HW2

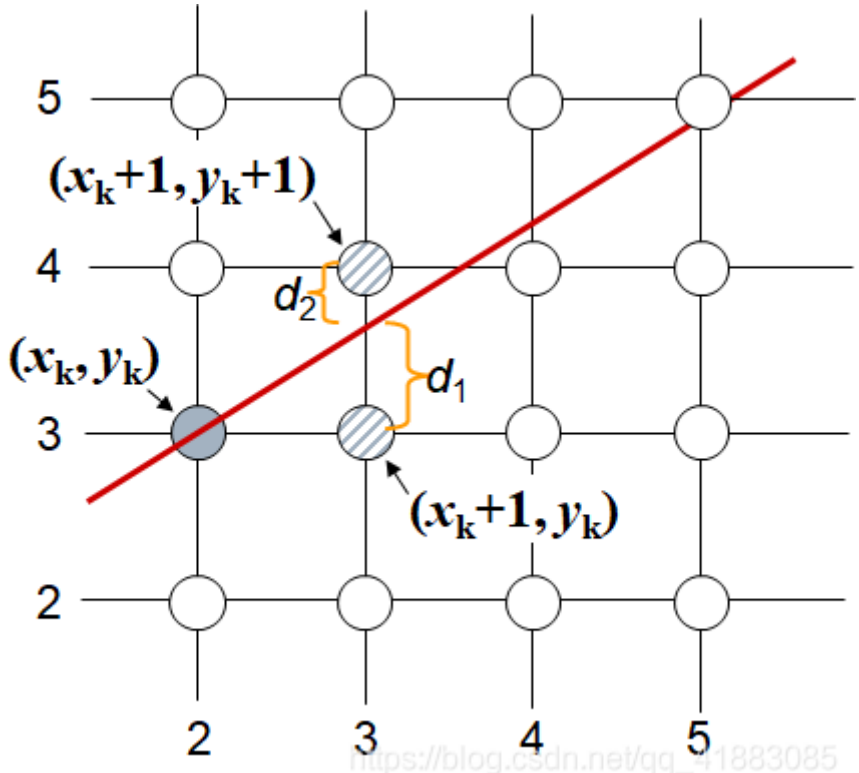
姓名	学号	专业
崔璨明	20337025	计算机科学与技术

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 ）该函数在 TRShaderPipeline.cpp 文件中，其中的 from 和 to 参数分别代表直线的起点和终点，screen_width 和 screen_height 是窗口的宽高，超出窗口的点应该被丢弃。 rasterized_points 存放光栅化插值得到的点。

答：Bresenham直线光栅化算法是由Bresenham提出的一种精确而有效的光栅线生成算法，可用于显示线、圆和其他曲线的整数增量运算。

算法从(x_k, y_k)开始，每次让x加一，然后根据直线方程计算相应的y，再选择离y最近的像素点，以下图为例，设直线方程为$y = kx + b$，则$y_k = kx_k + 1$



当x+1时：

$$x_{k+1} = x_k + 1$$

$$y' = kx_{k+1} + b = k(x_k + 1) + b$$

然后计算离y'最近的像素点：

$$d_1 = y' - y_k = k(x_k + 1) + b - y_k$$

$$d_2 = y_k + 1 - y' = y_k + 1 - k(x_k + 1) - b$$

若$d_1 > d_2$，即$d_1 - d_2 > 0$，则取$y_{k+1} = y_k + 1$，否则取$y_{k+1} = y_k$:

$$d_1 - d_2 = 2k(x_k + 1) - 2y_k + 2b - 1$$

上式中的k是直线的斜率，因此将上式作为判断标准需要做非常昂贵的浮点数除法运算。为了消去除法，注意到$k = \frac{\Delta y}{\Delta x}$，两边同时乘上$\Delta x > 0$，正负符号不变:

$$p_k = \Delta x \cdot (d_1 - d_2) = 2\Delta y \cdot (x_{k+1}) - 2\Delta x \cdot y_k + (2b - 1)\Delta x$$

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

$$where \quad c = (2b - 1)\Delta x + 2\Delta y$$

可以利用 p_k 的符号作为选取标准，另外有

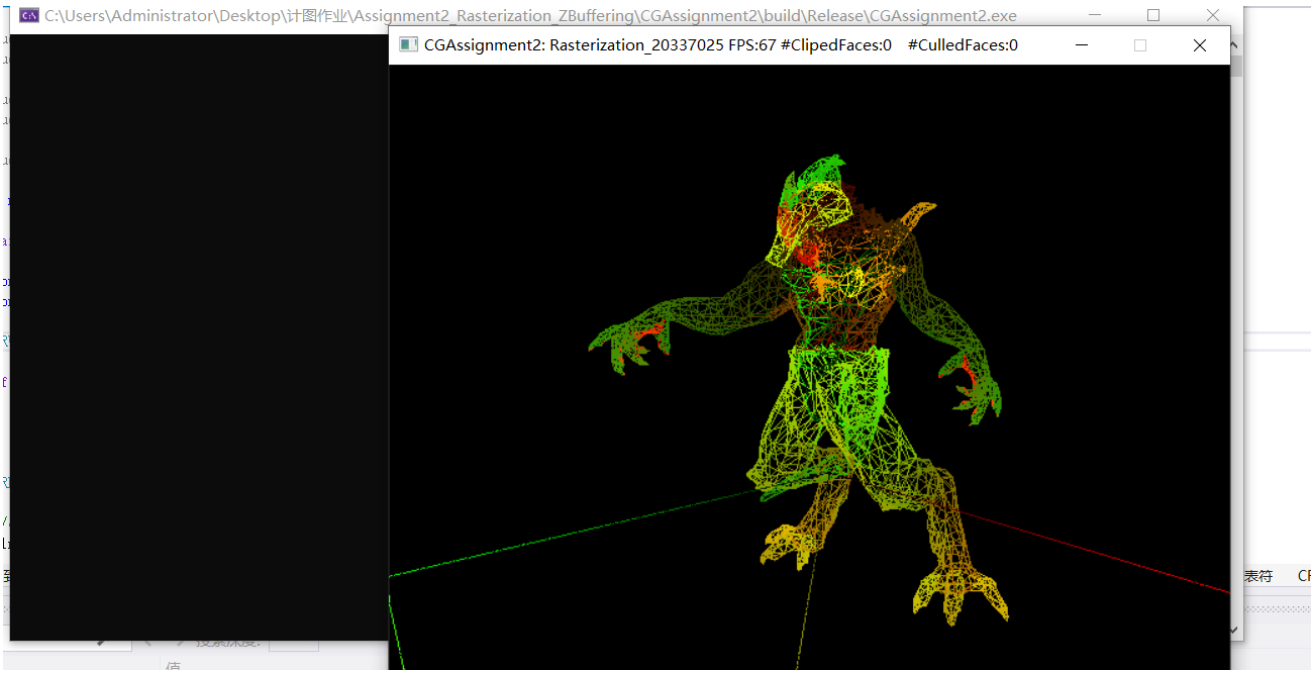
$$p_{k+1} - p_k = (2\Delta y \cdot x_k + 1 - 2\Delta x \cdot y_{k+1} + c) - (2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c) = 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

如果 $p_k <= 0$，则 $y_{k+1} = y_k$，有 $p_{k+1} = p_k + 2\Delta y$；如果 $p_k > 0$，则 $y_{k+1} = y_k + 1$，有 $p_{k+1} = p_k + 2\Delta y - 2\Delta x$。
根据以上流程如此迭代下去即可。
编写代码如下:

```
int dx = to.spos.x - from.spos.x;
int dy = to.spos.y - from.spos.y;
int stepX = 1, stepY = 1;
// 判断符号
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;
// slope k <= 1
if (dy <= dx) {
        int flag = d2y - dx;
        for (int i = 0; i <= dx; i++) {
                // linear interpolation
                VertexData tmp = VertexData::lerp(from, to, static_cast<double>
(i) / dx);
                // 判断是否在屏幕内,位于窗口外，则丢弃这些插值点。
                if (tmp.spos.x >= 0 && tmp.spos.x < screen_width
                        && tmp.spos.y >= 0 && tmp.spos.y < screen_height) {
                        rasterized_points.push_back(tmp);//存放插值点
                }
```

```
                sx += stepX;
                if (flag <= 0) {
                        flag += d2y;
                }
                else {
                        sy += stepY;
                        flag += d2y_minus_d2x;
                }
        }
}
//slope k > 1
else {
        int flag = d2x - dy;
        for (int i = 0; i <= dy; i++) {
                VertexData tmp = VertexData::lerp(from, to, static_cast<double>
(i) / dy);
                if (tmp.spos.x >= 0 && tmp.spos.x < screen_width
                        && tmp.spos.y >= 0 && tmp.spos.y < screen_height) {
                        rasterized_points.push_back(tmp);
                }
                sy += stepY;
                if (flag <= 0) {
                        flag += d2x;
                }
                else {
                        sx += stepX;
                        flag += d2y_minus_d2x;
                }
        }
}
```

实现效果如下：



Task 2

实现简单的齐次空间裁剪，简述你是怎么做的。
该函数在 TRRenderer.cpp 文件中，输入的 v0 、 v1 和 v2 是三角形的三个顶点，其中的 v0.cpos 、v1.cpos 和 v2.cpos 分别存储在齐次裁剪空间的顶点坐标（注意这是个四维齐次坐标）。如果顶点坐标在可见的视锥体之内的话，那么 x 、y 、z 的取值应该在 [-w,w] 之间，而 w 应该在 [near,far] 之间。（ near 和 far 是视锥体的近平面和远平面）

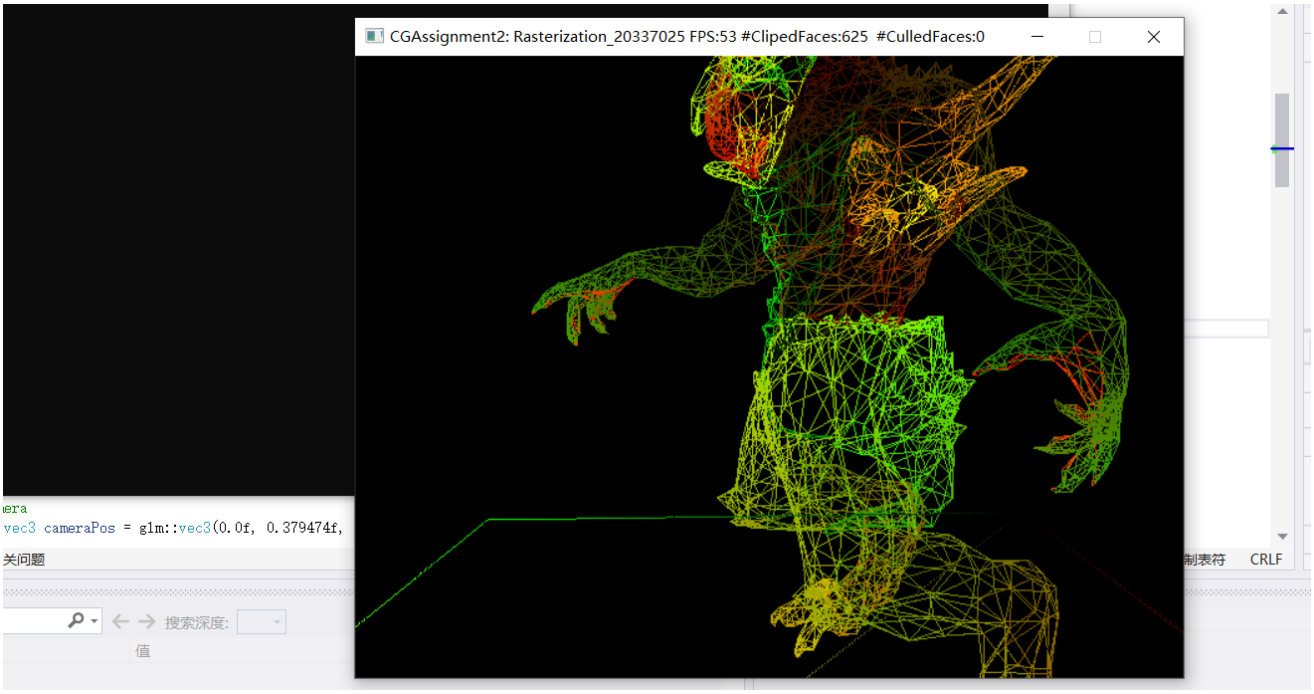
答：
齐次空间裁剪是指发生在执行顶点着色之后、透视除法之前的一段操作。其目标非常直接，因为我们并没有必要去渲染视锥体之外的物体，这样可以省去很多计算开销。

对传入的三角形的三个顶点进行判断即可，当有一个不在视锥体内，则将整个三角形除去。
编写代码如下：

```
//左
if (v0.cpos.x < -v0.cpos.w && v1.cpos.x < -v1.cpos.w && v2.cpos.x < -v2.cpos.w)
{
    return {};
}
//右
if (v0.cpos.x > v0.cpos.w && v1.cpos.x > v1.cpos.w && v2.cpos.x > v2.cpos.w) {
    return {};
}
//上
if (v0.cpos.y < -v0.cpos.w && v1.cpos.y < -v1.cpos.w && v2.cpos.y < -v2.cpos.w)
{
    return {};
}
//下
if (v0.cpos.y > v0.cpos.w && v1.cpos.y > v1.cpos.w && v2.cpos.y > v2.cpos.w) {
    return {};
}
//近
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 {};
}
//远
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 {};
}
//前
if (v0.cpos.z < -v0.cpos.w && v1.cpos.z < -v1.cpos.w && v2.cpos.z < -v2.cpos.w)
{
    return {};
}
```

```
//后
if (v0.cpos.z > v0.cpos.w && v1.cpos.z > v1.cpos.w && v2.cpos.z > v2.cpos.w) {
    return {};
}
return { v0, v1, v2 };
```

实现效果如下：



当滑动鼠标滚轮放大时，可以观察到ClipedFaces的值变大，且一直在动态变化。

Task 3

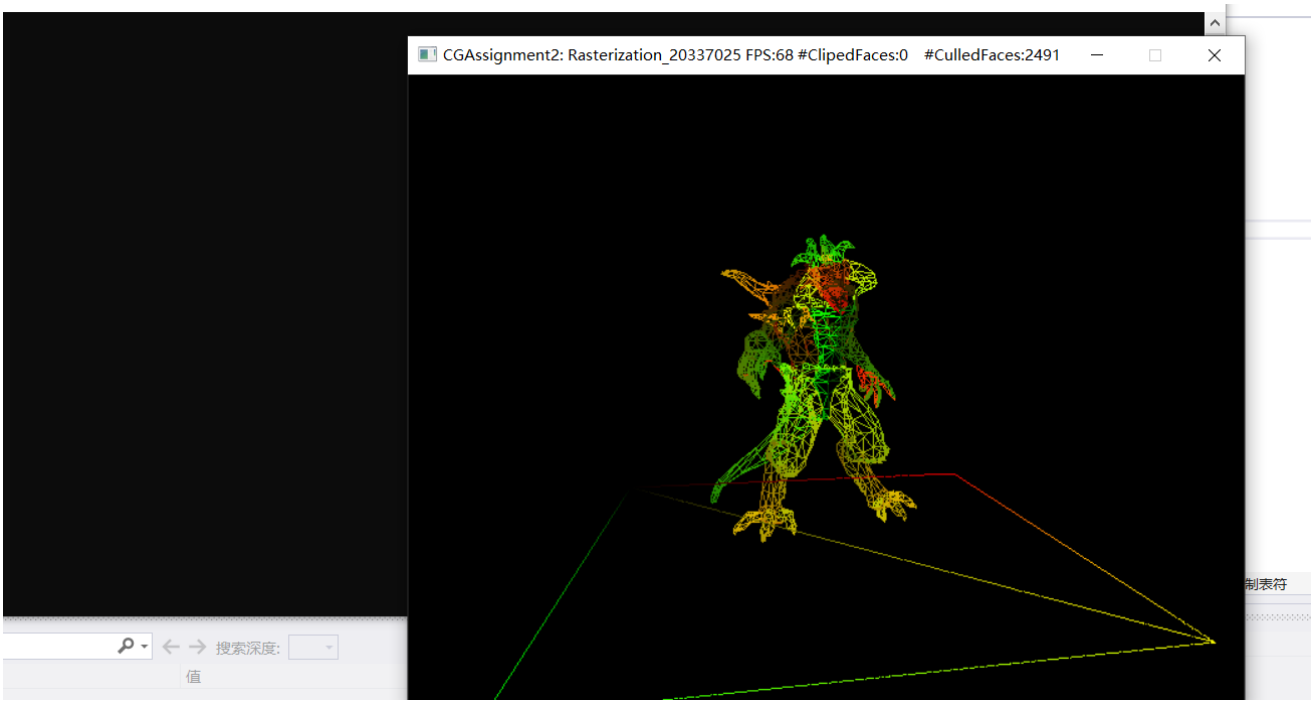
实现三角形的背向面剔除，简述你是怎么做的。你要填充的函数在 TRRenderer.cpp 文件中 `isTowardBackFace()`，其中传入的参数为ndc空间下的三角形顶点坐标。

答：
直接通过叉乘得到三角形的法线朝向，然后与视线方向（0，0，1）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，具体代码如下：

```
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 m = v1 - v0;
        glm::vec3 n = v2 - v1;
        glm::vec3 view = glm :: vec3(0, 0, 1);
        glm::vec3 fa_v = glm::cross(m, n);//法向量
        float charge = glm::dot(fa_v, view);//点乘然后进行判断
```

```
        if (charge < 0)
                return true;
        else
                return false;
}
```

实现效果如下：



Task 4

实现基于Edge-function的三角形填充算法。你要填充的函数在 TRShaderPipeline.cpp 文件中。

答：基于Edge-function的三角形填充算法首先计算三角形的包围盒，包围盒是一个四条边平行于行列的长方形，恰巧包围住这个三角形。它的计算也相当简单：所有顶点坐标对应维度的最小值构成的点为 p，最大值构成的的点为 q，包围盒就由这两个点表示。

然后遍历包围盒内的每一个像素点，计算其是否在三角形内部，计算方法可以用重心坐标来计算，设三角形内部一点P，有：

$$\vec{AP} = u\vec{AB} + v\vec{AC}$$

即

$$u\vec{AB} + v\vec{AC} + \vec{PA} = 0$$

将其拆开，可以得到：

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

其中$0 <= u, v <= 1$。
考虑二维的三角形，可以写成矩阵形式：

$$[u \quad v \quad 1] \begin{bmatrix} A\vec{B}_x \\ A\vec{C}_x \\ P\vec{A}_x \end{bmatrix} = 0$$

$$[u \quad v \quad 1] \begin{bmatrix} A\vec{B}_y \\ A\vec{C}_y \\ P\vec{A}_y \end{bmatrix} = 0$$

由此可见，向量$\vec{n} = (u, v, 1)$同时垂直于向量$(A\vec{B}_x, A\vec{C}_x, P\vec{A}_x)$和向量$(A\vec{B}_y, A\vec{C}_y, P\vec{A}_y)$，即可以通过向量叉乘求出$\vec{n}$。由于涉及到浮点数，可能$\vec{n}$的z分量不会一定等于1.0,
令$\vec{n}$的三个分量是 (a, b, c),代入原式：

$$a\vec{AB} + b\vec{AC} + c\vec{PA} = 0$$

$$P = (1 - a/c - b/c)A + a/cB + b/cC, c \neq 0$$

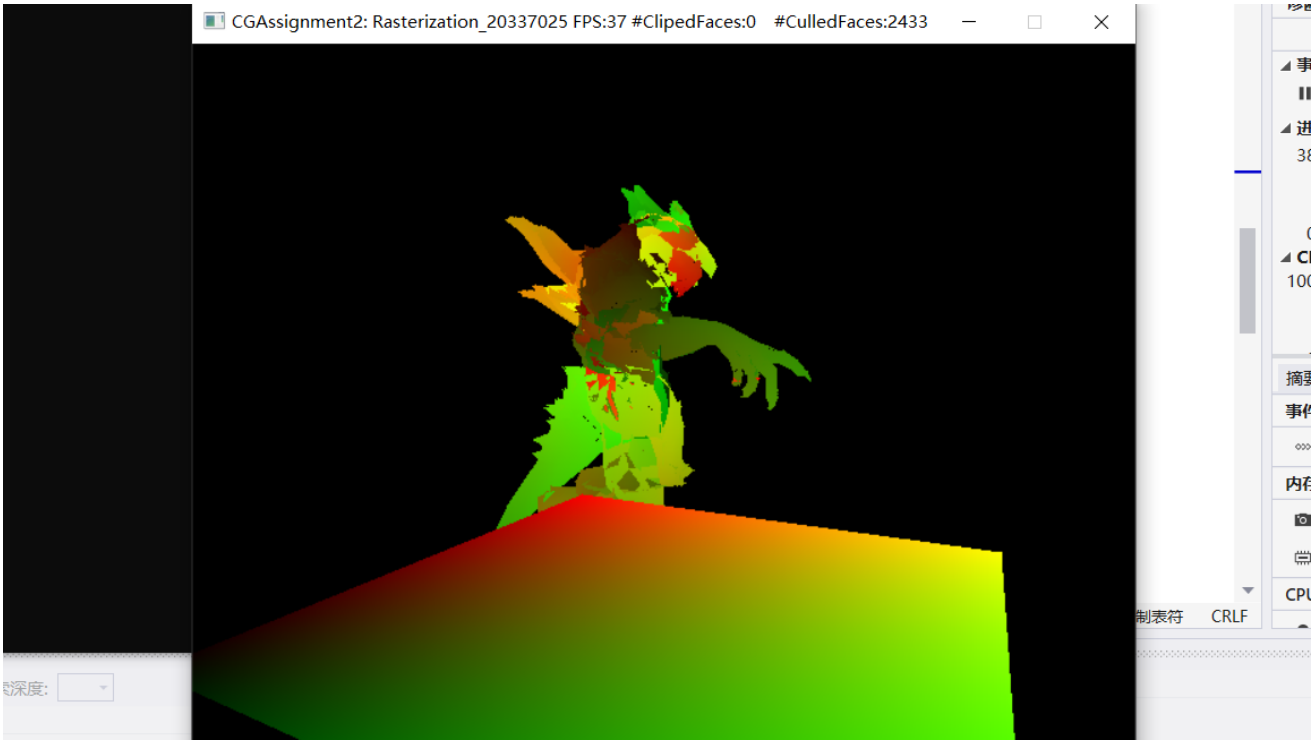
若$0 <= a/c <= 1, 0 <= b/c <= 1, (1 - a/c - b/c) >= 0$，则P在三角形内（上）,否则P在三角形外部。

根据以上分析编写相应代码如下：

```
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)
{
        //For instance:
        int max_X = std::max({ v0.spos.x, v1.spos.x, v2.spos.x });
        int max_Y = std::max({ v0.spos.y, v1.spos.y, v2.spos.y });
        int min_X = std::min({ v0.spos.x, v1.spos.x, v2.spos.x });
        int min_Y = std::min({ v0.spos.y, v1.spos.y, v2.spos.y });
        for (int i = min_X; i <= max_X; i++) {
                for (int j = min_Y; j <= max_Y; j++) {
                        glm::vec3 x = glm::vec3(v1.spos.x - v0.spos.x, v2.spos.x
- v0.spos.x, v0.spos.x - i);
                        glm::vec3 y = glm::vec3(v1.spos.y - v0.spos.y, v2.spos.y
- v0.spos.y, v0.spos.y - j);
                        glm::vec3 n = glm::cross(x, y);//叉乘得到n
                        glm::vec3 w = { 1.0f - n.x/n.z - n.y/n.z, n.x / n.z, n.y
/ n.z };
                                //判断
                        if (w.x >= 0 && w.y >= 0 && w.x + w.y <= 1) {
```

```
                                VertexData interploated =
VertexData::barycentricLerp(v0, v1, v2, w);
                                interploated.spos.x = i, interploated.spos.y =
j;
                                rasterized_points.push_back(interploated);
                        }
                }
        }
}
```

实现效果如下，由于尚未实现深度测试，所以没有呈现正确的前后遮挡关系：



Task 5

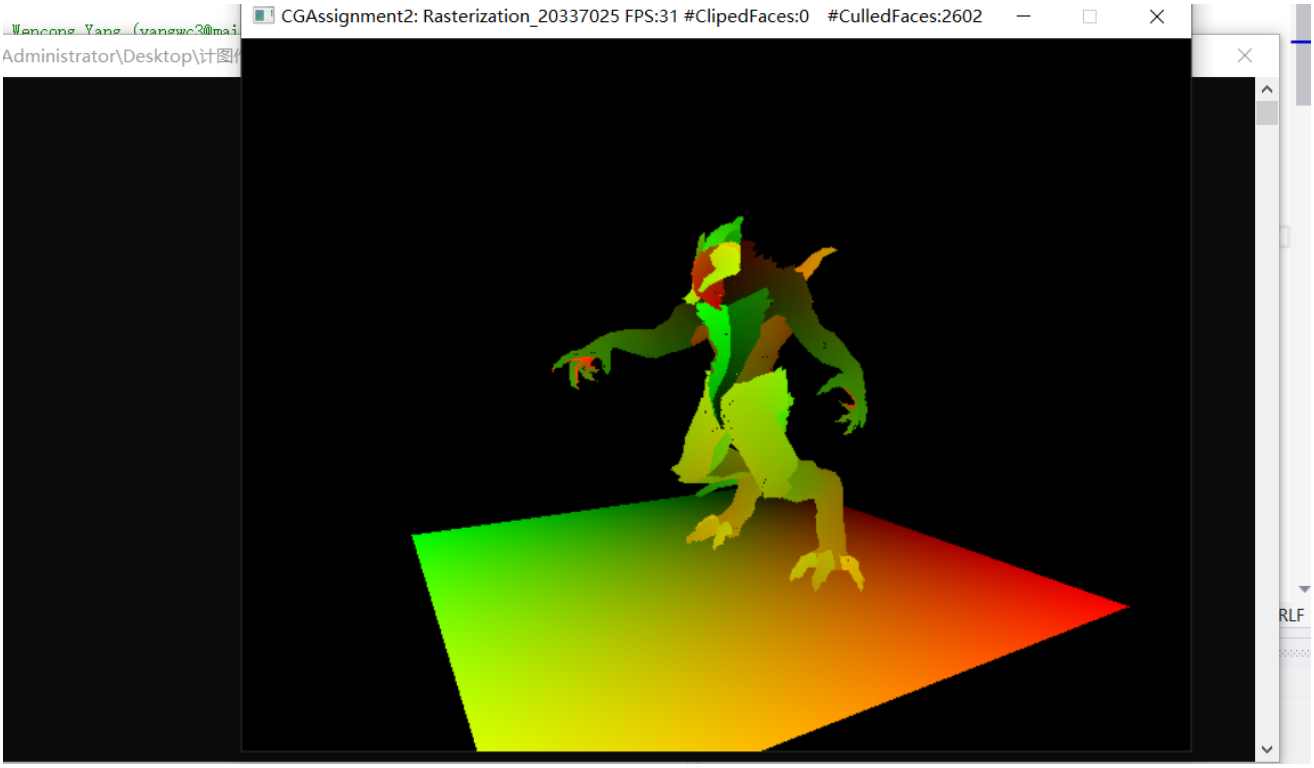
实现深度测试，这里只需编写一行代码即可，你需要填充代码的地方在文件 TRRenderer.cpp 的 renderAllDrawableMeshes 函数，只需写一个 if语句即可。

答：`points.cpos.z`是当前片元的深度，通过函数`m_backBuffer->readDepth()`得到当前深度缓冲的深度值，将两者进行比较即可：

```
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
        {
        //Perspective correction after rasterization
        if (m_backBuffer->readDepth(points.spos.x, points.spos.y) <
```

```
points.cpos.z)
                continue;
        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);
        }
}
```

实现效果如下，可见经过深度测试后，实现了正确的遮挡关系，达到了预期的效果：



Task 6

遇到的问题&困难：

1. 在一开始实现Bresenham直线光栅化算法时，没有考虑直线的斜率k<1的情况，虽然程序也可以正常运行，但肯定效果不是很好，在经过同学的提醒后在原来的？Task 1代码上加入斜率k<1的情况即可。即将k>1的那部分代码的x和y互换即可。
2. 在Task 4实现基于Edge-function的三角形填充算法时，一开始没有参考实验指导中的用重心法来判断一个点是否在三角形的内部，而是采取了更加复杂的方法（面积法：若ABC的面积=APB的面积+BPC的面积+APC的面积，则说明P点在三角形的内部；若ABC的面积<APB的面积+BPC的面积+APC的面积，则说明o点在三角形的外部），导致代码量比较大，且效率很低，根据实验指导上的提示采用重心法后，效率高了很多。

体会：

在完成了直线光栅化、三角形填充光栅化、齐次空间简单裁剪、背面剔除之后，我最大的感受就是计算机图形学在节省不必要的开销和提高效率方面做出的努力，如通过齐次空间简单

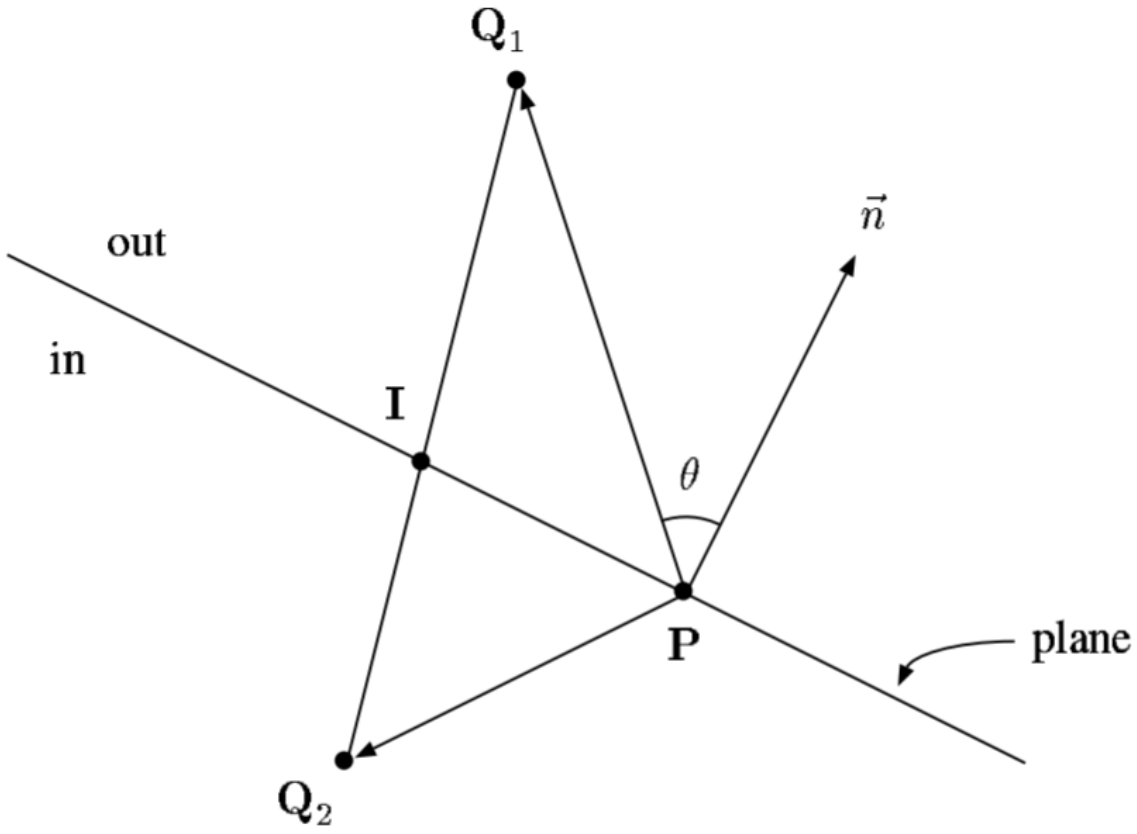
裁剪除去渲染视锥体之外的物体，这省去不必要的计算开销；通过三角形的背向面剔除来节省片段着色器的执行数；在Edge-function中通过包围盒来缩小遍历的范围.....而正是通过这些方法，我们才能高效地将图像表现出来，并花费小的开销来达到更好的效果，我想这也是我们得以用较低配置的设备游玩到一些画质较为精美的游戏的重要原因。

除此之外，在一步步跟着指导完成作业的过程中，我学习到了许多新的知识和技能，并能将课上学习到的理论知识转化为实践的内容，这让我受益匪浅。

Task 7

在简单的齐次空间裁剪，三角形只要有一个顶点在视锥体范围之外则将三角形剔除，这种方法虽然简单，但缺点是当有一些三角形大部分处于视锥体内，仅有一小撮位于视锥体外时，将其进行简单裁剪会使得图像缺失较多，为了达到更好的效果，正确的做法应该是求出三角形与裁剪平面的交线的两个端点P1，P2。并将P1,P2和三角形位于视锥体内的顶点重新组合成新的三角面并保留，仅裁剪掉交线以外的部分。

首先需要判断线与面的关系，如图所示：



d1，d2的几何含义为向量在法线上带符号的投影距离：

$$d_1 = (Q_1 - P) \cdot \vec{n}$$

$$d_1 = (Q_1 - P) \cdot \vec{n}$$

可以得到：

1. 如果d1>0,d2>0，则边Q1Q2完全处于裁剪平面外侧

2. 如果d1<0,d2<0，则边Q1Q2完全处于裁剪平面内侧
3. 如果d1*d2<0，则边Q1Q2与裁剪平面相交与点
   当边与裁剪平面相交时，求出交点I，计算过程如下：

$$I = Q_1 + t(Q_2 - Q_1)$$

   其中$t = \frac{|Q_1 I|}{|Q_1 Q_2|} = \frac{d_1}{d_1 - d_2}$，将交点保存起来即可。
   根据以上过程写出裁剪算法的代码如下：

```
//clip_plane结构体定义在TRRenderer.h头文件中
struct clip_plane {
        glm::vec4 P;//平面
        glm::vec4 n;//法线
};
//clip_with_plane函数定义在TRRenderer.h头文件中，实现在TRRenderer.cpp文件
std::vector<TRShaderPipeline::VertexData> TRRenderer::clip_with_plane(clip_plane
& c_plane, std::vector<TRShaderPipeline::VertexData>& vert_list) const {
        int i;
        int num_vert = vert_list.size();
        int previous_index,current_index;
        std::vector<TRShaderPipeline::VertexData> in_list;
        for (i = 0; i < num_vert; i++) {
                current_index = i;
                previous_index = (i - 1 + num_vert) % num_vert;
                TRShaderPipeline::VertexData &pre_vertex =
vert_list[previous_index]; //边的起始点
                TRShaderPipeline::VertexData &cur_vertex =
vert_list[current_index];  //边的终止点
                float d1 = glm::dot(pre_vertex.cpos / pre_vertex.cpos.w -
c_plane.P,c_plane.n);
                float d2 = glm::dot(cur_vertex.cpos / cur_vertex.cpos.w -
c_plane.P, c_plane.n);
                //存交点
                if (d1 * d2 < 0) {
                        float t = d1 / (d1 - d2);
                        TRShaderPipeline::VertexData I =
TRShaderPipeline::VertexData::lerp(pre_vertex, cur_vertex, t);
                        in_list.push_back(I);
                }
                //直接存入
                if (d2 < 0) {
                        in_list.push_back(cur_vertex);
                }
                }
        return in_list;
}
```

有了裁剪函数后，对于这样一个由6个平面组成的视锥体进行裁剪，只需依次以每一个面进行裁剪，并且前一次裁剪的输出为下一次裁剪的输入即可，每个平面的信息如下：

- top plane: (0,1,0,1)，法线为n=(0,cosα/2,-sinα/2)
- bottom plane: (0,-1,0,1)，法线为n=(0,−cosα/2,-sinα/2)
- left plane: (1,0,0,1)，法线为n=(cosα/2,0,-sinα/2)
- right plane: (-1,0,0,1)，法线为n=(-cosα/2,0,-sinα/2)
- near plane: (0,0,1,1)，法线为n=(0,0,1)
- far plane: (0,0,-1,1)，法线为n=(0,0,−1)

其中，n,f分别为近投影平面，和远投影平面的距离,α为fov的大小。
代码实现如下：

```
std::vector<TRShaderPipeline::VertexData> TRRenderer::cliping(const
TRShaderPipeline::VertexData &v0,const TRShaderPipeline::VertexData &v1,const
TRShaderPipeline::VertexData &v2) const
{
	std::vector<TRShaderPipeline::VertexData> in_list =
std::vector<TRShaderPipeline::VertexData>({ v0, v1, v2 });

	//top
	in_list = clip_with_plane(clip_plane({ glm::vec4(0, 1, 0, 1),
glm::vec4(0,std::cosf(M_PI / 4.0f),-std::sinf(M_PI / 4.0f), 0) }), in_list);
	//bottom
	in_list = clip_with_plane(clip_plane({ glm::vec4(0, -1, 0, 1),
glm::vec4(0, -std::cosf(M_PI / 4.0f), -std::sinf(M_PI / 4.0f), 0) }), in_list);
	//left
	in_list = clip_with_plane(clip_plane({ glm::vec4(1, 0, 0, 1),
glm::vec4(std::cosf(M_PI / 4.0f), 0,-std::sinf(M_PI / 4.0f), 0) }), in_list);
	//right
	in_list = clip_with_plane(clip_plane({ glm::vec4(-1, 0, 0, 1),
glm::vec4(-std::cosf(M_PI / 4.0f), 0, -std::sinf(M_PI / 4.0f), 0) }), in_list);
	//near
	in_list = clip_with_plane(clip_plane({ glm::vec4(0, 0, 1, 1),
glm::vec4(0, 0, 1, 0) }), in_list);
	//fars
	in_list = clip_with_plane(clip_plane({ glm::vec4(0, 0, -1, 1),
glm::vec4(0, 0, -1, 0) }), in_list);
	return in_list;
}
```

实现效果如下，可见图像放大后边缘丝滑了不少：