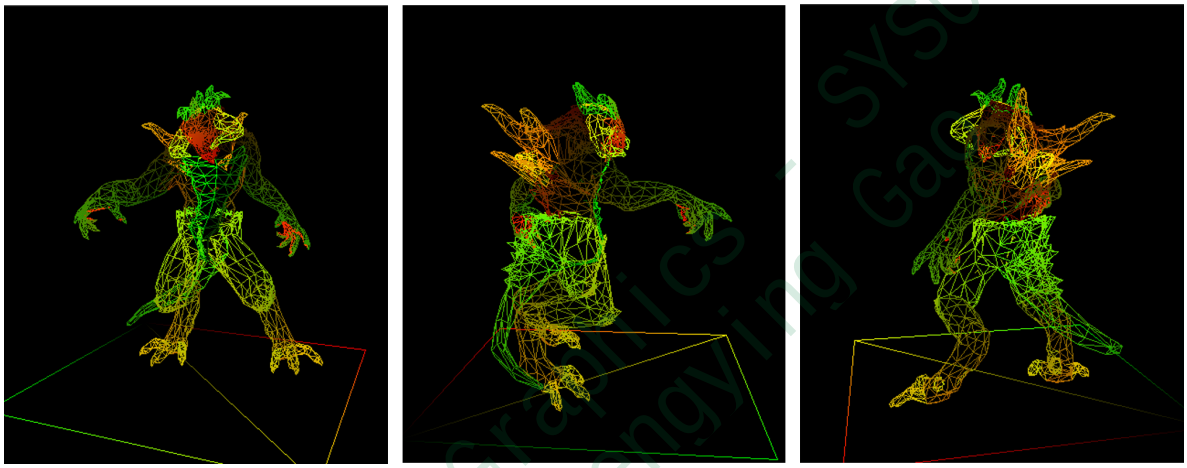
Assignment 1: 3D Transformation

Computer Graphics Teaching Stuff, Sun Yat-Sen University

Due Date: **10月12号晚上12点之前**

Submission: `sysu_cg_2022@163.com`

> 到目前为止，你们已经学习了如何使用矩阵变换来排列和投影三维空间中的物体到二维的屏幕上，因此本次作业是关于三维矩阵变换的具体实践。在接下来的作业中（包含本次），我们的代码框架提供了一个基于CPU的模拟OpenGL渲染管线的光栅化渲染器，你们需要在这个框架上去实现渲染器的一些特性，本次作业要求你实现软光栅化渲染器的三维变换部分。



本次作业你们将实现的效果图

1、作业概述

电子计算机通过二维的显示屏向用户展示其计算的内容和数据，透过这个二维的屏幕，我们能看到的并不仅仅是二维的内容，还有三维的广阔空间。这其中蕴含了怎样的三维空间变换到二维空间的奥妙？本次作业将带领你们通过手动的编程实践来切身体会三维空间的物体是如何显示到二维的屏幕上的。通过本次实践，你将领略到线性代数的魅力所在！

在目前的基于光栅化的渲染管线中，从三维的二维的转换，历经的变换矩阵包含**模型矩阵**、**观察矩阵**、**投影矩阵**和**视口变换矩阵**。学习了可编程管线的OpenGL之后，你应该知道这些变换通常是在顶点着色器发生的（视口变换在顶点着色器之后），矩阵变换作用的对象是三维物体的几何顶点数据。本次的作业要求你手动实现观察矩阵、投影矩阵和视口变换矩阵的计算，并学会用旋转、缩放等变换来对实现对三维物体的空间操纵。渲染器的绝大部分代码都已经实现好，你只需要在我们指定的地方填补代码空白即可，实际的代码量不大。

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`readme.pdf`文档。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，你只需了解GLM数学库的使用即可，而这在之前的`Assignment0`中已经学习使用过，所以本次作业不需要你们再去学习其他的前置内容。目录`CGAssignment1/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，无需修改；
- **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`main.cpp`文件，请大家着重注意这两个文件的代码。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：

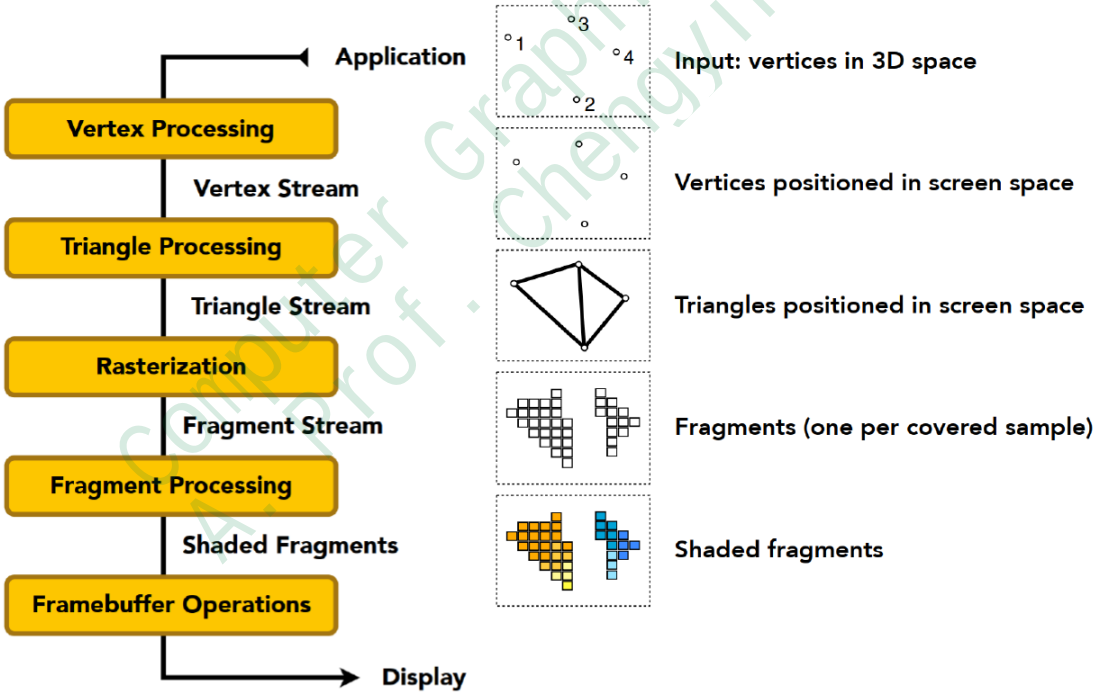


图1 基于光栅化的渲染管线

考虑到同学们的基础，同学们可能对渲染器的一些代码细节不是很理解，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！本次作业涉及的是几何顶点的变换（即图1的**Vertex Processing**阶段），涉及的顶点着色器代码我们已经写好：

```
void TRDefaultShaderPipeline::vertexShader(VertexData &vertex)
{
    //Local space -> World space -> Camera space -> Clip space
    vertex.pos = m_model_matrix * glm::vec4(vertex.pos.x, vertex.pos.y,
vertex.pos.z, 1.0f);
    vertex.spos = m_view_project_matrix * vertex.pos;
}
```

我们需要你完成的是`model`矩阵、`view`矩阵、`project`矩阵和`viewport`矩阵的计算。其中`viewport`矩阵是在光栅化之前使用，目的是将NDC空间的顶点数据变换到屏幕空间。顶点着色器的输入`VertexData`存储了几何顶点的数据，其中的`spos`存储投影变换后的顶点坐标。

```
class VertexData
{
public:
    glm::vec4 pos;
    glm::vec3 col;
    glm::vec3 nor;
    glm::vec2 tex;
    glm::vec4 spos; //Screen position
};
```

`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OpenGL理解到的渲染流程，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。

我们要求你的通过代码计算矩阵的每个元素的取值，然后对矩阵元素逐个逐个进行赋值。需要特别注意的是`glm::mat4`矩阵是列主序的，`mat[x][y]`表示的是第`x`列第`y`行的元素，而不是第`x`行第`y`列的元素。以下面的矩阵为例，假设计算得到的矩阵元素记为：

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix}$$

则在代码中，你应该用以下的方式进行赋值：

```
glm::mat4 vpMat;
vpMat[0][0] = m00; vpMat[0][1] = m10; vpMat[0][2] = m20; vpMat[0][3] = m30;
vpMat[1][0] = m01; vpMat[1][1] = m11; vpMat[1][2] = m21; vpMat[1][3] = m31;
vpMat[2][0] = m02; vpMat[2][1] = m12; vpMat[2][2] = m22; vpMat[2][3] = m32;
vpMat[3][0] = m03; vpMat[3][1] = m13; vpMat[3][2] = m23; vpMat[3][3] = m33;
```

最后，在正式开始之前，请注意翻到本文档最后阅读相关的注意事项！

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、实现观察矩阵（View Matrix）的计算，如下所示，该函数在`TRRenderer.cpp`文件中。

```
glm::mat4 TRRenderer::calcViewMatrix(glm::vec3 camera, glm::vec3 target,
glm::vec3 worldUp)
{
    //Setup view matrix (world space -> camera space)
    glm::mat4 vMat = glm::mat4(1.0f);

    //Task 1: Implement the calculation of view matrix, and then set it to vMat
    //  Note: You can use any glm function (such as glm::normalize, glm::cross,
glm::dot) except glm::lookAt

    return vMat;
}
```

该函数在`main.cpp`中被调用：

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.5f, 3.7f);
glm::vec3 lookAtTarget = glm::vec3(0.0f);
renderer->setViewMatrix(TRRenderer::calcViewMatrix(cameraPos, lookAtTarget,
glm::vec3(0.0, 1.0, 0.0f)));
```

简述你是怎么做的。

Task 2、实现透视投影矩阵（Project Matrix）的计算，如下所示，该函数在`TRRenderer.cpp`文件中。

```
glm::mat4 TRRenderer::calcPerspProjectMatrix(float fovy, float aspect, float
near, float far)
{
    //Setup perspective matrix (camera space -> clip space)
    glm::mat4 pMat = glm::mat4(1.0f);

    //Task 2: Implement the calculation of perspective matrix, and then set it
to pMat
    //  Note: You can use any math function (such as std::tan) except
glm::perspective

    return pMat;
}
```

该函数在`main.cpp`中被调用：

```
renderer->setProjectMatrix(TRRenderer::calcPerspProjectMatrix(45.0f,
(float)width/ height, 0.0, 50.0f));
```

简述你是怎么做的。

Task3、实现视口变换矩阵（Viewport Matrix）的计算，如下所示，该函数在`TRRenderer.cpp`文件中。

```
glm::mat4 TRRenderer::calcViewPortMatrix(int width, int height)
{
    //Setup viewport matrix (ndc space -> screen space)
    glm::mat4 vpMat = glm::mat4(1.0f);

    //Task 3: Implement the calculation of viewport matrix, and then set it to
vpMat

    return vpMat;
}
```

该函数在`TRRenderer`的构造函数中被调用。简述你是怎么做的。如果**Task1**、**Task2**和**Task3**实现正确，那么你的运行的效果将是本文档开头贴出来的图片，而且三维物体随着程序的运行绕y轴在不停地旋转。你可以通过按住鼠标左键并横向拖动鼠标来旋转摄像机，通过滚动鼠标的滚轮来拉近或拉远摄像机的位置。请贴出你的实现效果。

Task4、程序默认物体绕着y轴不停地旋转，请你在`main.cpp`文件中稍微修改一下代码，使得物体分别绕x轴和z轴旋转。贴出你的结果。

```
static glm::mat4 model_mat = glm::mat4(1.0f);
//Rotation
{
    //Task 4: Make the model rotate around x axis and z axis, respectively
    model_mat = glm::rotate(model_mat, (float)deltaTime * 0.001f, glm::vec3(0,
1, 0));
}
```

Task5、仔细体会使物体随着时间的推进不断绕y轴旋转的代码，现在要你用`glm::scale`函数实现物体不停地放大、缩小、放大的循环动画，物体先放大至2.0倍、然后缩小至原来的大小，然后再放大至2.0，依次循环下去，要求缩放速度适中，不要太快也不要太慢。贴出你的效果，说说你是怎么实现的。（请注释掉前面的旋转代码）

```
//Scale
{
    //Task 5: Implement the scale up and down animation using glm::scale
function

}
```

Task6、现在要求你实现正交投影矩阵的计算，如下所示，该函数在`TRRenderer.cpp`文件中。

```
glm::mat4 TRRenderer::calcOrthoProjectMatrix(float left, float right, float
bottom, float top, float near, float far)
{
    //Setup orthogonal matrix (camera space -> homogeneous space)
    glm::mat4 pMat = glm::mat4(1.0f);

    //Task 6: Implement the calculation of orthogonal projection, and then set
it to pMat

    return pMat;
}
```

简述你是怎么做的。

Task7、实现了正交投影计算后，在`main.cpp`的如下代码中，分别尝试调用透视投影和正交投影函数，通过滚动鼠标的滚轮来拉近或拉远摄像机的位置，仔细体会这两种投影的差别。

```
// Task 6: try it with different kind of projection
{
    renderer->setProjectMatrix(TRRenderer::calcPerspProjectMatrix(45.0f,
static_cast<float>(width) / height, 0.0, 50.0f));
    //renderer->setProjectMatrix(TRRenderer::calcOrthoProjectMatrix(-2.0f,
+2.0f, -2.0f, +2.0f, 0.0, 50.0f));
}
```

Task8、完成上述的编程实践之后，请你思考并回答以下问题：

（1）请简述正交投影和透视投影的区别。

（2）从物体局部空间的顶点的顶点到最终的屏幕空间上的顶点，需要经历哪几个空间的坐标系？裁剪空间下的顶点的w值是哪个空间坐标下的z值？它有什么空间意义？

（3）经过投影变换之后，几何顶点的坐标值是被直接变换到了NDC坐标（即xyz的值全部都在$[-1,1]$范围内）吗？透视除法（Perspective Division）是什么？为什么要有这么一个过程？

参考链接1

参考链接2

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+HW1，例如19214044+张三+HW1.zip。
- **提交的文档请提交编译生成的pdf文件，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！）。**
- 禁止直接调用`glm::lookAt`、`glm::perspective`、`glm::ortho`等的取巧行为。
- 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。
- 推荐参考博客：软渲染器Soft Renderer：3D数学篇。