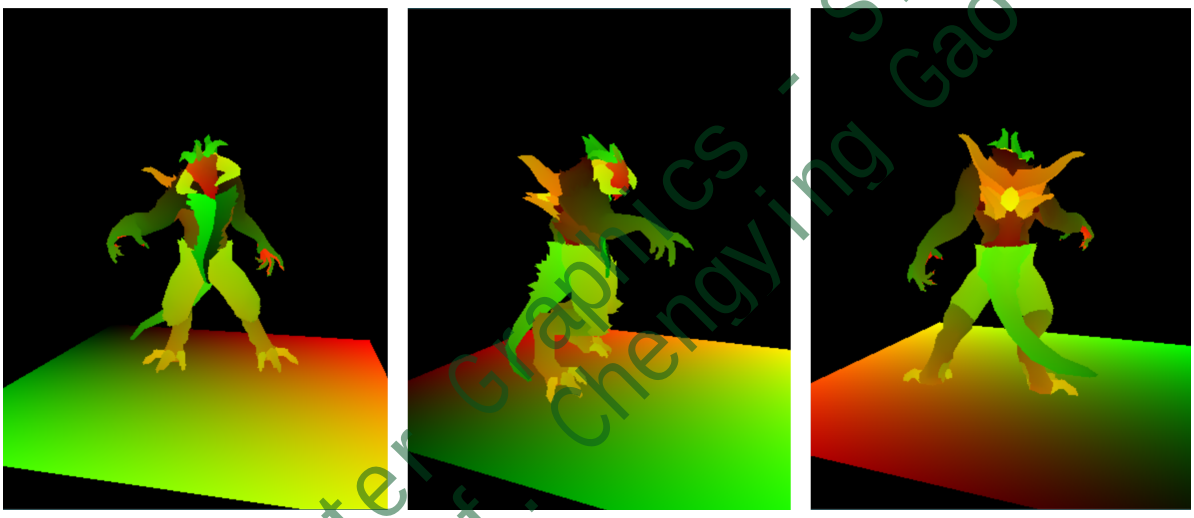
Assignment 2: Rasterization & Z-buffering

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。

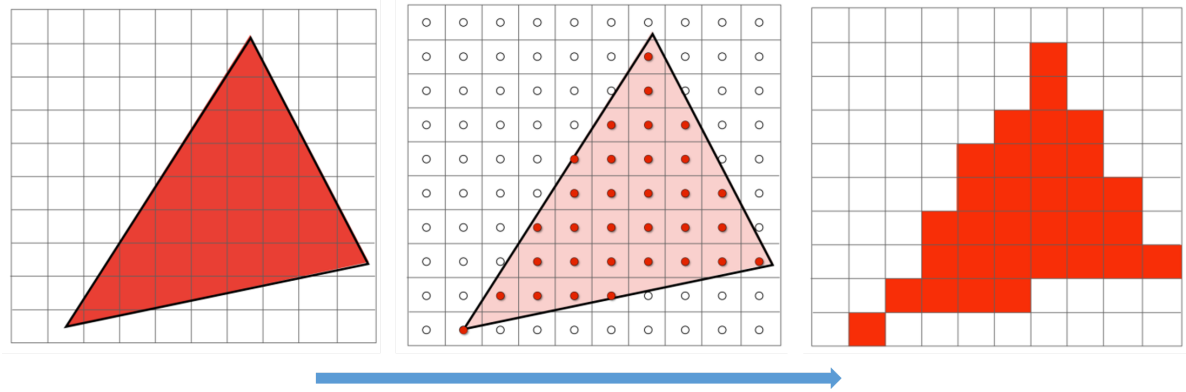


图1 三角形光栅化

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、简单的裁剪方法、背向面剔除和深度缓冲算法。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，请仔细阅读`CGAssignment2/readme.pdf`文档，基本上与`CGAssignment1`没有太大的差别。本次作业依赖的第三方库主要有三个，分别是：

- **GLM**：线性代数数学库，如果学过LearnOpenGL教程则你应该对这个数学库挺熟悉的
- **SDL2**：窗口界面库，主要用于创建窗口给并显示渲染的图片结果，本作业不需要你对这个库深入了解
- **TinyObjLoader**：模型数据加载库，用于加载obj模型，本作业不需要你对这个库深入了解

这些第三方库同学们无需太过关注，我们的框架代码已经构建好了相应的功能模块。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。

核心的渲染模块没有借助任何第三方库，完全是利用纯粹的代码构建了类似于OpenGL的渲染管线，同学们可以类比于OpenGL来阅读本代码框架（无任何硬件层面的加速和并行，因此通常叫软光栅化渲染器）。本次作业你需要修改和填补的代码的地方在`TRRenderer.cpp`文件和`TRShaderPipeline.cpp`文件，请大家着重注意这两个文件的代码。

`main.cpp`已经实现了窗口的创建、渲染器的实例化、渲染数据的加载、渲染循环的构建，请你类比在上一次作业中学习到的渲染流程，仔细体会其中的逻辑。对代码有任何的疑问，可在群里匿名讨论。渲染的核心逻辑在`TRRenderer.cpp`文件的`renderAllDrawableMeshes`函数（不需要你对这个函数做任何的修改），同学们可以结合第三节课讲的渲染管线流程进行理解：

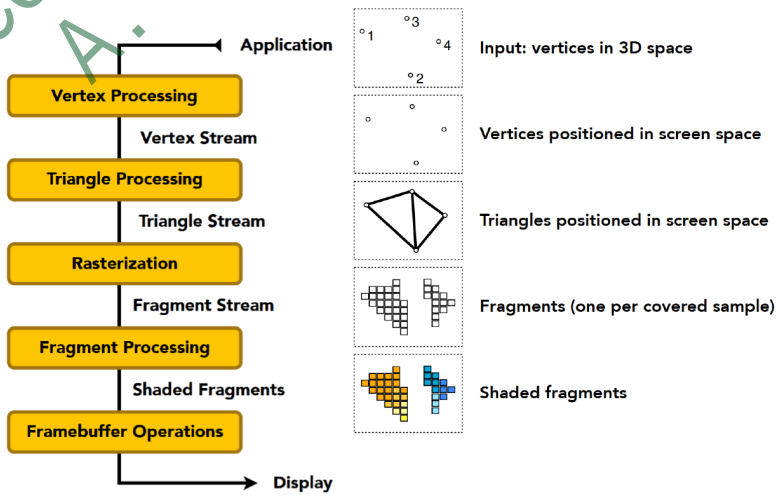


图2 基于光栅化的渲染管线

考虑到同学们的基础，同学们可能对渲染器的一些代码不是很理解，这个没关系，不理解的地方大家可以暂时放一下，随着学习的推进，后面可以回来再看。完成本次作业不需要你对整个框架代码做彻底的理解！本次作业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));
```

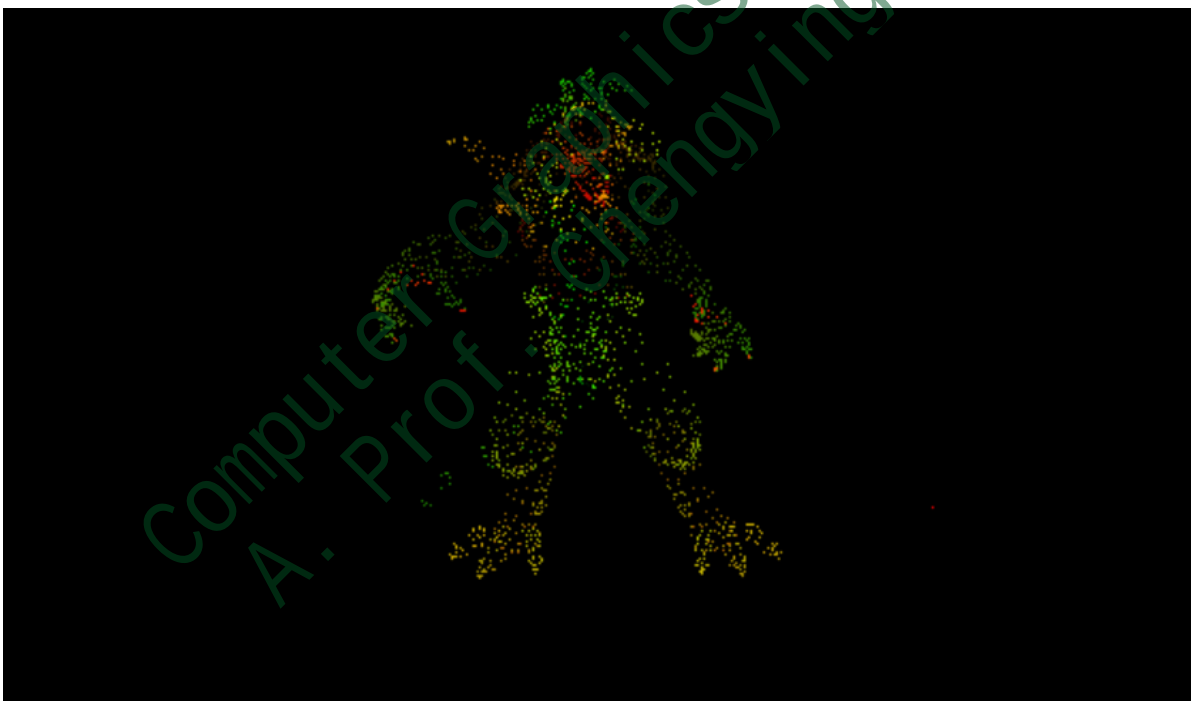
因此，`VertexData`的`spos`存储了该顶点在屏幕空间的`xy`坐标。这些变换都已经在渲染器中实现。同学们在实现光栅化时需要用的是对`VertexData`的线性插值函数`lerp`，其中`frac`是插值权重：

```
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。



3、作业描述

本次作业中，请你严格按照下面的顺序完成以下的任务：

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`）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)
{

    //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.

}
```

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

```
void TRShaderPipeline::rasterize_wire(
    const VertexData &v0,
    const VertexData &v1,
    const VertexData &v2,
    const unsigned int &screen_width,
    const unsigned int &screene_height,
    std::vector<VertexData> &rasterized_points)
{
    //Draw each line step by step
    rasterize_wire_aux(v0, v1, screen_width, screene_height, rasterized_points);
    rasterize_wire_aux(v1, v2, screen_width, screene_height, rasterized_points);
    rasterize_wire_aux(v0, v2, screen_width, screene_height, rasterized_points);
}
```

请贴出实现结果，并简述你是怎么做的。

Task 2、实现简单的齐次空间裁剪，简述你是怎么做的。参考链接

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）

```
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,
    //       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

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

请完成该函数，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，请返回`{}`。该裁剪函数在执行顶点着色之后、透视除法之前被调用，如果三角形被裁剪掉，则不会执行后续的光栅化和着色过程。**实现好之后，滑动鼠标滚轮将镜头拉近，将一些三角形推出屏幕之外，并注意窗口标题的`#ClipedFaces`的变化。**

注：本任务不需要你实现更为精细的Sutherland-Hodgeman裁剪算法。

Task3、实现三角形的背向面剔除，简述你是怎么做的。参考链接

关于面剔除的概念，请看这里。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，这就是背向面剔除的基本原理。

你要填充的函数在`TRRenderer.cpp`文件中，如下所示，其中传入的参数为ndc空间下的三角形顶点坐标。

```
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.

    return false;

}
```

该函数在光栅化之前被调用，可以剔除非常大一部分的三角形。**实现好之后，请仔细对比并贴出有面剔除和无面剔除的效果，并注意窗口标题的`#CulledFaces`变换情况。**

Task4、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：

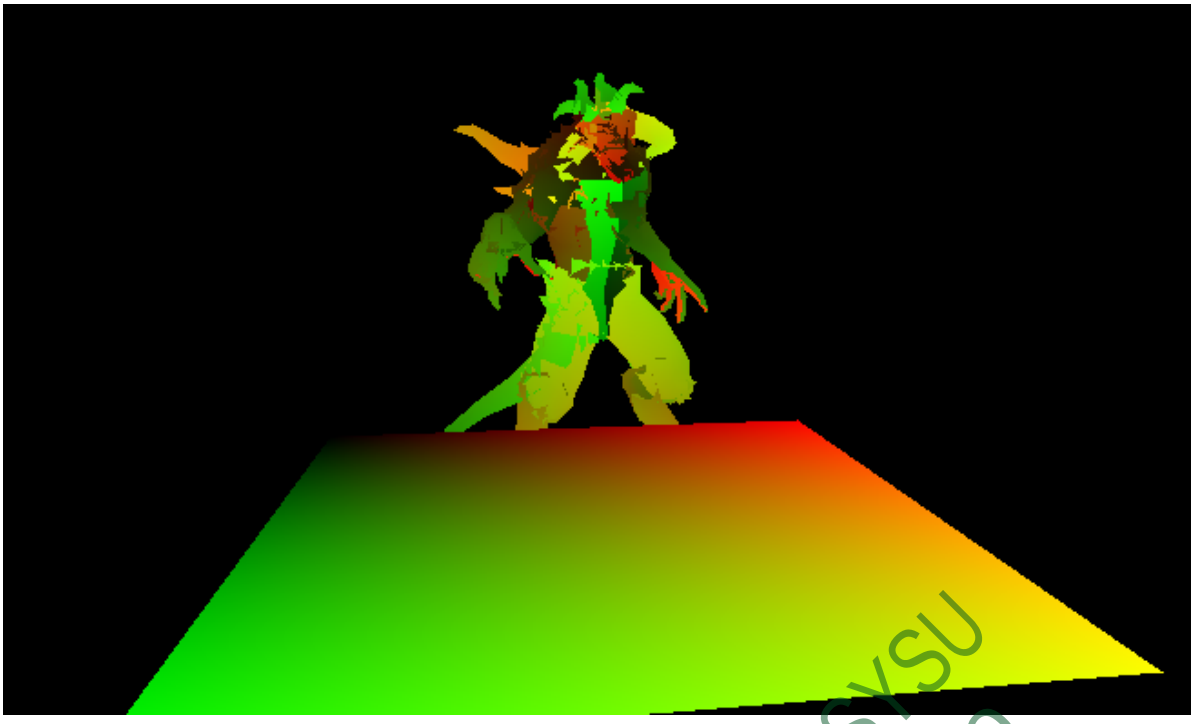
```
void TRShaderPipeline::rasterize_fill_edge_function(
    const VertexData &v0,
    const VertexData &v1,
    const VertexData &v2,
    const unsigned int &screen_width,
    const unsigned int &screene_height,
```

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

    //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.

}
```

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！



注：此任务并非要求实现Edge-walking算法，该算法比较古老，现在更流行Edge-function算法，容易并行。

Task5、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：

```
//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
    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!

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。

Task7、实现更为精细的齐次空间三角形裁剪（提高题，选做）。参考链接

在Task2中，只要有一个三角形在可见的范围内，我们就直接返回全部顶点。但实际上我们应该做更加精细的裁剪，把超出范围的给裁剪掉。这里涉及到稍微复杂一点的裁剪算法，如果你有兴趣、有能力，我们鼓励你尝试实现更精细的裁剪算法。

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。