计算机图形学实验报告

题目

1. 画一个立方体(cube)：边长为4， 中心位置为(0, 0, 0)。分别启动和关闭深度测试 `glEnable(GL_DEPTH_TEST)`、`glDisable(GL_DEPTH_TEST)`，查看区别，并分析原因。
2. 平移(Translation)：使画好的cube沿着水平或垂直方向来回移动。
3. 旋转(Rotation)：使画好的cube沿着XoZ平面的x=z轴持续旋转。
4. 放缩(Scaling)：使画好的cube持续放大缩小。
5. 在GUI里添加菜单栏，可以选择各种变换。
6. 结合Shader谈谈对渲染管线的理解

Hint: 可以使用GLFW时间函数`glfwGetTime()`，或者`<math.h>`、`<time.h>`等获取不同的数值

Bonus:

1. 将以上三种变换相结合，打开你们的脑洞，实现有创意的动画。比如：地球绕太阳转等。

作业要求：

1. 把运行结果截图贴到报告里，并回答作业里提出的问题。
2. 报告里简要说明实现思路，以及主要function/algorithm的解释。
3. 虽然learnopengl教程网站有很多现成的代码，但是希望大家全部手打，而不是直接copy。

引入GLM库

利用 openGL 进行 3D 绘图需要用到大量的数学矩阵运算，而 OpenGL 没有自带任何的矩阵和向量知识，需要我们自己定义数学类和函数，这相对比较麻烦。所以我们需要引入 GLM 库，GLM 能快速帮助我们实现各种数学矩阵运算。

前往 GLM官方github仓库，选择0.9.8.5版本，下载该版本的 glm-0.9.8.5.zip 或 glm-0.9.8.5.zip 压缩包，解压之后，在项目里引用以下文件即可：

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

画立方体

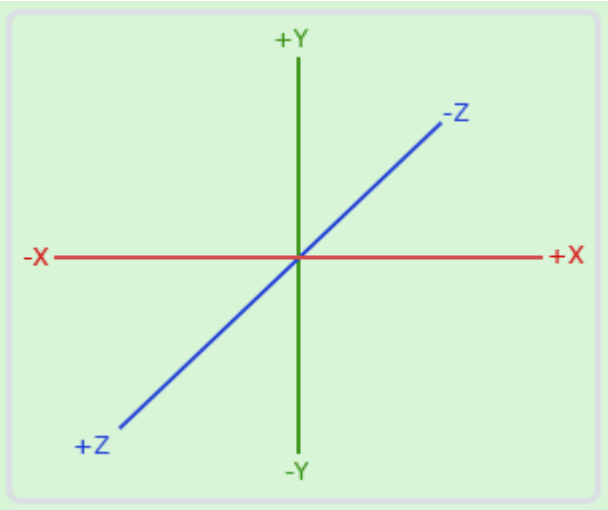
模型、观察和投影

利用 openGL 进行 3D 绘图时，首先要定义一个模型矩阵 model，这个矩阵包含了对 3D 物体的位移、缩放与旋转操作。这个模型矩阵创建如下：

```
glm::mat4 model;
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

然后，我们需要创建一个观察矩阵，因为我们创建的 3D 物体默认在世界坐标的原点(0, 0, 0)，而摄像机的位置也是(0, 0, 0)，所以，为了能看清楚3D物体的全貌，我们必须向前移动整个场景，这就是观察矩阵的作用。

在 openGL 中，世界坐标系的坐标表示为(x, y, z)，分别对应下图位置：



注意，将摄像机向后移动，和将整个场景向前移动是一样的。这里我们将场景向 z 轴的负方向移动，以便于摄像机能观察到物体。

```
glm::mat4 view;
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

最后我们需要做的是定义一个投影矩阵。我们希望在场景中使用透视投影，所以像这样声明一个投影矩阵：

```
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), screenWidth / screenHeight,
0.1f, 100.0f);
```

修改着色器

我们创建了模型、观察和投影矩阵，应该把它们传入着色器，着色器的修改如下：

```
const char* vertexShaderSource = "#version 330 core\n"
        "layout (location = 0) in vec3 aPos;\n"
        "layout (location = 1) in vec3 aColor;\n"
        "out vec3 ourColor;\n"
```

```
        "uniform mat4 model;\n"
        "uniform mat4 view;\n"
        "uniform mat4 projection;\n"
        "void main()\n"
        "{\n"
        "gl_Position = projection * view * model * vec4(aPos, 1.0);\n"
        "ourColor = aColor;\n"
        "}\0";
```

然后再程序中获取 uniform 变量，并赋值

```
//获取着色器程序uniform
unsigned int modelLoc = glGetUniformLocation(shaderProgram, "model");
unsigned int viewLoc = glGetUniformLocation(shaderProgram, "view");
unsigned int projectionLoc = glGetUniformLocation(shaderProgram, "projection");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]);
glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, &projection[0][0]);
```

立方体的顶点

要想渲染一个立方体，我们一共需要36个顶点（6个面 x 每个面有2个三角形组成 x 每个三角形有3个顶点），顶点如下：

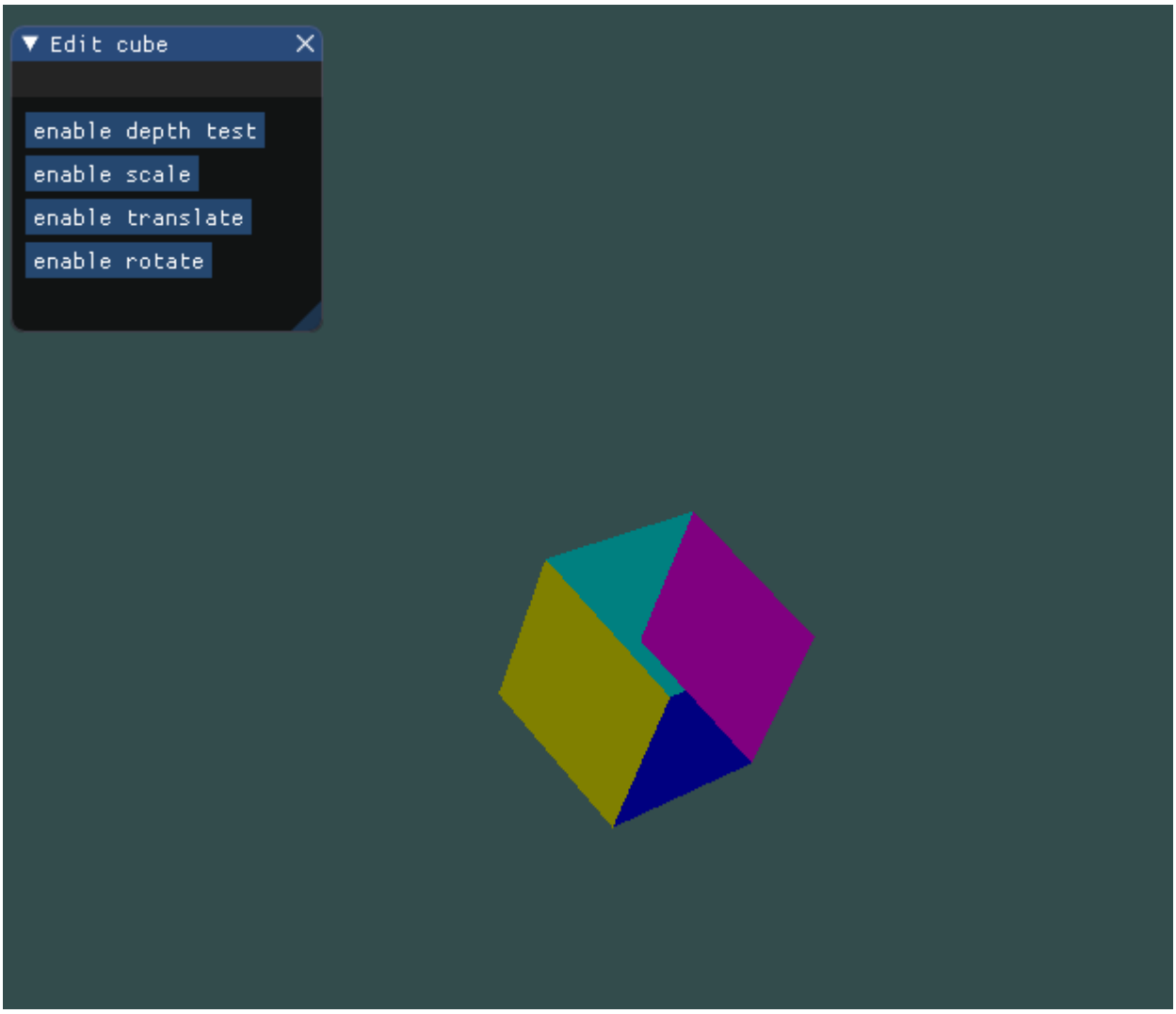
```
float vertices[] = {
    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,

    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
    -0.5f,  0.5f,  0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,

    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
```

```
        0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

       -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        0.5f, -0.5f, -0.5f,  1.0f, 1.0f,
        0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
        0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
       -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
       -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,

       -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
        0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
       -0.5f,  0.5f,  0.5f,  0.0f, 0.0f,
       -0.5f,  0.5f, -0.5f,  0.0f, 1.0f
  };
```

由于题目要求立方体边长为4，所以上述代码中，把0.5全部换成2即可。由于立方体边长增大，最终可能无法看清立方体全貌，所以需要把场景再往前移动，即观察矩阵中，增加场景向 z 轴的负方向移动的距离。

深度测试

完成之前的步骤，便可以画出一个立方体，如下图：



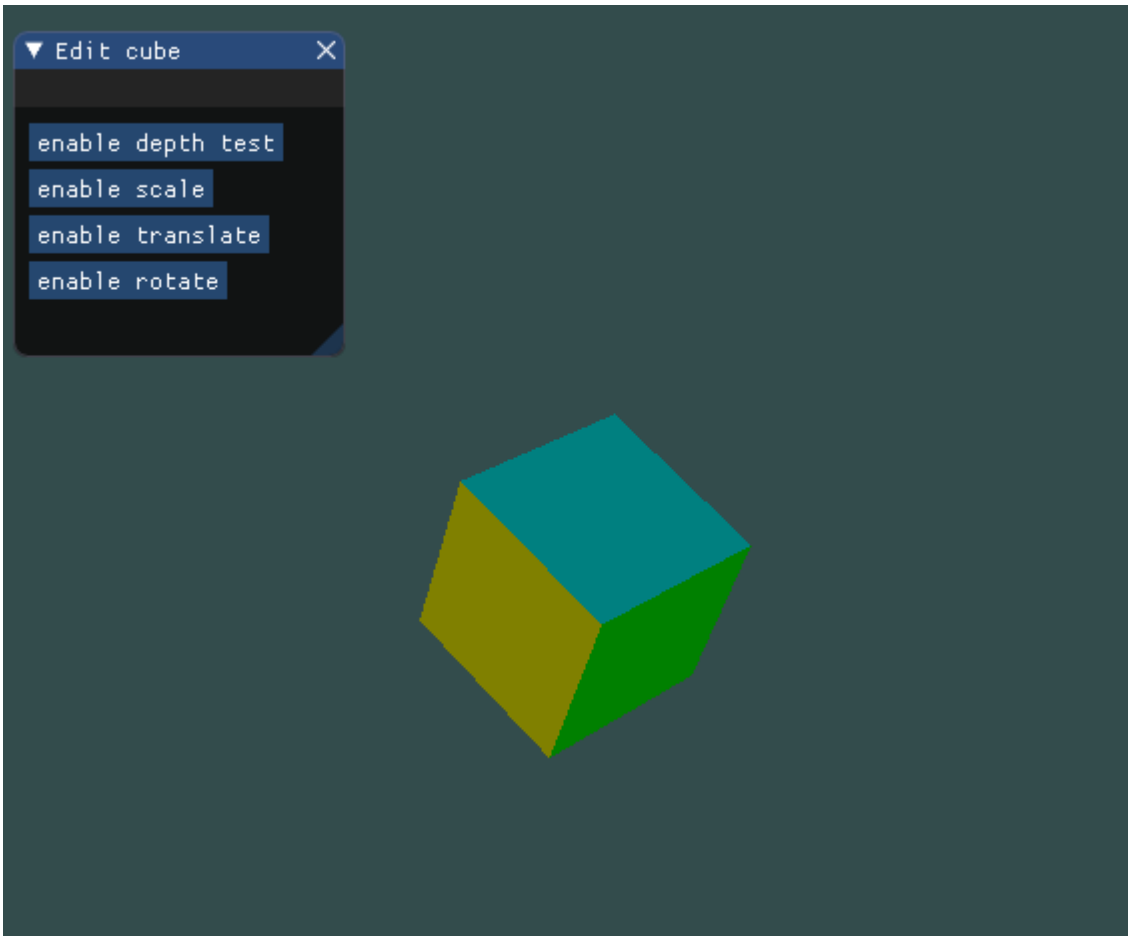
但可以看到，立方体的某些本应被遮挡住的面被绘制在了这个立方体其他面之上。之所以这样是因为 OpenGL 是一个三角形一个三角形地来绘制你的立方体的，所以即便之前那里有东西它也会覆盖之前的像素。因为这个原因，有些三角形会被绘制在其它三角形上面，虽然它们本不应该是被覆盖的。

为了解决这个问题，我们需要开启深度测试。OpenGL 存储图形的所有深度信息于一个 Z 缓冲中，也被称为深度缓冲。GLFW 会自动为你生成这样一个缓冲。深度值存储在每个片段里面，当片段想要输出它的颜色时，OpenGL 会将它的深度值和 z 缓冲进行比较，如果当前的片段在其它片段之后，它将会被丢弃，否则将会覆盖。

使用以下命令开启深度测试：

```
glEnable(GL_DEPTH_TEST);
```

最终的结果如下图：



立方体变换

平移

使用以下函数实现平移：

```
model = glm::translate(model, glm::vec3(posDelta, 0, 0));
```

该函数的第一个参数就是之前创建的模型矩阵 model，第二个参数是个三维向量，分别对应 x、y、z 的位置。我在代码中通过自增和自减改变 posDelta 的值，从而实现立方体在 x 轴上来回移动。

旋转

使用以下函数实现旋转：

```
model = glm::rotate(model, rotateAngle, glm::vec3(1.0f, 0.0f, 1.0f));
```

该函数的第一个参数就是之前创建的模型矩阵 model，第二个参数是旋转的弧度，这里我利用弧度的周期为 2pi，让变量 rotate 不断自增，实现立方体不断旋转。函数的第三个参数是旋转的轴，这里选择了 xoz 平面上 x=z 这条轴。

放缩

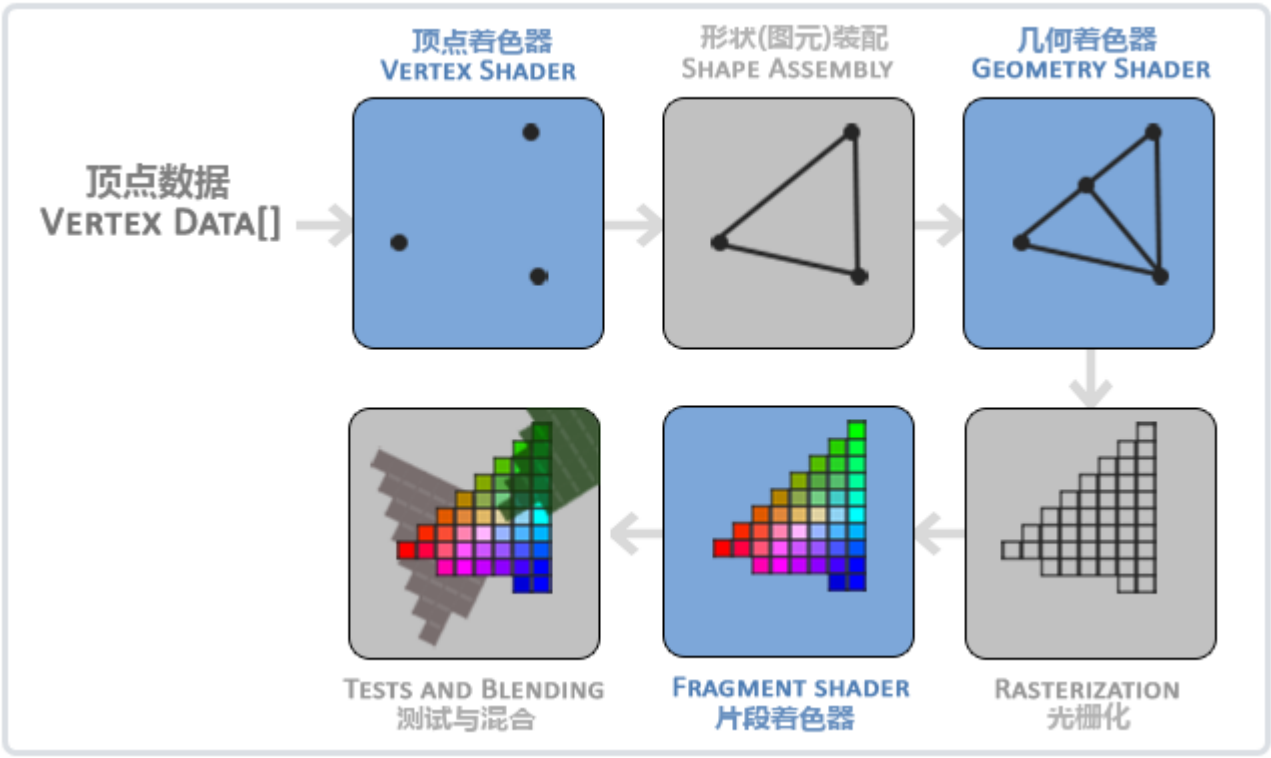
使用以下函数实现放缩：

```
model = glm::scale(model, glm::vec3(scaleDelta, scaleDelta, scaleDelta));
```

该函数的第一个参数就是之前创建的模型矩阵 model，第二个参数是个三位向量，对应 x、y、z三个方向放缩的倍数。

渲染管线的理解

下图可以抽象地展示出图形渲染管线的各个步骤：



首先，顶点数据是一系列顶点的集合。一个顶点是一个3D坐标的数据的集合。而顶点数据是用顶点属性表示的，它可以包含任何我们想用的数据。

图形渲染管线的第一个部分是顶点着色器，它把一个单独的顶点作为输入。顶点着色器主要的目的是把3D坐标转为另一种3D坐标，同时顶点着色器允许我们对顶点属性进行一些基本处理。

图元装配阶段将顶点着色器输出的所有顶点作为输入，并所有的点装配成指定图元的形状。

几何着色器把图元形式的一系列顶点的集合作为输入，它可以通过产生新顶点构造出新的图元来生成其他形状。

光栅化阶段会把图元映射为最终屏幕上相应的像素，生成供片段着色器使用的片段。在片段着色器运行之前会执行裁切。裁切会丢弃超出你的视图以外的所有像素，用来提升执行效率。

片段着色器的主要目的是计算一个像素的最终颜色，这也是所有 OpenGL 高级效果产生的地方。通常，片段着色器包含3D场景的数据（比如光照、阴影、光的颜色等等），这些数据可以被用来计算最终像素的颜色。

Alpha测试和混合阶段。这个阶段检测片段的对应的深度值，用它们来判断这个像素是其它物体的前面还是后面，决定是否应该丢弃。这个阶段也会检查alpha值（alpha值定义了一个物体的透明度）并对物体进行混合。