Homework 4 - Transformation

16340082 黄俊凯

立方体

因为 OpenGL 中没有矩形这种图元，我们可以用两个三角形来刻画立方体的一个面，所以画一个立方体需要12个三角形，在不使用EBO的情况下，我们需要用36个点来描述立方体，因为此时需要深度信息，即点在 z 轴上的取值不等于0，我们需要对其进行 Projection 的操作来将3D空间的点投影到2D平面上，同时为了让立方体看起来更真实，我们使用的是透视（Perspective）投影，即离我们越远的东西看起来越小。

实现

局部空间

立方体的边长为4，且中心点在 `(0,0,0)`

其中一个面的顶点坐标：

```
float vertices[] = {
    -2.0f, -2.0f, -2.0f, 1.0f, 0.0f, 0.0f,
     2.0f, -2.0f, -2.0f, 1.0f, 0.0f, 0.0f,
     2.0f,  2.0f, -2.0f, 1.0f, 0.0f, 0.0f,
     2.0f,  2.0f, -2.0f, 1.0f, 0.0f, 0.0f,
    -2.0f,  2.0f, -2.0f, 1.0f, 0.0f, 0.0f,
    -2.0f, -2.0f, -2.0f, 1.0f, 0.0f, 0.0f,
```

世界空间

从局部坐标系到空间坐标系的转化，只需要让顶点坐标与 `Model` 矩阵相乘即可，因为我们现在只有一个物体，所以并不需要将立方体平移，但为了让我们更好地观察它，我们可以让它沿着 `x=z` 这条直线旋转45度，暴露出更多的面：

```
model = glm::rotate(model, glm::radians(45.0f), glm::vec3(1.0f, 0.0f, 1.0f));
```

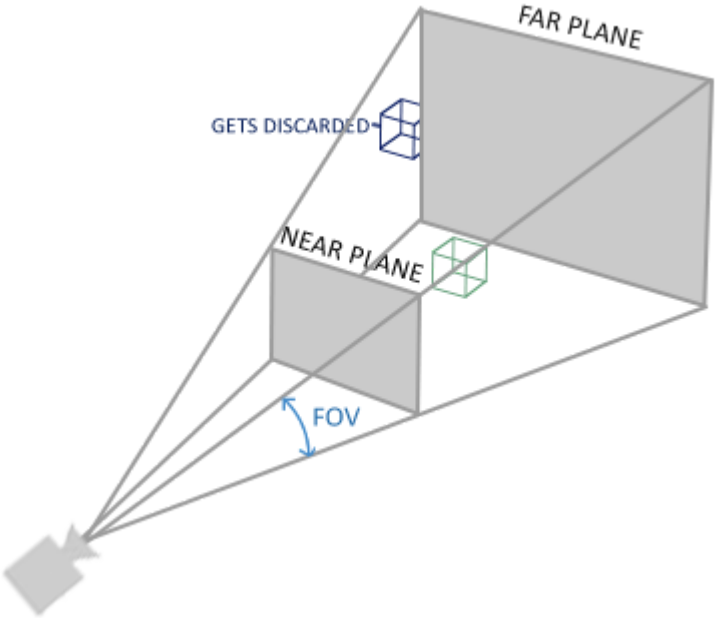
观察空间

观察空间是让世界空间坐标转化为用户视野前方的坐标而产生的结果，因为摄像机处在原点这个位置，我们会看不到完整的立方体，一个想法就是将我们带着摄像机后退几步，那反过来呢，也可以让立方体往前走几步，于是我们可以用一个 `View` 矩阵，在这里它只是一个平移矩阵，将其与立方体顶点坐标相乘，即可让立方体往前平移：

```
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -20.0f));
```

裁剪空间

OpenGL期望所有的坐标都能落在一个特定的范围内，且任何在这个范围之外的点都应该被裁剪掉，这里我们使用的是透视投影：



落在 `Near Plane` 和 `Far Plane` 之间这个棱台空间里的物体都可以被摄像机拍到，通过一个透视投影矩阵 `Projection` 即可完成投影操作：

```
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(45.0f), (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);
```

Vertex Shader

以上的这些矩阵，我们应该将其传入 `Vertex Shader` 以完成顶点坐标轴的变换：

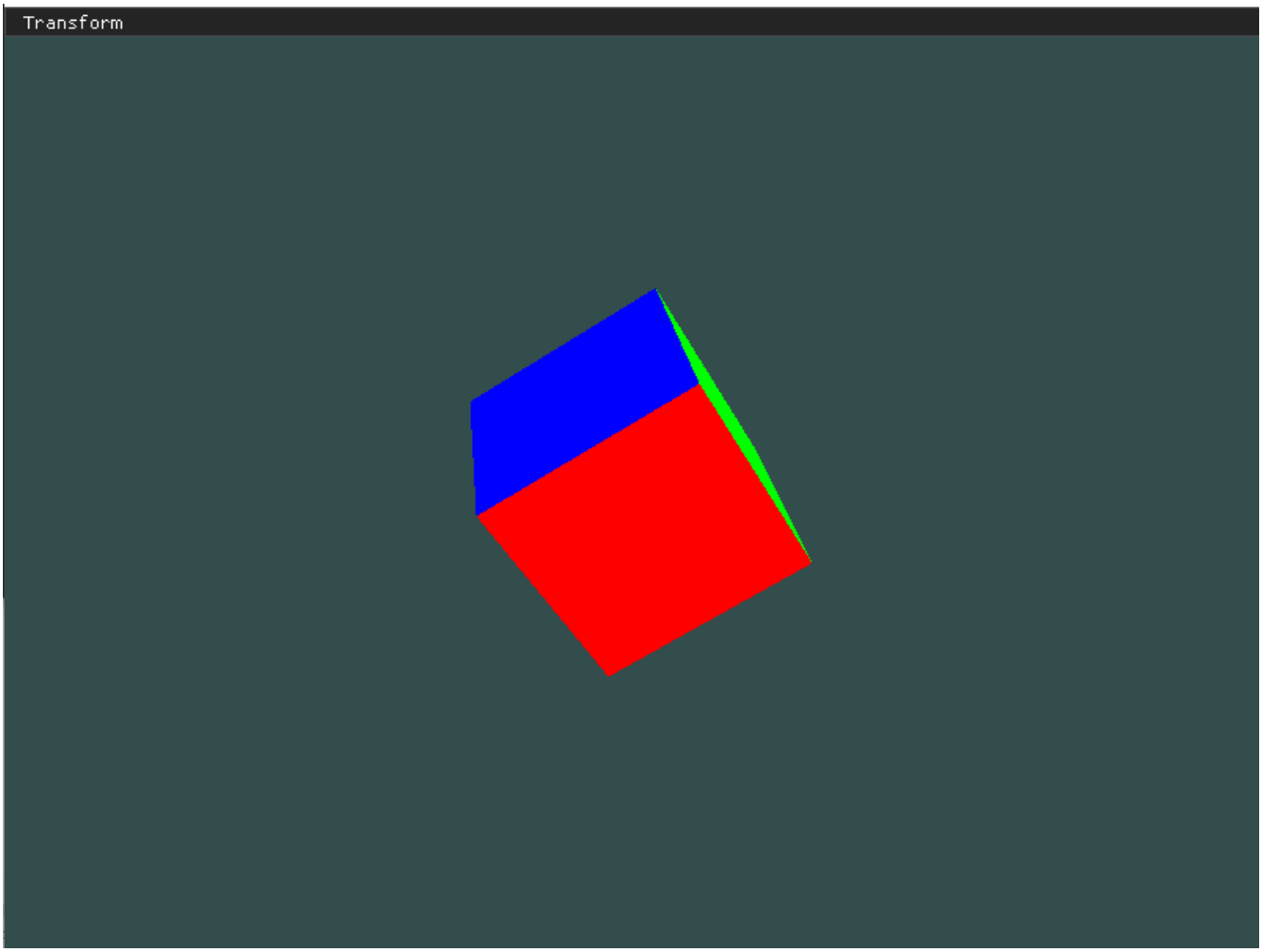
```
# vertex shader
# version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 ourColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    ourColor = aColor;
}
```

效果



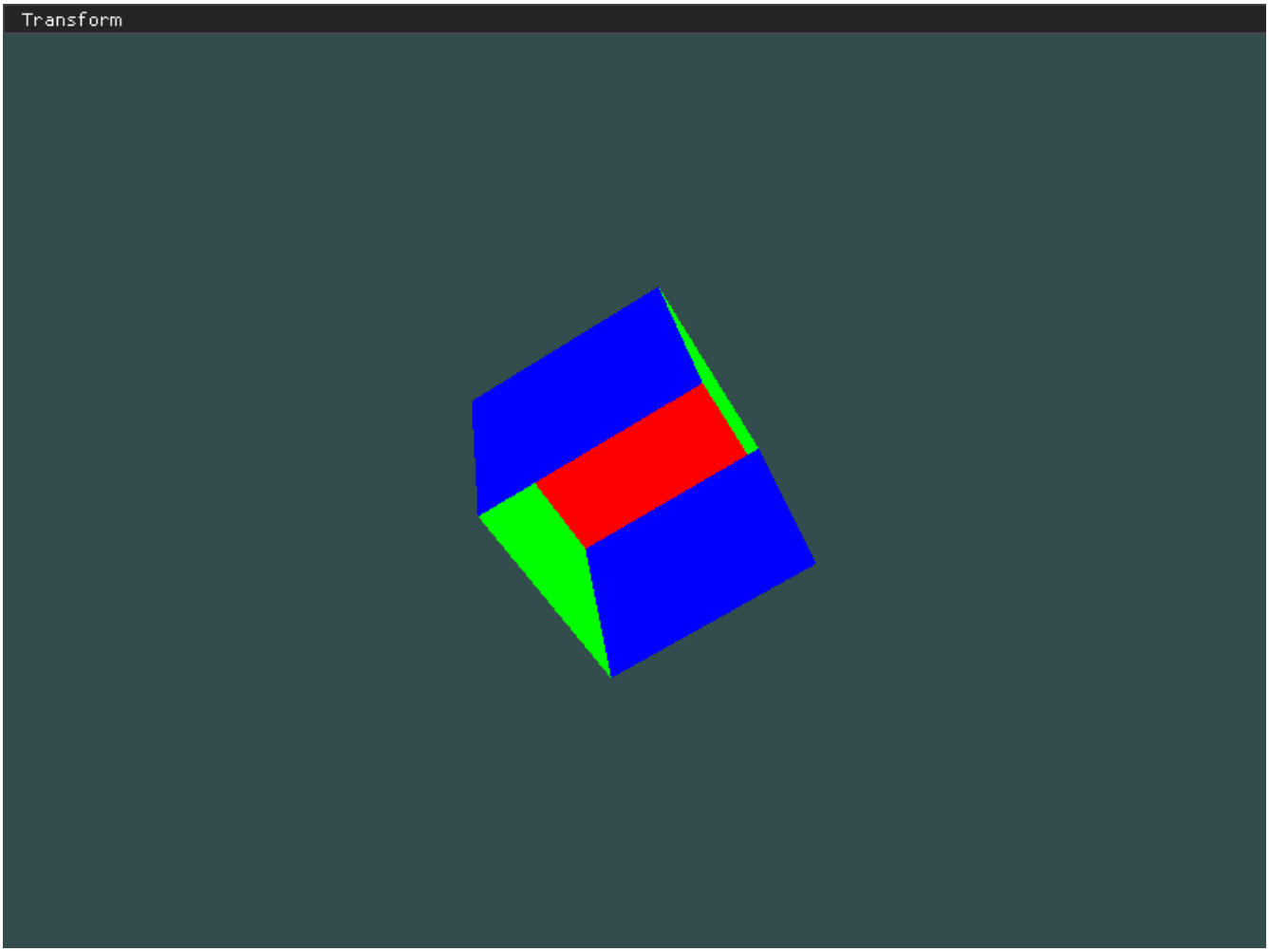
深度测试

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)。GLFW会自动为你生成这样一个缓冲（就像它也有一个颜色缓冲来存储输出图像的颜色）。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。

深度测试默认是关闭的，我们可以通过下面这个函数调用来开启：

```
glEnable(GL_DEPTH_TEST);
```

深度测试未开启的情况下，立方体的效果如下：



可以看到在绘制过程中，即使原来的位置上已经有东西了，OpenGL还是会继续用新的东西覆盖原来的像素。在开启深度测试后，OpenGL会将片段的深度值和z缓冲做比较，决定是否覆盖。

变换

平移

比起上述单纯地生成一个经过简单旋转的立方体，如果我们想要让立方体在x轴或y轴上移动，我们只需要将 `Model` 矩阵再左乘一个位移矩阵，使用 `glm:translate` 可以达到目的：

```
// Translation
// -----------
if (transform_type == 1) {
    model = glm::translate(model, glm::vec3((float)sin(glfwGetTime()) * 4, 0.0f, 0.0f));
}
```

这里我们让立方体沿着x轴来回移动，通过一个 `sin` 函数，我们可以把位移随着时间变化限制在 `[-4, 4]` 之间。

具体效果参见附带的GIF演示图

旋转

在实现立方体的过程中我们已经使用过 `glm:rotate` 这个函数了，它可以将 `model` 矩阵左乘一个旋转矩阵：

```
// Rotation
// --------
else if (transform_type == 2) {
    model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(1.0f, 0.0f, 1.0f));
}
```

具体效果参见附带的GIF演示图

缩放

缩放也很简单，使用 `glm:scale` 函数即可让 `Model` 矩阵左乘一个缩放矩阵：

```
// Scaling
// -------
else if (transform_type == 3) {
    float scale = (float)sin(glfwGetTime()) / 2 + 1;
    model = glm::scale(model, glm::vec3(scale, scale, scale));
}
```

这里同样使用 `sin` 函数来让立方体周期性的放大缩小，同时为了让缩放系数限定在一个范围内，我们可以对函数做一个简单的变换：`sin(t) / 2 + 1`

具体效果参见附带的GIF演示图

组合各种变换

这里实现的是一个立方体围绕着另一个立方体转动，同时两个立方体本身也会自转，实现效果也请参见GIF演示图。

环绕立方体

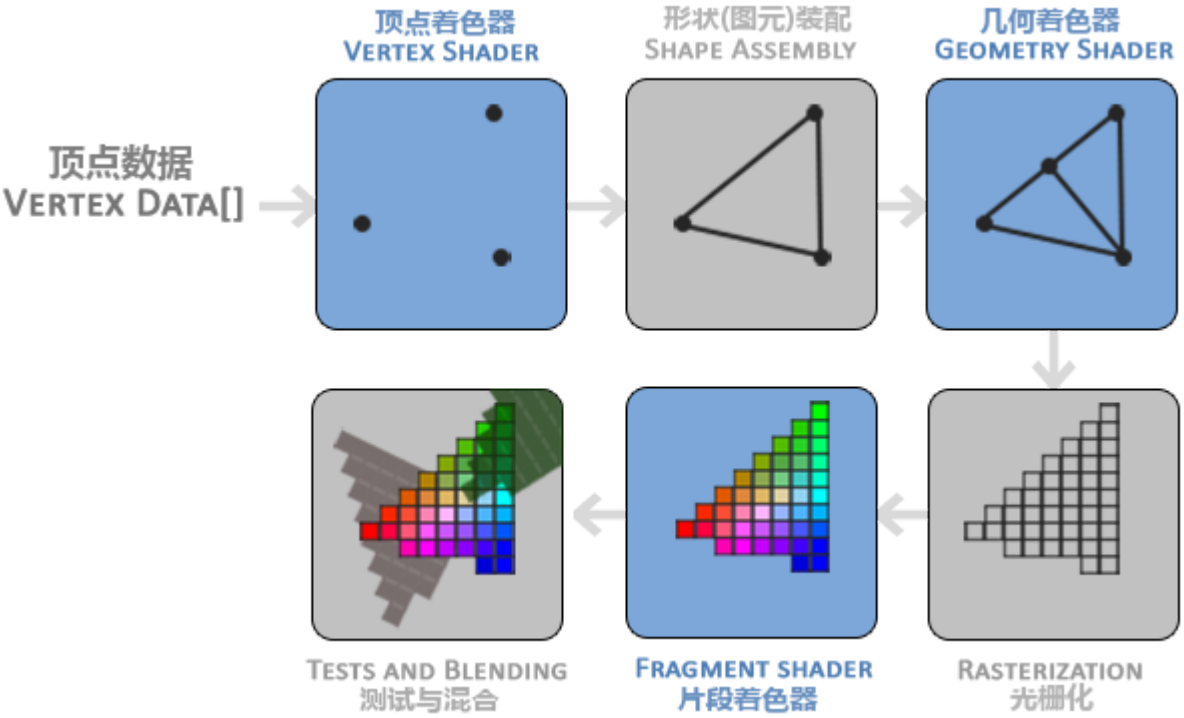
实现的思路其实很简单，首先立方体要自转，我们可以让 `Model` 先左乘一个随时间变换的旋转矩阵；其次为了让其能围绕中心立方体转动，只需要做两个操作：位移和绕y轴旋转，实现代码如下：

```
// Combination
// -----------
else if (transform_type == 4) {
    // Zoom out
    float surrounding_object_scale = 0.5;
    view = glm::translate(view, glm::vec3(0.0f, 0.0f, -20.0f));
    model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::translate(model, glm::vec3(0.0f, 0.0f, 15.0f));
    model = glm::rotate(model, (float)glfwGetTime() * 5, glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::scale(model, glm::vec3(surrounding_object_scale, surrounding_object_scale, surrounding_object_scale));

    // centering object
    float centering_object_scale = 1.2;
    glm::mat4 model2 = glm::mat4(1.0f);
    model2 = glm::rotate(model2, (float)glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));
    model2 = glm::rotate(model2, glm::radians(45.0f), glm::vec3(1.0f, 0.0f, 1.0f));
    model2 = glm::scale(model2, glm::vec3(centering_object_scale, centering_object_scale, centering_object_scale));
    shader.setMat4("model", model2);
    // render centering object
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

对渲染管线的理解

> 引用自 LearnOpenGL:



1. 首先，我们以数组的形式传递3个3D坐标作为图形渲染管线的输入，用来表示一个三角形，这个数组叫做顶点数据(Vertex Data)；
2. 图形渲染管线的第一个部分是**顶点着色器(Vertex Shader)**，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（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。
3. 图元装配(Primitive Assembly)阶段将顶点着色器输出的所有顶点作为输入（如果是GL_POINTS，那么就是一个顶点），并所有的点装配成指定图元的形状；本节例子中是一个三角形。
4. 图元装配阶段的输出会传递给**几何着色器(Geometry Shader)**。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（或是其它的）图元来生成其他形状。例子中，它生成了另一个三角形。
5. 几何着色器的输出会被传入光栅化阶段(Rasterization Stage)，这里它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(Fragment Shader)**使用的片段(Fragment)。在片段着色器运行之前会执行裁切(Clipping)。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. 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。

在我们做过的这几次作业中，我们已经对 `Vertex Shader` 和 `Fragment Shader` 有过一些实践，例如在这次作业中，我们需要对顶点进行变换，而这个操作在 `Vertex Shader` 中进行，也就是我上面所写的 `Vertex Shader Code` ，其将顶点坐标依次左乘 `Model` `View` `Projection` 三个矩阵，将局部坐标转换到**裁剪空间**。而 `Fragment Shader` 的任务主要是计算像素的颜色，目前为止我们只是简单地传入顶点地颜色值，并没有更多复杂的操作。可编程管线让我们更灵活地实现目标。