Homework 4 - Transformation

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。

- 对于正方体的每个面，由两个三角形拼接而成。而正方体的边长为 4，中心位置为（0,0,0），因此，顶点各个轴向偏离中心的长度可设置为 2.0f，同时为各个顶点加上颜色属性。因此，正方体的部分顶点坐标如下：

```
// 正方体六个面顶点处理，每个面由两个三角形合成
float vertices[] = {
    // 正面
    -2.0f, -2.0f,  2.0f,  0.0f, 0.0f, 1.0f,
    2.0f, -2.0f,  2.0f,   0.0f, 0.0f, 1.0f,
    2.0f,  2.0f,  2.0f,   0.0f, 0.0f, 1.0f,
    2.0f,  2.0f,  2.0f,   0.0f, 0.0f, 1.0f,
    -2.0f, 2.0f, 2.0f,  0.0f, 0.0f, 1.0f,
    -2.0f, -2.0f, 2.0f,  0.0f, 0.0f, 1.0f,
    // 上面
    -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,
    // 左面
```

- 然后绑定 VAO 和 VBO。
- 默认来说，OpenGL 能观察到的窗口范围为(1.0f，1.0f)，而现在正方体的长度为 4，为了能观察到整个正方体，需要调整摄像机的位置。在这里，使用 glm 库的 lookAt 函数来设置，实现如下：

```
glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 5.0f, 20.0f),
    glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

- 然后，在顶点着色器加上 3 个 uniform 的 mat4 分别表示 model，view 和 projection 矩阵。

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 vColor;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

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

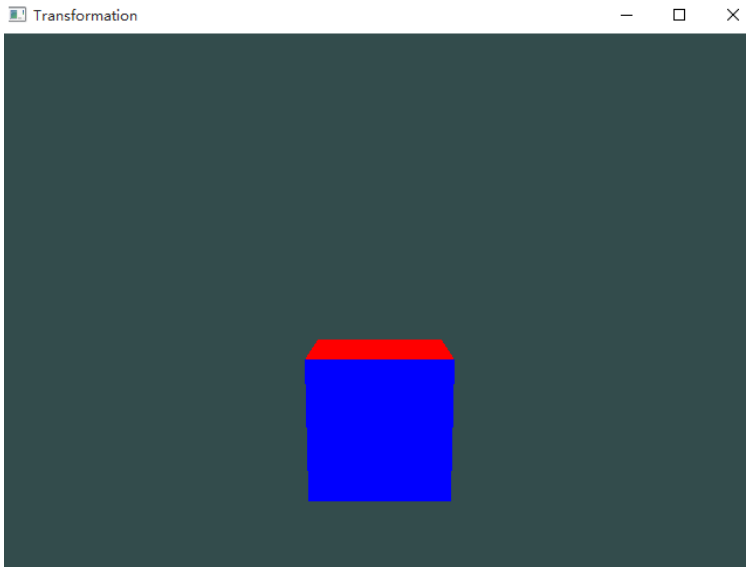
- 使用 glEnable(GL_DEPTH_TEST)开启深度测试，使用 glDisable(GL_DEPTH_TEST)关闭深度测试。并在循环中使用 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)，如下：

```
// 是否激活深度测试
glEnable(GL_DEPTH_TEST);
//glDisable(GL_DEPTH_TEST);
```

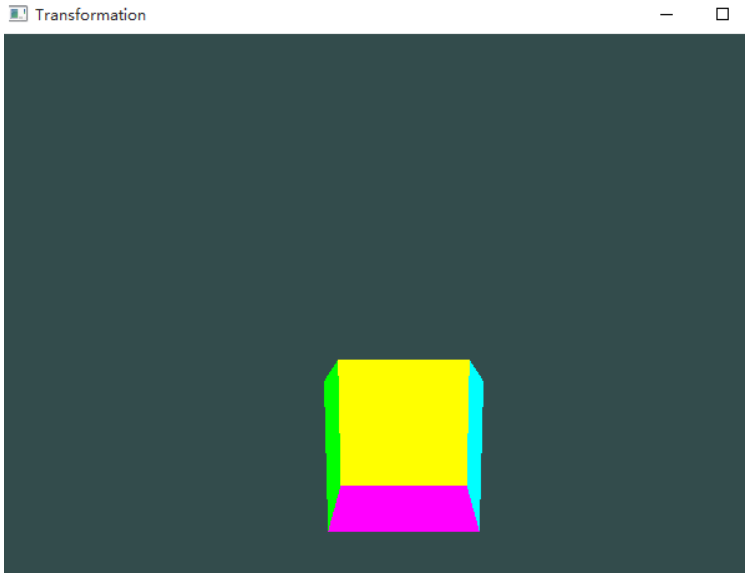
➢ 此外，还需要设置 view 和 projection，如下：

```
glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 5.0f, 20.0f),
    glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 projection = glm::perspective(45.0f, (float)WIN_WIDTH / (float)WIN_HEIGHT, 1.0f, 100.0f);
my_shader.use();
my_shader.setMat4("view", glm::value_ptr(view));
my_shader.setMat4("projection", glm::value_ptr(projection));
```

实现结果如下：



开启深度测试



关闭深度测试

解释：

开启深度测试的时候，可以看出是一个正方体，而关闭深度测试后，看起来像是正方体的前面和上面都被掀去了。即原本应该遮挡住后面的那些面，反而是被覆盖了。出现这种现象，是因为 OpenGL 绘制立方体时，是一个三角形一个三角形地绘制的，所以，未开启深度测试时，即便原本那个像素有东西，也会被新绘制的三角形覆盖。而开启了深度测试后，则会根据深度来选择保留的像素。

2. 平移(Translation)：使画好的 cube 沿着水平或垂直方向来回移动。

- 在每一帧中，使用时间来控制平移矩阵。其中，获取时间的函数为 glfwGetTime()，为了使 cube 来回移动，需要使用一个周期函数，以时间为周期，控制移动周期，这里采用 sin 函数。然后，使用 glm 的 translate 函数来生成移动矩阵。实现效果请见演示视频.mp4。

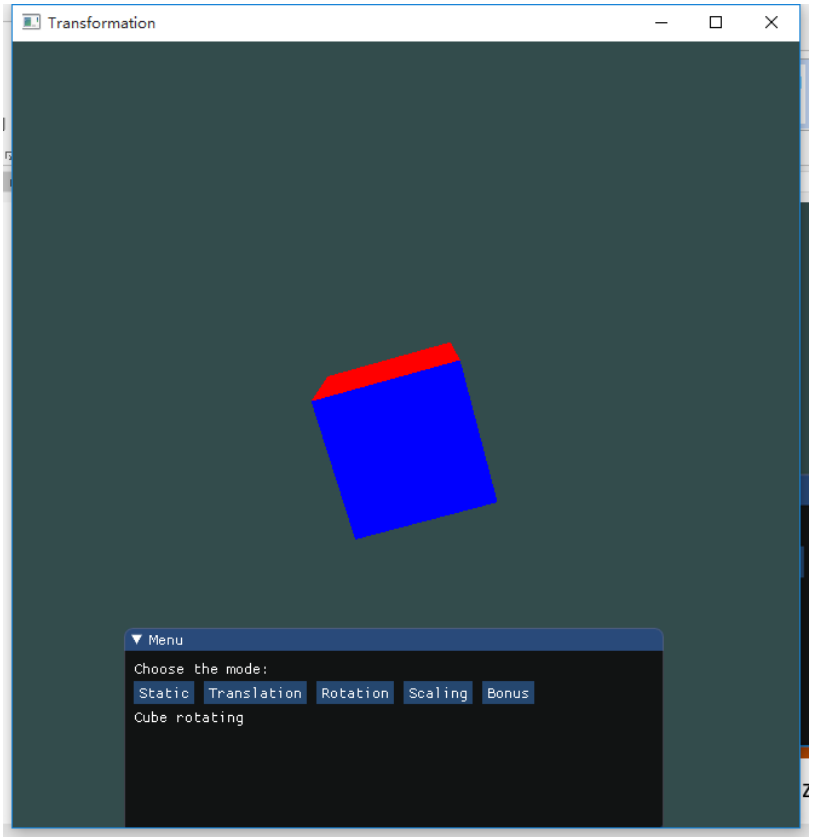
```
model = glm::translate(model, (float)sin(glfwGetTime()) * glm::vec3(2.0f, 0.0f, 0.0f));
my_shader.setMat4("model", glm::value_ptr(model));
```

3. 旋转(Rotation)：使画好的 cube 沿着 XoZ 平面的 x=z 轴持续旋转。

- 在每一帧中，使用时间来控制旋转角度。获取时间的函数为 glfwGetTime()，使用 glm 的 rotate 函数来生成旋转矩阵。

```
model = glm::rotate(model, (float)glfwGetTime() * 5.0f, glm::vec3(0.0f, 0.0f, 1.0f));
my_shader.setMat4("model", glm::value_ptr(model));
```

- 个别帧如下，具体实现效果请见演示视频.mp4。

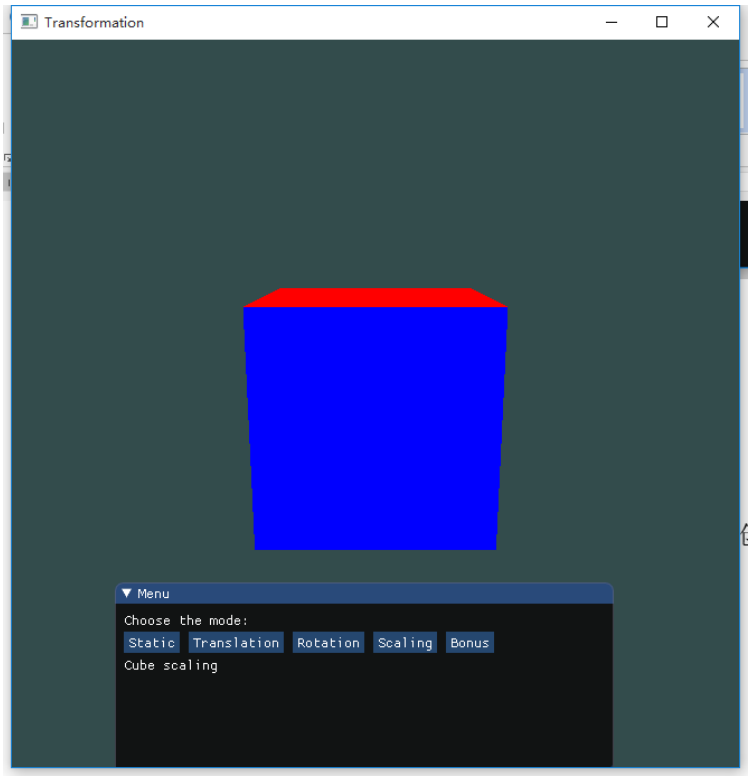


4. 放缩(Scaling)：使画好的 cube 持续放大缩小。

- 在每一帧中，使用时间来控制缩放尺度。其中，获取时间的函数为 glfwGetTime()，为了使 cube 持续放大缩小，需要使用一个周期函数，以时间为周期，控制缩小或放大周期，这里采用 sin 函数。然后，使用 glm 的 scale 函数来生成尺度变换矩阵。

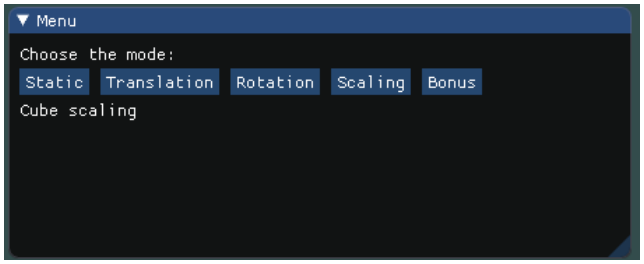
```
model = glm::scale(model, (float)abs(sin(glfwGetTime()))*glm::vec3(2.0f, 2.0f, 2.0f));
my_shader.setMat4("model", glm::value_ptr(model));
```

➢ 个别帧如下，具体实现效果请见演示视频.mp4。



5. 在 GUI 里添加菜单栏，可以选择各种变换。

➢ 实现的菜单栏效果如下：

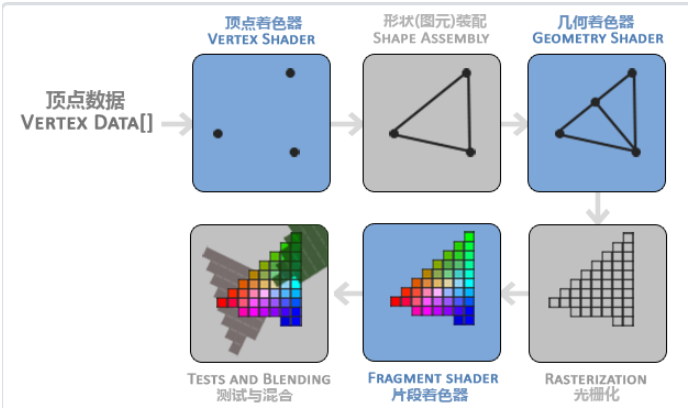


➢ 使用 ImGui:: Text 来显示文本内容，使用 ImGui:: Button 来改变模式，从容切换各种变换。为了使各个 button 在同一行，使用在每个定义的 button 后面加上 ImGui:: Sameline 函数。如下：

```
//设置菜单样式
{
    ImGui::Begin("Menu");
    ImGui::Text("Choose the mode:");
    if (ImGui::Button("Static"))
        mode = 0;
    ImGui::SameLine();
    if (ImGui::Button("Translation"))
        mode = 1;
    ImGui::SameLine();
    if (ImGui::Button("Rotation"))
        mode = 2;
    ImGui::SameLine();
    if (ImGui::Button("Scaling"))
        mode = 3;
    ImGui::SameLine();
    if (ImGui::Button("Bonus"))
        mode = 4;
```

6. 结合 Shader 谈谈对渲染管线的理解

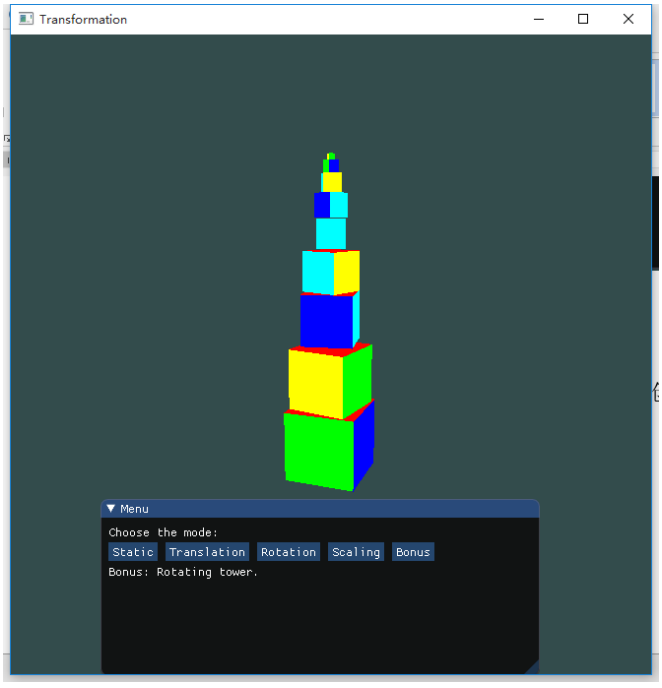
Shader（着色器）是运行在 CPU 上的较为短小的程序片段。这些程序片段为图形渲染管线的某个特定部分而运行，用于告诉图形硬件如何计算和输出图像。从基本意义上来说，着色器只是一种把输入转化为输出的程序。着色器也是一种非常独立的程序，因为它们之间不能相互通信；它们之间唯一的沟通只有通过输入和输出。概括而言，Shader 是可编程图形管线的算法片段，分为 Vertex Shader 和 Fragment Shader。渲染管线是显示芯片内部处理图形信号相互独立的并行处理单元，也称为渲染流水线。渲染管线由几个阶段组成，每个阶段都有输入和输出，前一阶段的输出是后一阶段的输入。简单而言，渲染管线就是对数据按一定的顺序进行各种特定的处理，最后得出可视化的图像。如下：



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。

- 在这里，实现一个类似于塔的会旋转的“塔”，由 9 个正方体组成，正方体的六面颜色相同。从下往上，正方体的大小越来越小，旋转速度越来越快，同时，相邻两个正方体的旋转方向相反。个别帧如下，具体实现效果请见演示视频.mp4。



- 对于这 9 个正方体，可以通过对 Basic 中的正方体进行平移、缩放来得到。可以通过数组定义这些正方体的平移距离、缩放尺度以及旋转速度。
- 以(0,0,0)为中心，各个正方体的平移尺度如下：

```
// world space positions of our cubes
glm::vec3 cube_positions[] = {
    glm::vec3(0.0f, -5.0f, 0.0f),
    glm::vec3(0.0f, -1.5f, 0.0f),
    glm::vec3(0.0f, 3.0f, 0.0f),
    glm::vec3(0.0f, 8.5f, 0.0f),
    glm::vec3(0.0f, 17.0f, 0.0f),
    glm::vec3(0.0f, 26.0f, 0.0f),
    glm::vec3(0.0f, 39.5f, 0.0f),
    glm::vec3(0.0f, 64.5f, 0.0f),
    glm::vec3(0.0f, 135.0f, 0.0f)
};
```

- 各个正方体的缩放尺度如下：

```
//正方形的缩放处理
glm::vec3 cube_scale[] = {
    glm::vec3(0.65f, 0.65f, 0.65f),
    glm::vec3(0.55f, 0.55f, 0.55f),
    glm::vec3(0.45f, 0.45f, 0.45f),
    glm::vec3(0.35f, 0.35f, 0.35f),
    glm::vec3(0.25f, 0.25f, 0.25f),
    glm::vec3(0.2f, 0.2f, 0.2f),
    glm::vec3(0.15f, 0.15f, 0.15f),
    glm::vec3(0.1f, 0.1f, 0.1f),
    glm::vec3(0.05f, 0.05f, 0.05f)
};
```

- 对于旋转速度，使用 glfwGetTime()函数乘以一个系数来控制。其中，当这个系数为负数时，旋转方向与系数为整数时相反。在此，对 9 个正方体旋转时的系数定义如下：

```
//用于控制正方形旋转的速率
float rotate_rate[] = {
    -0.5f, 1.5f, -2.5f, 3.5f, -4.5f, 5.5f, -6.5f, 7.5f, -8.5f
};
```

- 最后，在 while(!glfwWindowShouldClose(window))循环中，使用一个 for 循环来渲染这 9 个正方体。如下：

```
glBindVertexArray(VAO);

for (unsigned int i = 0; i < 9; i++)
{
    // calculate the model matrix for each object and pass it to shader before drawing
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::scale(model, cube_scale[i]);
    model = glm::rotate(model, (float)glfwGetTime()*rotate_rate[i], glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::translate(model, cube_positions[i]);

    my_shader.setMat4("model", glm::value_ptr(model));

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```