Homework 4 - Transformation

16340256 **谢玮鸿**

Basic:

> 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

（一）绘制立方体

首先考虑一个立方体所包含的数据，即是8个顶点数据。要画出一个立方体，需要一个一个面地画；而每一个面是一个正方形，可以划分成两个三角形。也就是说，一个立方体由6*2个三角形画成。在之前的作业中，已经能比较熟悉地画出一个三角形了，现在绘制多个三角形已构成一个立方体。

为了减少开销，使用索引缓冲对象(Element Buffer Object, EBO)，这样只需要存储8个顶点数据就行了，而顶点索引用于存储12个三角形的顶点信息。顶点数据以及顶点索引如下：

```
// 顶点数据 和 顶点索引
float vertices[] = {
    // 位置                 颜色
    -2.0f, -2.0f, -2.0f,    0.4f, 1.0f, 0.4f,
     2.0f, -2.0f, -2.0f,    1.0f, 0.4f, 0.4f,
     2.0f,  2.0f, -2.0f,    0.4f, 0.4f, 1.0f,
    -2.0f,  2.0f, -2.0f,    0.5f, 0.5f, 0.5f,

    -2.0f, -2.0f,  2.0f,    1.0f, 0.4f, 0.4f,
     2.0f, -2.0f,  2.0f,    0.4f, 1.0f, 0.4f,
     2.0f,  2.0f,  2.0f,    0.5f, 0.5f, 0.5f,
    -2.0f,  2.0f,  2.0f,    0.4f, 0.4f, 1.0f
};

unsigned int indices[] = {
    0, 1, 2,    // 前面-1
    0, 2, 3,    // 前面-2
    4, 5, 6,    // 后面-1
    4, 6, 7,    // 后面-2

    0, 3, 7,    // 左面-1
    0, 4, 7,    // 左面-2
    1, 2, 6,    // 右面-1
    1, 5, 6,    // 右面-2

    0, 1, 5,    // 下面-1
    0, 4, 5,    // 下面-2
    2, 3, 6,    // 上面-1
    3, 6, 7     // 上面-2
};
```

然后创建索引缓冲对象，并绑定EBO，然后用 `glBufferData` 把索引复制到缓冲里。

```
// 创建EBO
unsigned int EBO;
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

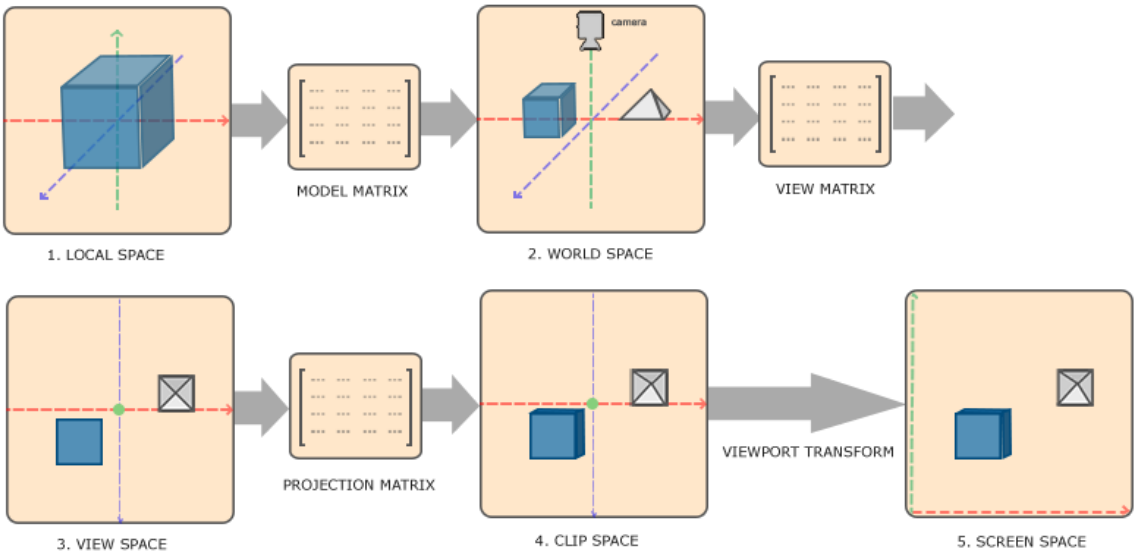
在绘制图的时候，将索引个数更改为36个即可(12个三角形*3个顶点)。

```
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);       // 索引个数 36
```

这时候绘制出来的立方体，在屏幕上显示只是一个平面，为了观察更加自然，接下来先做坐标变换。

（二） 坐标变换

为了将坐标从一个坐标系变换到另一个坐标系，我们需要用到几个变换矩阵，最重要的几个分别是模型(Model)、观察(View)、投影(Projection)三个矩阵。顶点坐标起始于局部空间(Local Space)，在这里它称为局部坐标(Local Coordinate)，它在之后会变为世界坐标(World Coordinate)，观察坐标(View Coordinate)，裁剪坐标(Clip Coordinate)，并最后以屏幕坐标(Screen Coordinate)的形式结束。



1. 在开始进行3D绘图时，我们首先创建一个模型矩阵。这个模型矩阵包含了位移、缩放与旋转操作，它们会被应用到所有物体的顶点上，以变换它们到全局的世界空间。
2. 接下来我们需要创建一个观察矩阵。我们想要在场景里面稍微往后移动，以使得物体变成可见的（当在世界空间时，我们位于原点(0,0,0)）。过将场景沿着z轴负方向平移来实现。它会给我们一种我们在往后移动的感觉。
3. 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，使得离你越远的东西看起来更小。

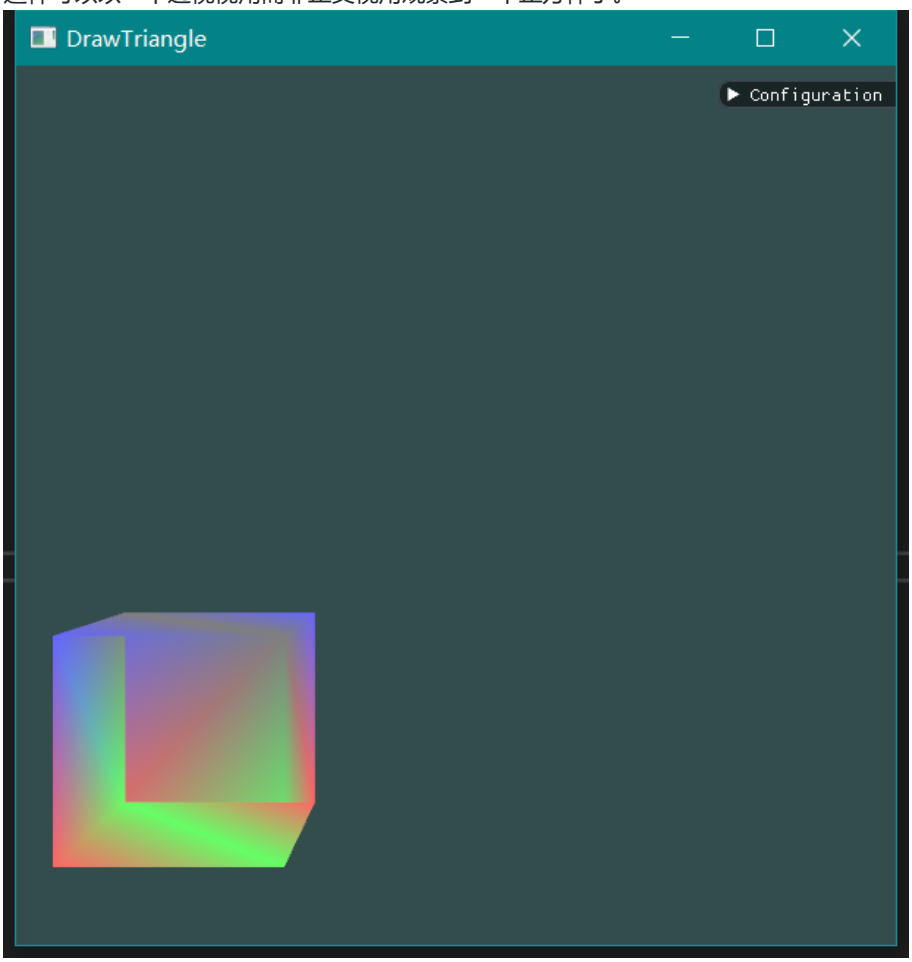
值得注意的是，矩阵传入着色器通常在每次的渲染迭代中进行，因为变换矩阵会经常变动。

```
// 观察矩阵
glm::mat4 view = glm::mat4(1.0f);
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -10.0f));

// 投影矩阵
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);

// 模型矩阵
glm::mat4 model = glm::mat4(1.0f);

// ...

// 将矩阵传入着色器
unsigned int modelLoc = glGetUniformLocation(shaderProgram, "model");
unsigned int viewLoc = glGetUniformLocation(shaderProgram, "view");
unsigned int projLoc = glGetUniformLocation(shaderProgram, "projection");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));
```

矩阵传入着色器后，通过矩阵相乘实现坐标变换。顶点着色器定义如下：

```
// 顶点着色器
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, view, projection;\n"
"void main() {\n"
"   gl_Position = projection * view * model * vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"   ourColor = aColor;\n"
"}\0";
```

这样可以以一个透视视角而非正交视角观察到一个正方体了。

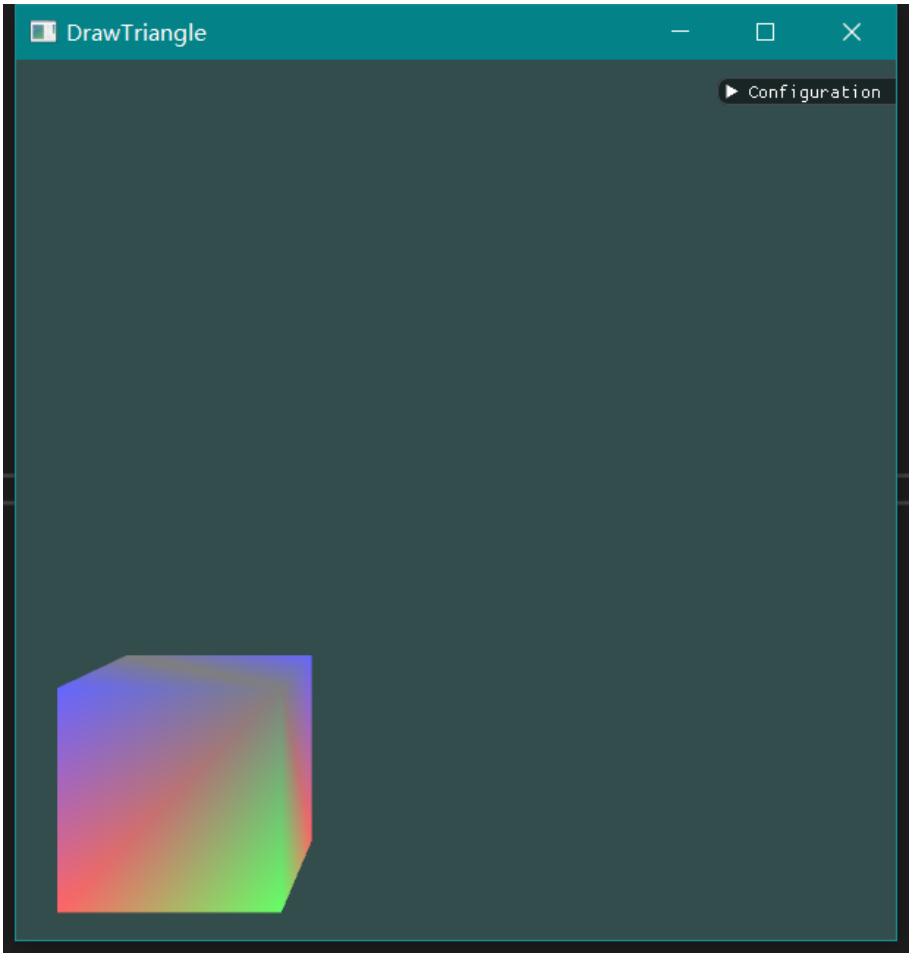


（三）深度测试

OpenGL存储它的所有深度信息于一个Z缓冲(Z-buffer)中，也被称为深度缓冲(Depth Buffer)。深度值存储在每个片段里面（作为片段的z值），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。这个过程称为深度测试(Depth Testing)，它是由OpenGL自动完成的。直观来说，就是观察者只会观察到离观察者近的一面，其之后的都会被覆盖，相当于平时观察物体一样。

深度测试是默认关闭的（使用代码 `glDisable(GL_DEPTH_TEST)` 是一样的效果），如上图所示，会看到整个立方体6个面。启用深度测试，需要开启GL_DEPTH_TEST： `glEnable(GL_DEPTH_TEST)` 。另外，还需要在每次渲染迭代之前清除深度缓冲（否则前一帧的深度信息仍然保存在缓冲中），通过glClear函数完成： `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` 。

开启深度测试后，就可以看到一个这样的立方体，与关闭深度测试时绘制的立方体对比，不再有"透视眼"效果了。



(四) 平移、旋转、放缩变换

使用GLM库（0.9.9版本），方便进行矩阵运算，并结合ImGui，选择各种变换，分为自动挡和手动挡。先前提到过，模型矩阵包含了位移、缩放与旋转操作，它们会被应用到所有物体的顶点上，以变换它们到全局的世界空间。

> glm::mat4 model = glm::mat4(1.0f);

首先是平移变换。通过变换参数 `trans_x` 和 `trans_y` ，更改立方体x/y坐标，进行水平或垂直方向上的移动。把模型矩阵和一个位移向量传递给 `glm::translate` 函数来完成这个工作的，即 `model = glm::translate(model, glm::vec3(trans_x, trans_y, 0.0f));` 。也可以通过勾选ImGui上的AutoTranslating的选项，让立方体在水平方向上(-2.0f - 2.0f之间)来回移动，实现代码如下：

```
// 平移
if (autoTrans) {
    model = glm::translate(model, glm::vec3(trans_x, trans_y, 0.0f));
    trans_x += transArg * 0.02;
    if (trans_x > 2.0f)
        transArg = -1;
    else if (trans_x < -2.0f)
        transArg = 1;
}
else
    model = glm::translate(model, glm::vec3(trans_x, trans_y, 0.0f));
```

放缩变换同理，换成 `glm::scale` 函数即可。除了手动缩放，也勾选ImGui上的AutoScaling的选项，让立方体0.8倍和1.2倍大小之间自动放缩，实现代码如下：

```
// 缩放
if (autoScale) {
    model = glm::scale(model, glm::vec3(scale, scale, scale));
    scale += scaleArg * 0.01;
    if (scale > 1.2f)
        scaleArg = -1;
    else if (scale < 0.8f)
```

```
            scaleArg = 1;
    }
    else
        model = glm::scale(model, glm::vec3(scale, scale, scale));
```

最后是旋转变换，除了变换矩阵，还需要提供 旋转角度 和 转轴 的参数。使用GLFW的时间函数来获取不同时间的角度，并让立方体沿着XoZ平面的x=z轴持续旋转。

```
    // 旋转
    if (autoRotate)
        model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(1.0f, 0.0f, 1.0f));  // 绕XoZ轴旋转
```

非常值得注意的一点是，如果旋转和平移操作共同进行，在写代码的时候要注意先后顺序。由于矩阵计算顺序的原因，实际出来的效果和代码的先后顺序是反过来的。比如说，要是先写了旋转部分，再写平移部分，会出现立方体平移后再绕转轴旋转；反之，立方体会绕轴旋转了再连同轴一起平移。

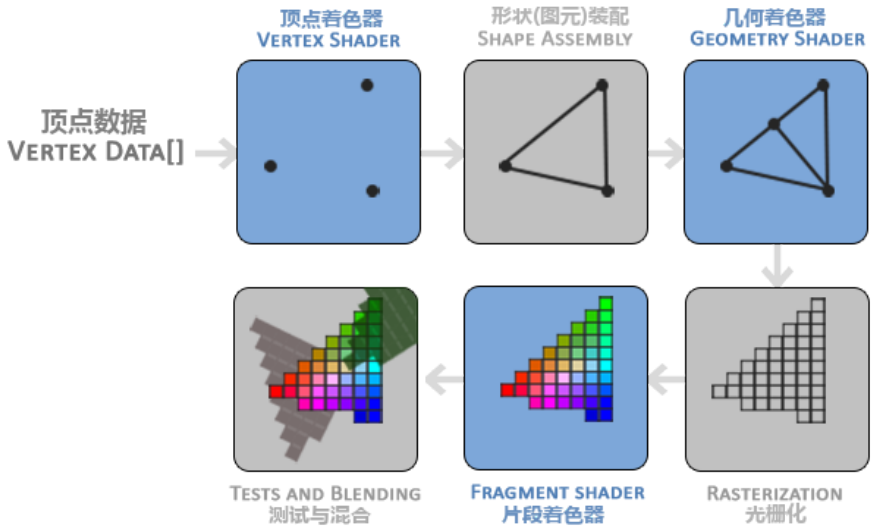
呈现的效果请看演示视频。演示一共包括了：水平+垂直平移、自动水平平移、缩放、自动来回缩放、旋转 以及 各种组合运动；还有包括Bonus作业的演示，是立方体自转+公转的效果

（五） 结合shader对渲染管线的理解

首先需要明确的是，在OpenGL中，所有东西都是在3D空间中的，但是屏幕却只能显示2D像素数组，这就需要OpenGL将3D坐标转变成适应屏幕的2D像素。这一个处理过程就是由OpenGL的**图形渲染管线(Graphics Pipeline)**管理的。

> 图形渲染管线可以被划分为两个主要部分：第一部分把3D坐标转换为2D坐标，第二部分是把2D坐标转变为实际的有颜色的像素。

图形渲染管线可以被划分为几个阶段，每个阶段将前一阶段的输出作为输入。它们在GPU上为每一个渲染管线阶段运行各自的小程序，快速处理数据。而这些小程序叫做**着色器(Shader)**。下图是图形渲染管线每个阶段的抽象展示，其中**蓝色部分**代表的是着色器部分。



图形渲染管线的工作方式大概是：以数组形式传递3D坐标，作为图形渲染管线的输入。这个数组称为顶点数据，顶点由3D位置、颜色等属性组成。然后指定数据所表示的渲染类型，如点、线、三角形，称之为 图元。

1. 顶点着色器：把3D坐标转为另一种3D坐标（后面会解释），同时顶点着色器允许我们对顶点属性进行一些基本处理。
2. 图元装配：将所有的点装配成指定图元的形状。
3. 几何着色器：可以通过产生新顶点构造出新的（或是其它的）图元来生成其他形状。
4. 光栅化阶段：这里它会把图元映射为最终屏幕上相应的像素。
5. 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。
6. Alpha测试和混合(Blending)阶段。这个阶段检测片段的对应的深度（和模板(Stencil)）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(Blend)。

以上，大概就是渲染管线的过程。

Bonus:

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。

结合缩放、平移和旋转操作，作出立方体绕自身中心自转 + 绕z轴公转的运动。实现如下：

```
if (revolution) {
    model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));  // 绕Z轴旋转
    model = glm::translate(model, glm::vec3(3.0f, 0.0f, 0.0f));
    model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));  // 绕自身中心旋转
    model = glm::scale(model, glm::vec3(0.3f, 0.3f, 0.3f));
}
```

阅读的时候按照代码相反顺序阅读即可，首先将立方体边长缩小到0.3倍，绕自身的z轴中心转动，即实现自转；然后向x轴正方向平移一段距离，再绕屏幕中心的z轴旋转，即实现公转。