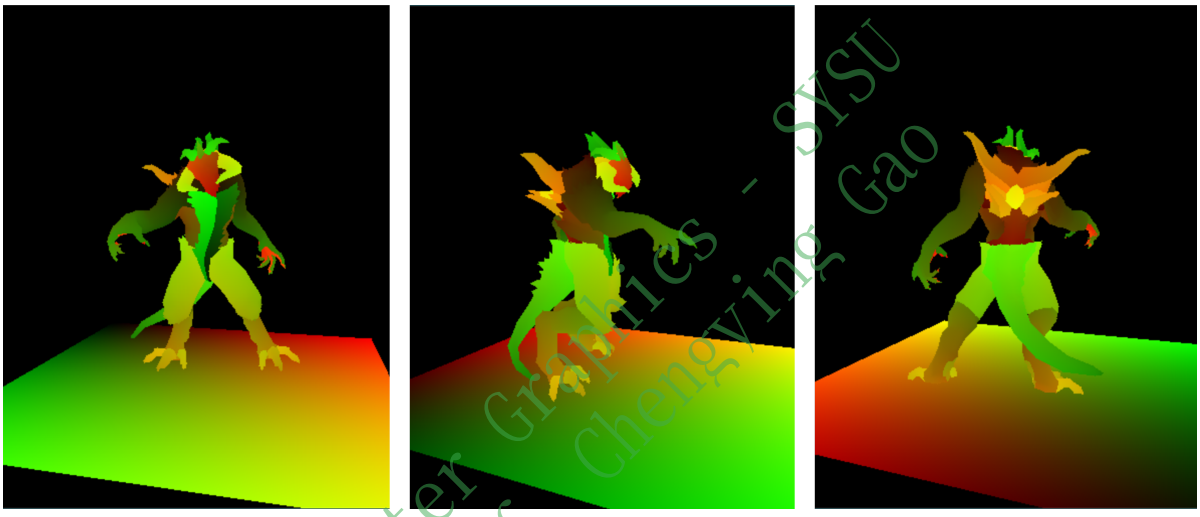
Assignment 2: Rasterization & Z-buffering

Computer Graphics Teaching Stuff, Sun Yat-Sen University

Due Date: 具体截止日期见群公告

*Submission: Send the report (**In .PDF Format**) to mailbox* (邮箱地址见群公告)

> 在上次作业中，我们了解并实现了视图变换、投影变换和视口变换，并尝试用旋转变换和缩放变换对三维物体进行操纵，最终屏幕上显示的是一个线框绘制的网格模型，可以明显看到这个网格模型的基本几何单元是三角形。我们知道，经过一系列矩阵变换之后，三维空间的三角形最终会投影到屏幕上，此时我们拿到了三个顶点在屏幕上的坐标。接下来的任务就是要确定这个屏幕空间的三角形覆盖了哪些像素，这就是光栅化！本次作业的任务就是要你们实现光栅化的算法。



本次作业你们将实现的效果图

1、作业概述

光栅化是将向量图形格式表示的图像转换成位图以用于显示器或者打印机输出的过程。目前我们的电子计算机采用栅格点阵的方式来显示图像、图形等数据，对于输入的连续信号（例如三角形），计算机会对该信号进行离散地采样。对应到三角形的光栅化过程，就是确定哪些栅格点（亦或者说像素点）被三角形覆盖了，我们会对被三角形覆盖了的像素点进行着色，从而最终在屏幕上显示出输入的三角形的形状和颜色。

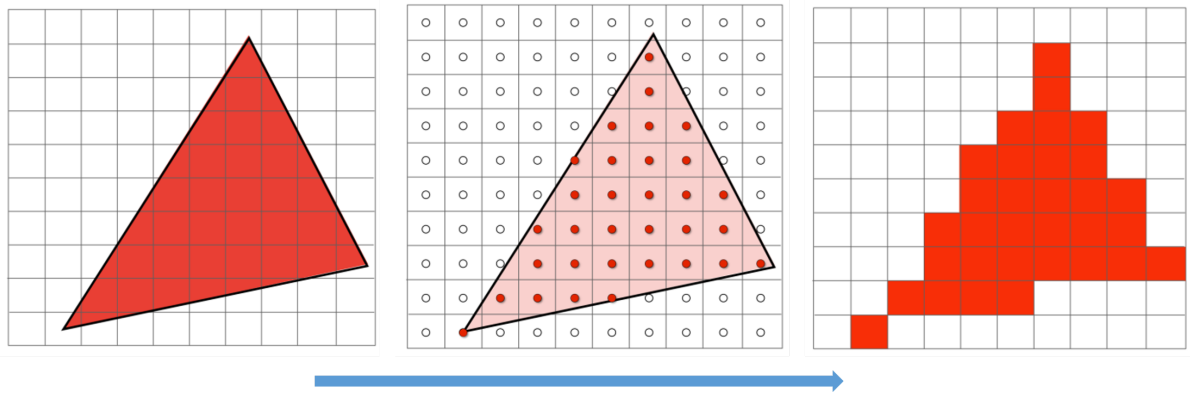


图1 三角形光栅化

本次作业你们将在给定的代码框架下，实现Bresenham直线光栅化、三角形填充光栅化和深度缓冲算法，进一步，实现并熟悉课程中所讲的Z-buffer算法以及反走样算法。完成本次作业，你将对整个渲染管线有更进一步的深入理解。

2、代码框架

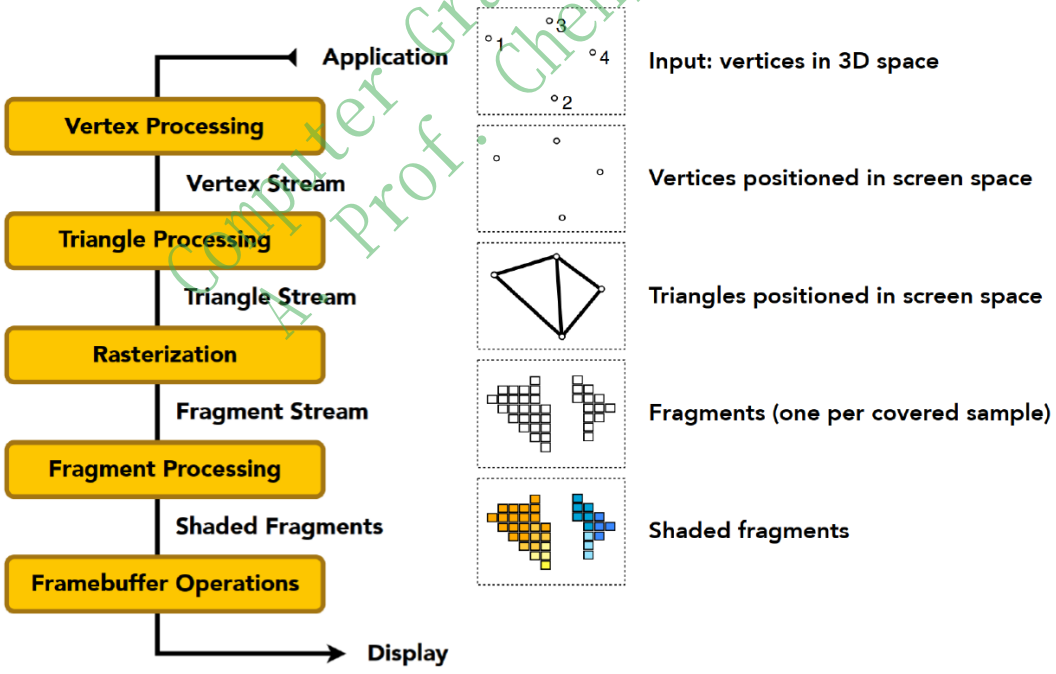
关于本次作业的框架代码部署和构建，分为基础任务的代码框架以及提升任务的代码框架，请分别仔细阅读对应的文档。

一、CGAssignment2

请仔细阅读`CGAssignment2/readme.pdf`文档，此文档与`CGAssignment1/readme.pdf`文档类似。

目录`CGAssignment2/src`存放我们的渲染器的所有代码文件：

- **main.cpp**：程序入口代码，负责执行主要的渲染循环逻辑；
- **TRFrameBuffer(.h/.cpp)**：帧缓冲类，存放渲染的结果（包括颜色缓冲和深度缓冲），你无需修改此文件；
- **TRShaderPipeline(.h/.cpp)**：渲染管线类，负责实现顶点着色器、光栅化、片元着色器的功能；
- **TRWindowsApp(.h/.cpp)**：窗口类，负责创建窗口、显示结果、计时、处理鼠标交互事件，无需修改；
- **TRDrawableMesh(.h/.cpp)**：可渲染对象类，负责加载obj网格模型、存储几何顶点数据，无需修改；
- **TRRenderer(.h/.cpp)**：渲染器类，负责存储渲染状态、渲染数据、调用绘制。
-



本代码涉及的是三角形处理、光栅化和深度测试（主要为图2的**Triangle Processing**和**Rasterization**部分）。经过矩阵变换之后，顶点着色器的输出的几何顶点数据存在`VertexData`对象中，如下所示：

```
class VertexData
{
public:
    glm::vec4 pos;  //World space position
    glm::vec3 col;  //World space color
```

```
    glm::vec3 nor;  //World space normal
    glm::vec2 tex;  //World space texture coordinate
    glm::vec4 cpos; //Clip space position
    glm::ivec2 spos;//Screen space position

    //Linear interpolation
    static VertexData lerp(const VertexData &v0, const VertexData &v1, float
frac);
    static VertexData barycentricLerp(const VertexData &v0, const VertexData &v1,
const VertexData &v2, glm::vec3 w);
};
```

该类的定义在`TRShaderPipeline.h`文件中。其中的`cpos`存储了经过投影变换之后的顶点坐标，该顶点坐标在齐次裁剪空间，然后经过透视除法变换到ndc空间，最后经过视口变换变换到屏幕空间，如下所示：

```
vert[0].spos = glm::ivec2(m_viewportMatrix * vert[0].cpos + glm::vec4(0.5f));
vert[1].spos = glm::ivec2(m_viewportMatrix * vert[1].cpos + glm::vec4(0.5f));
vert[2].spos = glm::ivec2(m_viewportMatrix * vert[2].cpos + glm::vec4(0.5f));
```

因此，`VertexData`的`spos`存储了该顶点在屏幕空间的`xy`坐标。这些变换都已经在渲染器中实现。同学们在实现光栅化时需要用的是对`VertexData`的线性插值函数`lerp`，其中`frac`是插值权重：

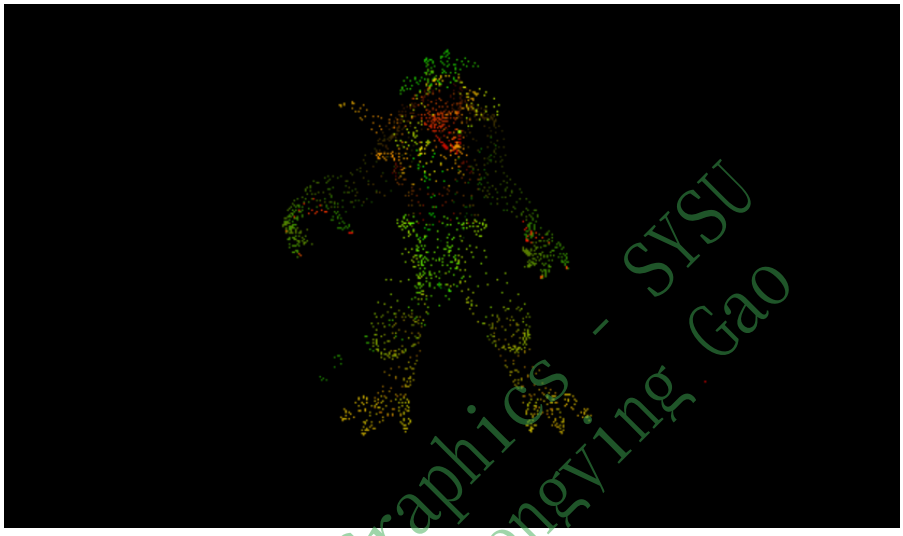
```
TRShaderPipeline::VertexData TRShaderPipeline::VertexData::lerp(
    const TRShaderPipeline::VertexData &v0,
    const TRShaderPipeline::VertexData &v1,
    float frac)
{
    //Linear interpolation
    VertexData result;
    result.pos = (1.0f - frac) * v0.pos + frac * v1.pos;
    result.col = (1.0f - 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;
}
```

以及基于重心坐标的插值函数`barycentricLerp`（本质上也是线性插值），其中`w`s是插值权重：

```
TRShaderPipeline::VertexData TRShaderPipeline::VertexData::barycentricLerp(
    const VertexData &v0,
    const VertexData &v1,
    const VertexData &v2,
    glm::vec3 w)
{
    VertexData result;
    result.pos = w.x * v0.pos + w.y * v1.pos + w.z * v2.pos;
    result.col = w.x * v0.col + w.y * v1.col + w.z * v2.col;
    result.nor = w.x * v0.nor + w.y * v1.nor + w.z * v2.nor;
```

```
    result.tex = w.x * v0.tex + w.y * v1.tex + w.z * v2.tex;
    result.cpos = w.x * v0.cpos + w.y * v1.cpos + w.z * v2.cpos;
    result.spos.x = w.x * v0.spos.x + w.y * v1.spos.x + w.z * v2.spos.x;
    result.spos.y = w.x * v0.spos.y + w.y * v1.spos.y + w.z * v2.spos.y;

    return result;
}
```

之所以要用到线性插值，是因为光栅化的输入仅仅是三角形的三个顶点的几何属性，对于光栅化后三角形内部的点，我们需要通过插值的方式来确定其几何属性，用以后续的着色处理。关于更多线性插值的概念，请见维基百科。对于直线光栅化，插值权重的计算本质上就是直线方程；对于三角形填充光栅化，插值权重的计算需要计算重心坐标。成功构建好代码框架之后，编译运行，你将得到如下的结果，此时没有做光栅化，只是输出显示网格的几何顶点。



二、CGAssignment2-Improve

该代码用于提升作业和挑战作业的编写。

目录`CGAssignment2-Improve`存放我们的渲染器的所有代码文件：

- **main.cpp**：程序入口代码，负责执行主要的渲染循环逻辑；
- **Triangle(.h/.cpp)**：可渲染对象类，定义三角形相关属性，无需修改；
- **rasterizer(.h/.cpp)**：渲染器类，负责存储渲染状态、渲染数据、调用绘制。

本代码涉及的主要是对Z-buffer算法的实现（也为图2的**Triangle Processing**和**Rasterization**部分）。

*该代码作业中所使用到的库为**Eigen 与Opencv**，请确保这两个库的配置正确！*

对于代码的编译，可以参考CGAssignment2的代码框架进行编译：

如果你使用Windows系统，可参考该链接进行库配置。

【请注意：不同于基础任务，本任务在使用cmake-gui创建项目时请选用x64，因为opencv依赖库的lib是x64的】

如果使用Linux系统，可参考该链接进行Opencv库的安装，此代码采用./CGAssignment2-Improve进行运行。

3、作业描述

本次作业中，本次作业中，我们将提供**基础（分数占比 60%）**、**提升（分数占比 25%左右）**和**挑战（分数占比 15%左右）**作业，我们鼓励大家尽可能地完成所有作业，但还请根据自身能力选择并完成任务，**因为我们杜绝任何形式地抄袭！**

请你按照下面的顺序完成以下的任务：

基础任务：

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`）参考链接。

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。

```
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)
{

    // 1: 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.

}
```


该函数在`rasterize_wire`函数中被调用，对三角形的每一条边都进行直线光栅化。

请贴出实现结果，并简述你是怎么做的。

2、实现基于Edge-function的三角形填充算法。参考链接

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。你要填充的函数在`TRShaderPipeline.cpp`文件中，如下所示：

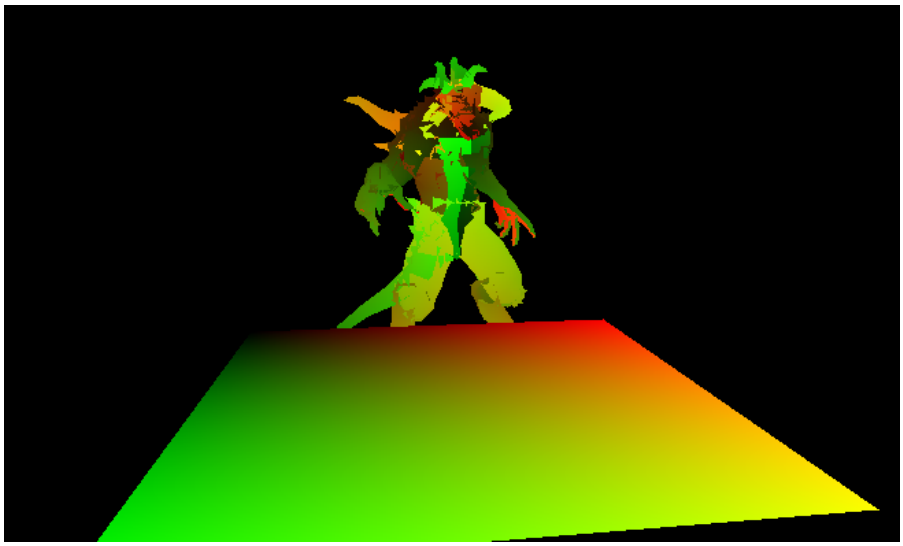
```
void TRShaderPipeline::rasterize_fill_edge_function(
    const VertexData &v0,
    const VertexData &v1,
    const VertexData &v2,
    const unsigned int &screen_width,
    const unsigned int &screene_height,
```

```
    std::vector<VertexData> &rasterized_points)
{
    //Edge-function rasterization algorithm

    // 2: 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.

}
```

关于重心坐标的计算，可参考这里，请特别注意边界条件。**实现好之后，请注意在`TRRenderer.cpp`文件的`renderAllDrawableMeshes`函数，把`m_shader_handler->rasterize_wire`这个函数调用注释掉，改为调用`m_shader_handler->rasterize_fill_edge_function`函数。**

```
//Rasterization stage
{
    //Transform to screen space & Rasterization
    {
        vert[0].spos = glm::ivec2(m_viewportMatrix * vert[0].cpos +
glm::vec4(0.5f));
        vert[1].spos = glm::ivec2(m_viewportMatrix * vert[1].cpos +
glm::vec4(0.5f));
        vert[2].spos = glm::ivec2(m_viewportMatrix * vert[2].cpos +
glm::vec4(0.5f));

        //m_shader_handler->rasterize_wire(vert[0], vert[1], vert[2],
        //  m_backBuffer->getWidth(), m_backBuffer->getHeight(),
rasterized_points);
        m_shader_handler->rasterize_fill_edge_function(vert[0], vert[1], vert[2],
            m_backBuffer->getWidth(), m_backBuffer->getHeight(),
rasterized_points);
    }
}
```


实现结果可能看起来会很奇怪，如下图所示（例如模型的脚反而被地板遮挡住了），请不要担心！此时尚未实现深度测试，所以没有呈现正确的前后遮挡关系！



3、实现深度测试，这里只需编写一行代码即可。

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。经过光栅化的线性插值，每个片元都有一个深度值，存储在 `cpos.z` 中。在着色阶段，我们可以用当前片元的 `cpos.z` 与当前深度缓冲的深度值进行比较，如果发现深度缓冲的取值更小（即更近），则应该直接不进行着色器并写入到帧缓冲。

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：

```
// 3: 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
    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);
}
```

正确实现了深度测试，那么你将会编译运行得到本文档开头的图片效果。恭喜你！

提升任务：

为了帮助你更好地掌握光栅化与**Z-buffer算法**，我们希望你能正确光栅化三角形，并通过深度缓冲区，**正确绘制两个三角形的前后关系。**

为此你需要正确填写并调用在文件 `rasterizer.cpp` 的 `rasterize_triangle` 函数，该函数的内部工作流程如下：

1、创建三角形的2维包围盒。

2、遍历此包围盒内的所有像素（使用其整数索引）。然后，使用像素中心的屏幕空间坐标来检查中心点是否在三角形内。

3、如果在内部，则将其位置处的插值深度值与深度缓冲区中的相应值进行比较。

4、如果当前点更靠近相机，请设置像素颜色并更新深度缓冲区。

因为我们只知道三角形三个顶点处的深度值，所以对于三角形内部的像素，我们需要使用插值的方法得到其深度值。插值的深度值被存储在`z_interpolated`中。请注意我们是如何初始化 depth buffer 和注意 z values 的符号。为了方便同学们写代码，我们将 z 进行了反转，保证都是正数，并且越大表示离视点越远。

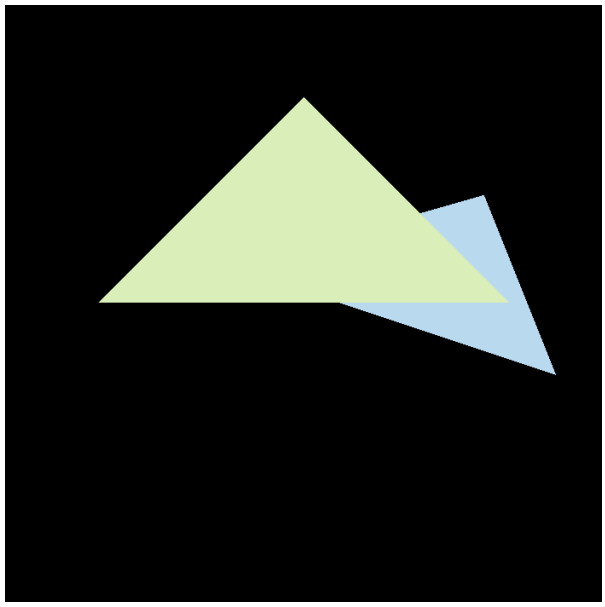
你需要填充的函数如下所示：

```
void rst::rasterizer::rasterize_triangle(const Triangle& t) {
    auto v = t.toVector4();

    // Task_improve: Find out the bounding box of current triangle.
    // iterate through the pixel and find if the current pixel is inside the
triangle

    // If so, use the following code to get the interpolated z value.

    //auto[alpha, beta, gamma] = computeBarycentric2D(x, y, t.v);
    //float w_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma /
v[2].w());
    //float z_interpolated = alpha * v[0].z() / v[0].w() + beta * v[1].z() /
v[1].w() + gamma * v[2].z() / v[2].w();
    //z_interpolated *= w_reciprocal;

    // TODO : set the current pixel (use the set_pixel function) to the color of
the triangle (use getColor function) if it should be painted.
}
```

为了你能更顺利地完成此任务，我们已经帮助你实现测试点是否在三角形内部的函数。你也可以修改此函数，按照自己的方式去判定。

```
static bool insideTriangle(int x, int y, const Vector3f* _v)
{
    Vector3f Q = {float(x), float(y), 0};
    Vector3f p0p1 = _v[1] - _v[0];
    Vector3f p0Q = Q - _v[0];
    Vector3f p1p2 = _v[2] - _v[1];
    Vector3f p1Q = Q - _v[1];
    Vector3f p2p0 = _v[0] - _v[2];
    Vector3f p2Q = Q - _v[2];

    //The class definition is already counterclockwise, so we only have to
consider the same positive case
    return p0p1.cross(p0Q).z() > 0 && p1p2.cross(p1Q).z() > 0 &&
p2p0.cross(p2Q).z()>0;
}
```

如果程序实现正确，你将能看到如下所示的输出图像，同时，该图像会保存到build文件夹下。



挑战任务：

你可能会注意到，当我们放大图像时，图像边缘会有锯齿感!

用 super-sampling 处理 Anti-aliasing : 我们可以用 super-sampling来解决这个问题，即对每个像素进行 2 * 2 采样，并比较前后的结果 (这里并不需要考虑像素与像素间的样本复用)。需要注意的点有，对于像素内的每一个样本都需要维护它自己的深度值，即每一个像素都需要维护一个 samplelist。

为此，你可能需要在对应位置添加超采样相关的数组及函数，主要修改仍然是在文件 `rasterizer.cpp` 的 `rasterize_triangle` 函数中。

最后，如果你实现正确的话，你得到的三角形不应该有不正常的黑边。

注意事项:

- 将作业文档、源代码一起压缩打包，文件命名格式为：学号+姓名+HW2，例如19214044+张三+HW2.zip。
- **提交的文档请提交编译生成的pdf文档，请勿提交markdown、docx以及图片资源等源文件!**
- **提交代码只需提交源代码文件即可，请勿提交教程文件、作业描述文件、工程文件、中间文件和二进制文件（即删掉build目录下的所有文件！）。**
- 禁止作业文档抄袭，我们鼓励同学之间相互讨论，但最后每个人应该独立完成。
- 可提交录屏视频作为效果展示（只接收mp4或者gif格式），请注意视频文件不要太大。