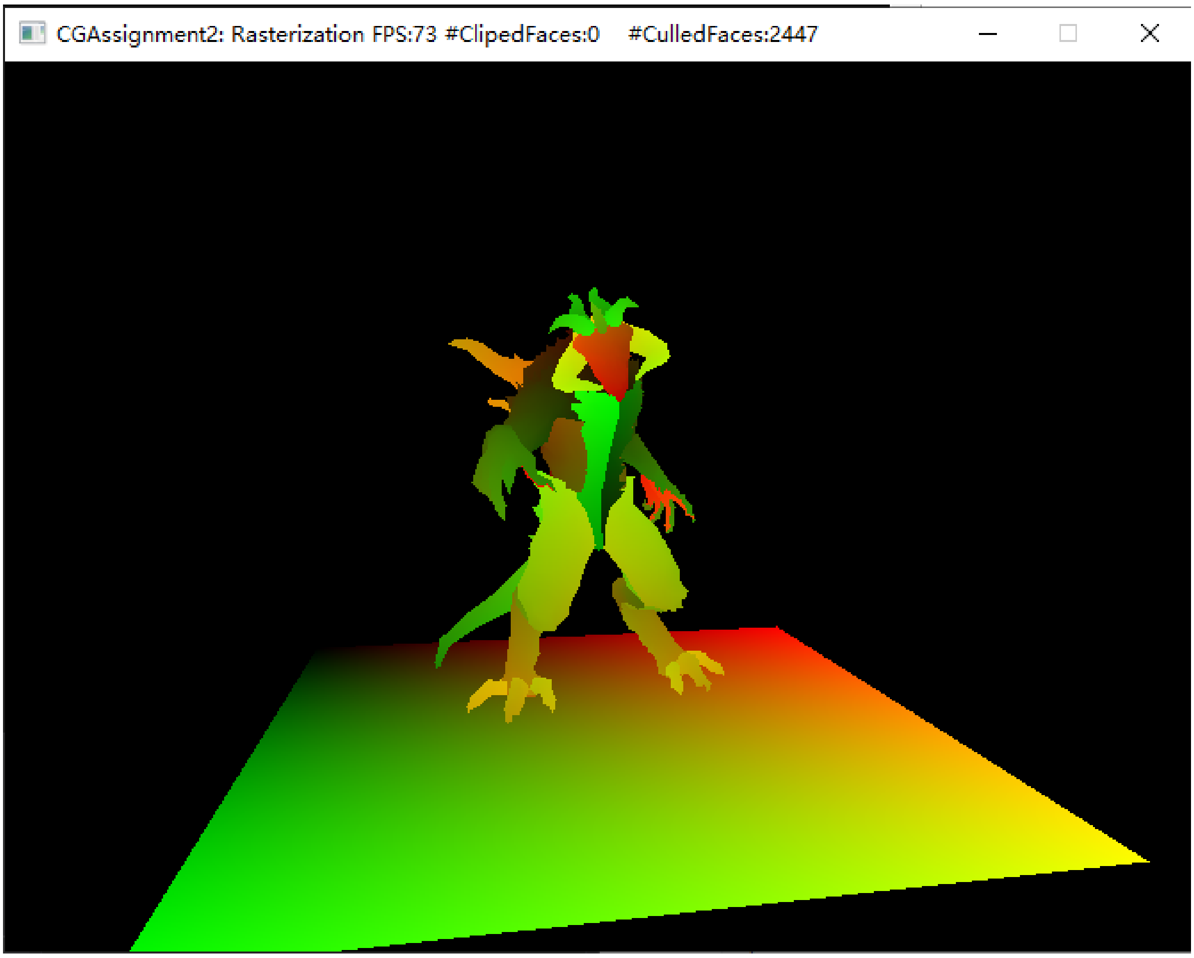
Task5

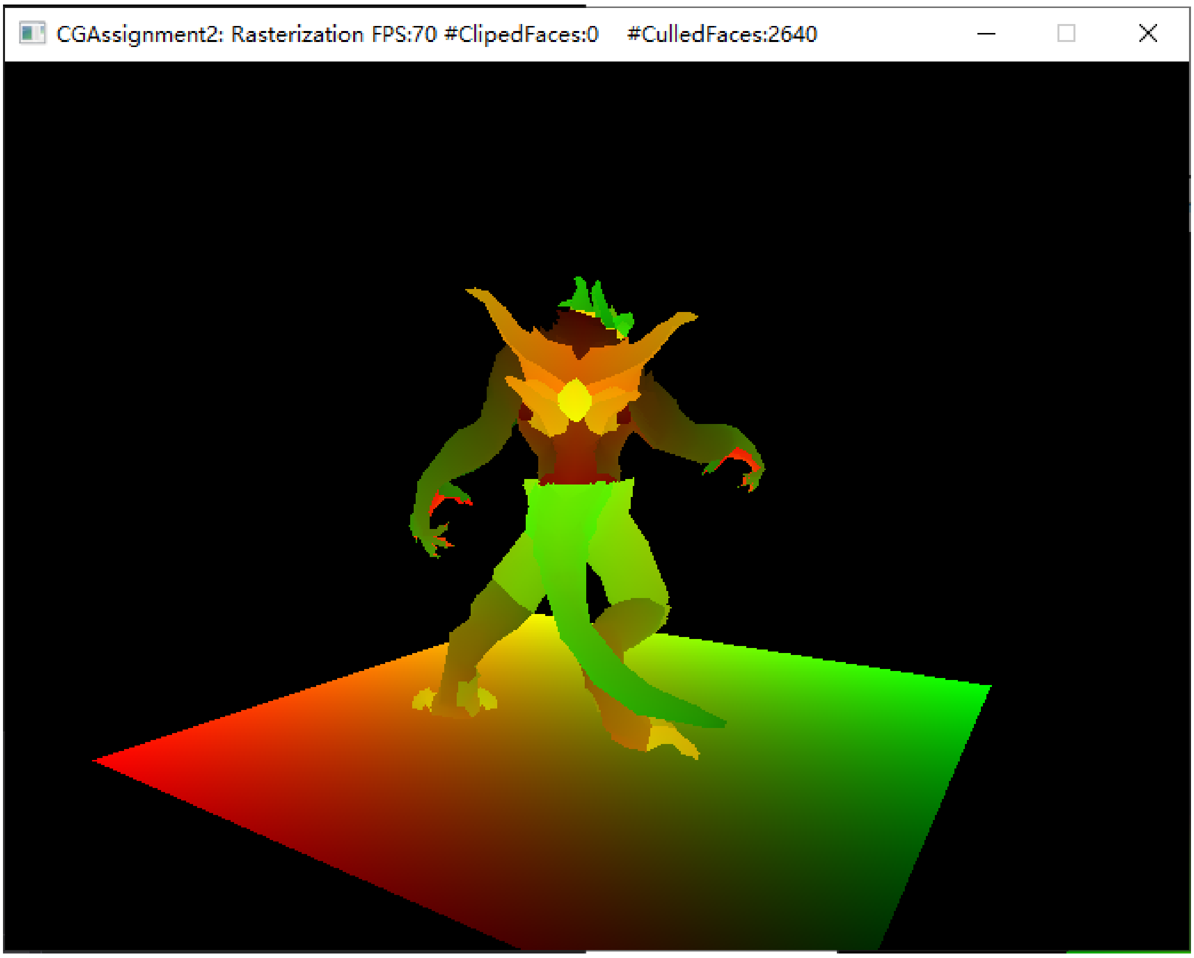
实现深度测试。

代码

```
for (auto &points : rasterized_points)
{
//Task5: Implement depth testing here
// Note: You should use m_backBuffer->readDepth() and points.spos to read the
depth in buffer
//       points.cpos.z is the depth of current fragment
    if (m_backBuffer->readDepth(points.spos.x, points.spos.y) > points.cpos.z)
//判断深度插值是否大于像素点的深度值，否则跳过
    {
        //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);
    }
}
```

实现效果





Task6

实现了直线光栅化、三角形填充光栅化、齐次空间简单裁剪、背面剔除之后，谈谈你遇到的问题、困难和体会。

1. 在实现其次空间简单裁剪时理解了cpos和spos的区别。
2. 在实现背面剔除时一开始剔除了正面。
3. 在实现三角形光栅化时遇到了非常多的问题：

   (1) 浮点数误差导致的孔洞（这个问题请教了助教）

   (2) 判断点是否在三角形中（没有正确理解同号的意义，这个看了很久数学推算）

在解决完这些问题之后，我对图形渲染管线和光栅化的过程有了更深刻的理解，同时也养成了每一步都写注释的好习惯。