Homework 4

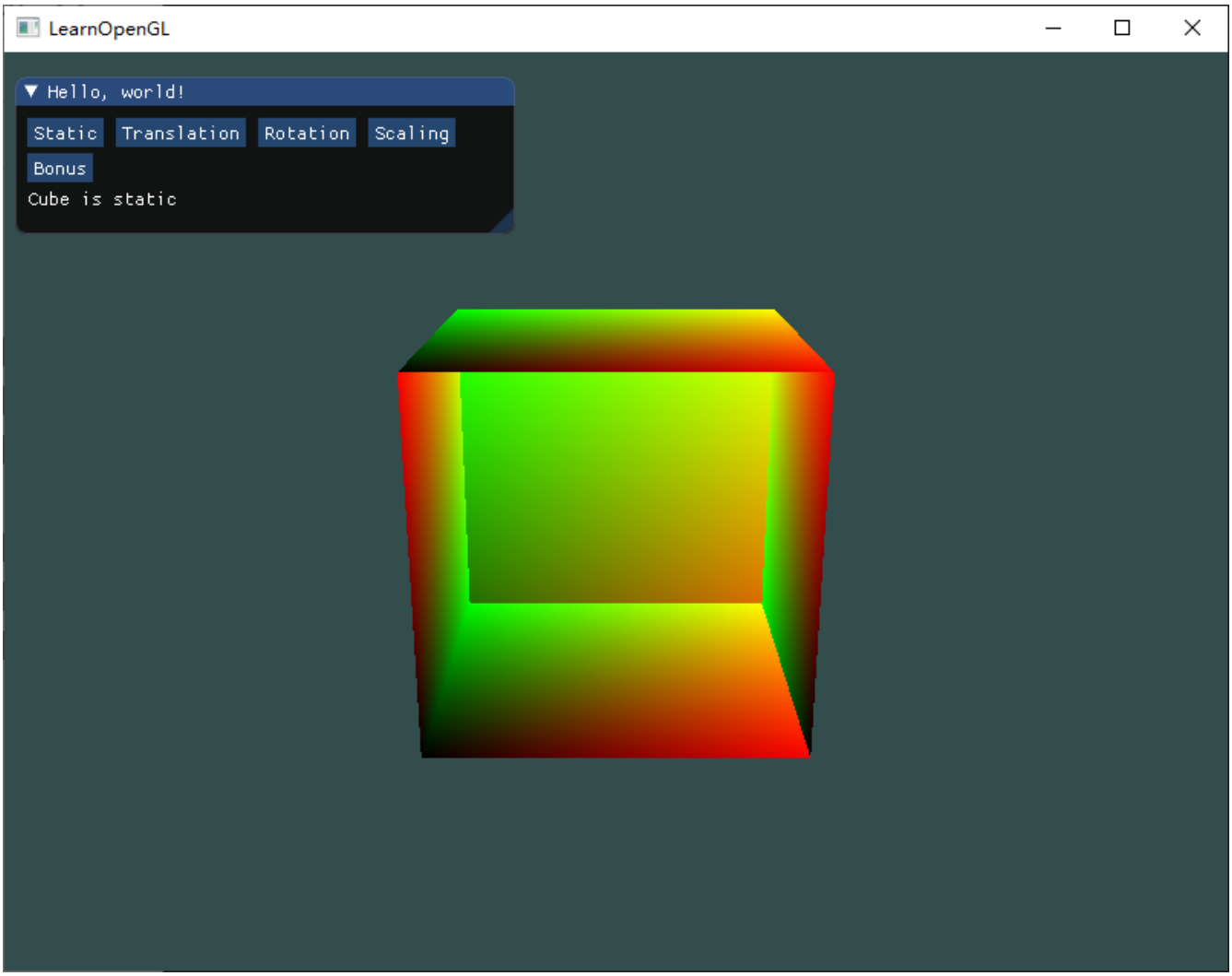
16340282 袁之浩

Basic

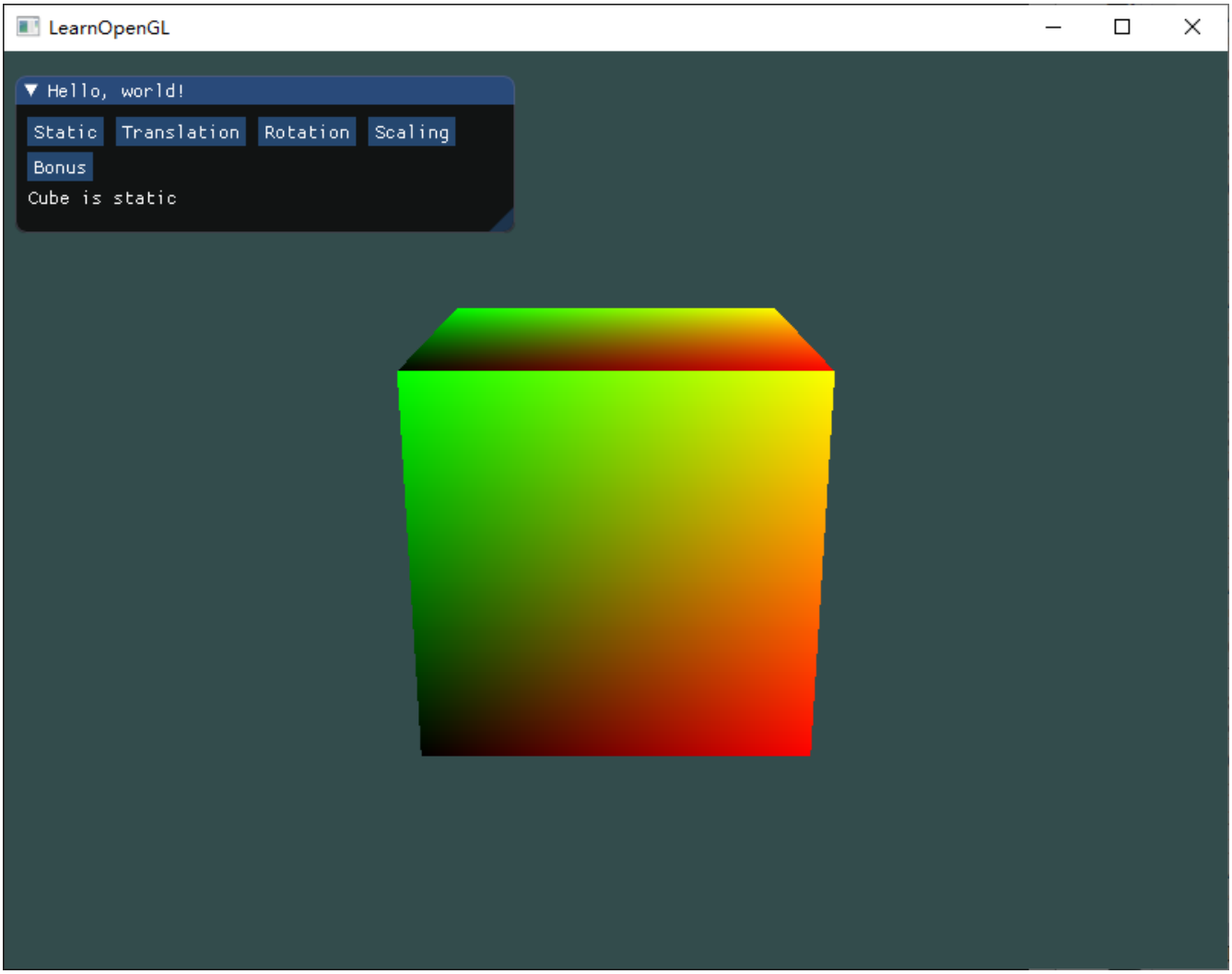
1. 画一个立方体(cube)：边长为4， 中心位置为(0, 0, 0)

实验截图

关闭深度测试



开启深度测试



算法实现

要渲染一个立方体，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）。所以首先在vertices数组中定义每个顶点的位置和颜色。

```
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,
     ... ...
 }
```

然后我们使用glDrawArrays来绘制立方体，但这一次总共有36个顶点。

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

定义观察矩阵和投影矩阵

```
glm::mat4 view;
// 注意，我们将矩阵向我们要进行移动场景的反方向移动。
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), screenWidth / screenHeight, 0.1f,
100.0f);
```

顶点着色器,声明变换矩阵model，观察矩阵view，投影矩阵projection，再与顶点坐标相乘得到最终位置。

```
#version 330 core

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

out vec4 ourColor;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

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

因为变换矩阵会经常变动，所以通常再每次的渲染迭代中，将矩阵传入着色器。

```
// pass them to the shaders (3 different ways)
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]);
ourShader.setMat4("projection", projection);
```

现象解释

在没有开始深度测试时，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。这是因为OpenGL是一个三角形一个三角形地来绘制立方体的，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。

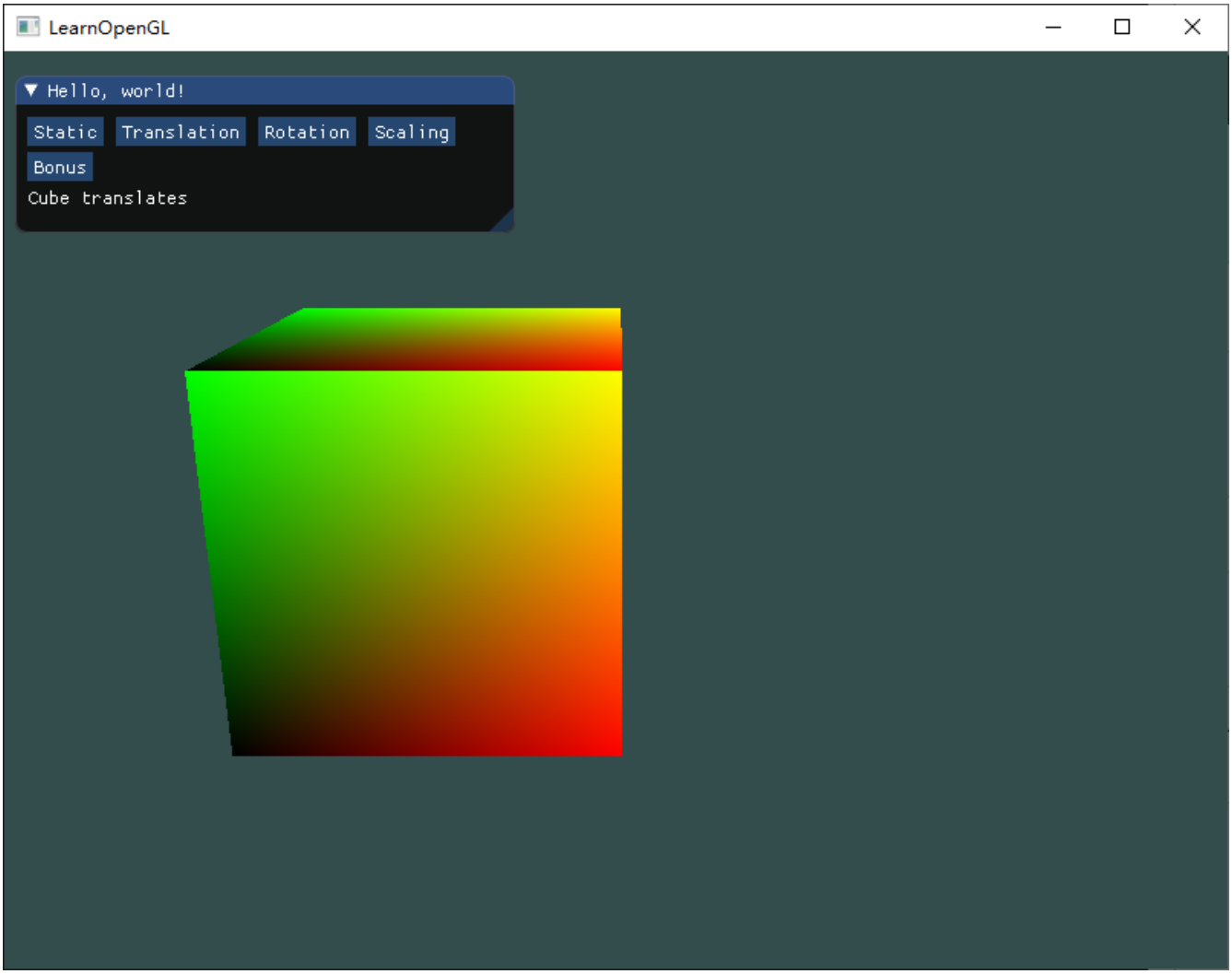
OpenGL存储它的所有深度信息于一个Z缓冲(Z-buffer)中，也被称为深度缓冲(Depth Buffer)。GLFW会自动为你生成这样一个缓冲（就像它也有一个颜色缓冲来存储输出图像的颜色）。深度值存储在每个片段里面（作为片段的z值），当片段想要输出它的颜色时，OpenGL会将它的深度值和z缓冲进行比较，如果当前的片段在其它片段之后，它将会被丢弃，否则将会覆盖。这个过程称为深度测试(Depth Testing)，它是由OpenGL自动完成的。

开启深度测试，并在每次渲染迭代之前清除深度缓冲，观察到立方体渲染正常。

```
glEnable(GL_DEPTH_TEST);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

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

实验截图



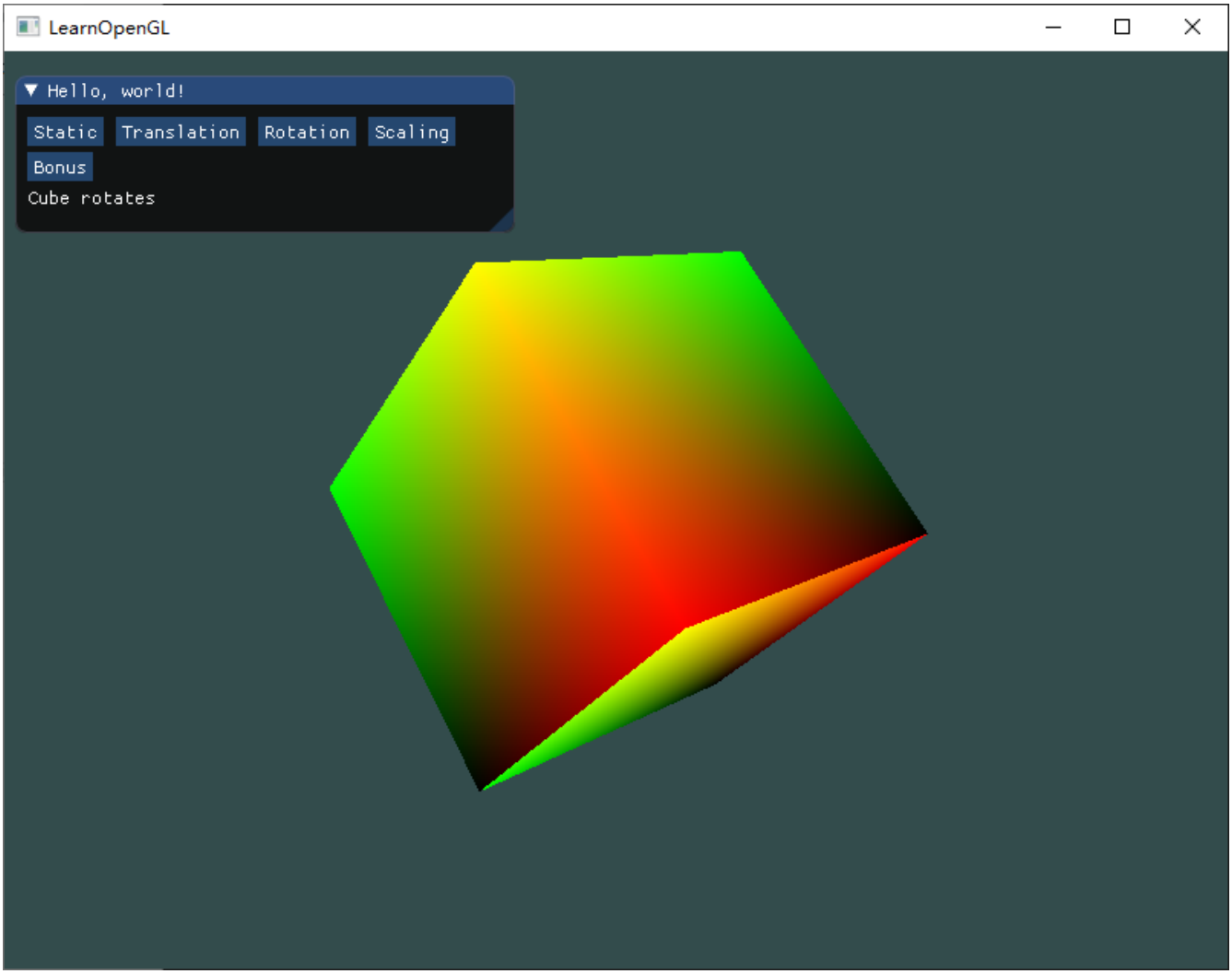
算法实现

使用glm::translate函数来改变model矩阵，实现水平平移，使用sin和glfGetTime来实现来回移动。

```
model = glm::translate(model, (float)sin(glfwGetTime()) * glm::vec3(0.5f, 0.0f, 0.0f));
```

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

实验截图



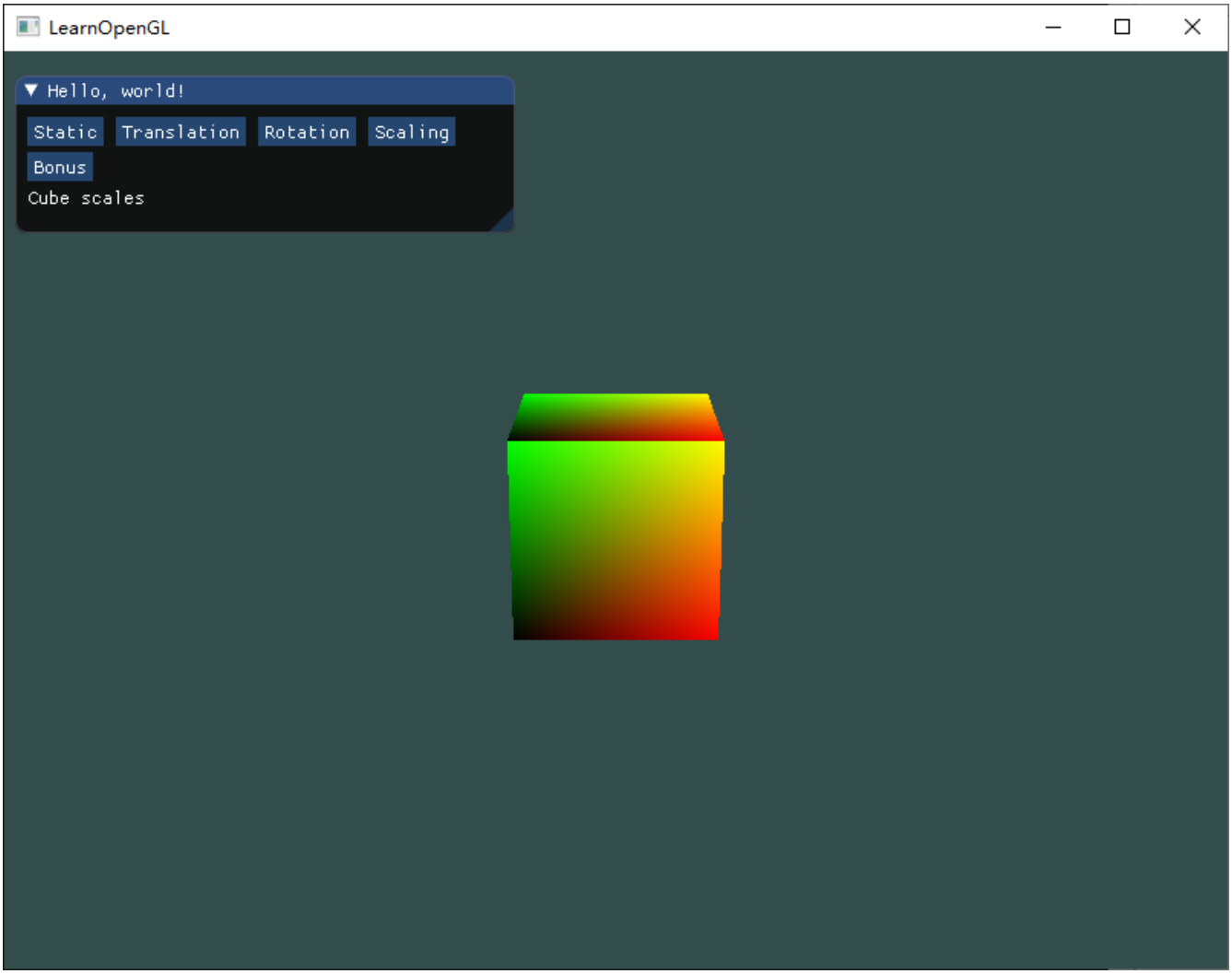
算法实现

使用glm::rotate函数来改变model矩阵，实现旋转

```
model = glm::rotate(model, (float)glfwGetTime() * 2.0f, glm::vec3(1.0f, 0.0f, 1.0f));
```

4. 放缩(Scaling)：使画好的cube持续放大缩小

实验截图



算法实现

使用glm::scale函数来改变model矩阵，使用abs(sin(glfwGetTime()))来实现周期变化

```
model = glm::scale(model, (float)abs(sin(glfwGetTime())) * glm::vec3(2.0f, 2.0f, 2.0f));
```

5. 在GUI里添加菜单栏，可以选择各种变换

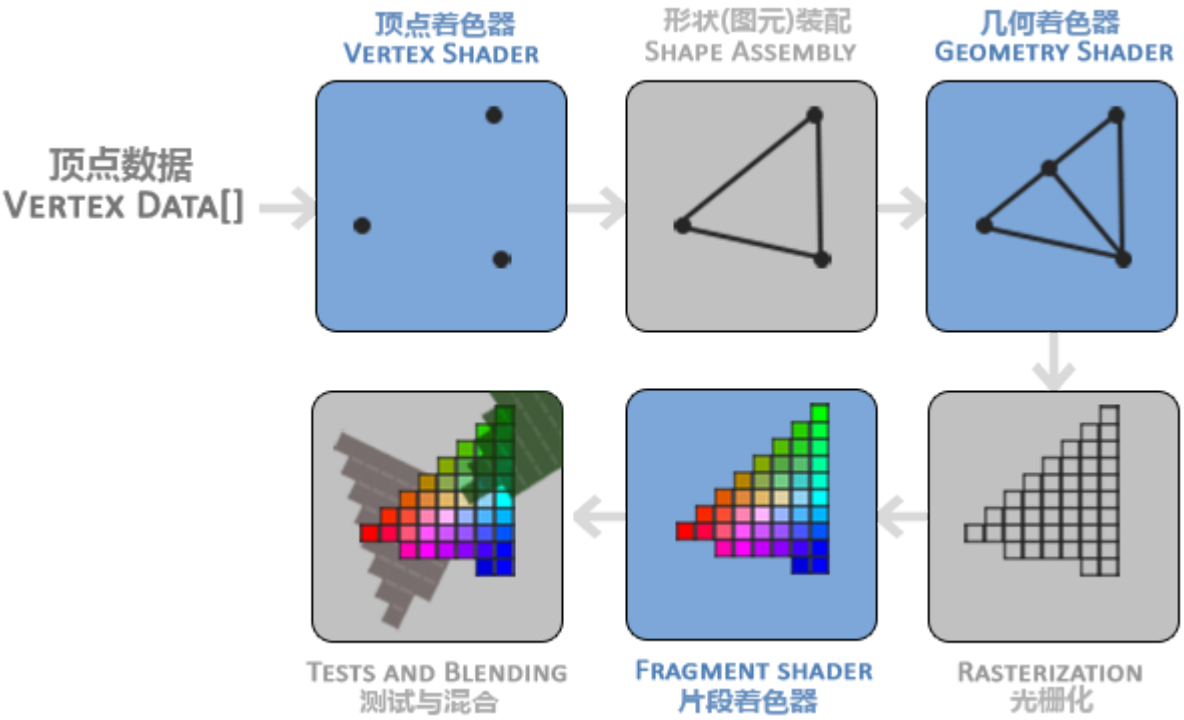
算法实现

使用ImGui::Button来选择和切换各种变换

```
// 模式选择框
            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;
        if (ImGui::Button("Bonus"))
            mode = 4;
```

6. 结合Shader谈谈对渲染管线的理解



图形渲染管线接受一组3D坐标，然后把它们转变为屏幕上的有色2D像素输出。图形渲染管线可以被划分为几个阶段，每个阶段将会把前一个阶段的输出作为输入。

当今大多数显卡都有成千上万的小处理核心，它们在GPU上为每一个（渲染管线）阶段运行各自的小程序，从而在图形渲染管线中快速处理你的数据。这些小程序叫做**着色器(Shader)**。

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（局部坐标）转为另一种3D坐标（裁剪坐标），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(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。

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。

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。

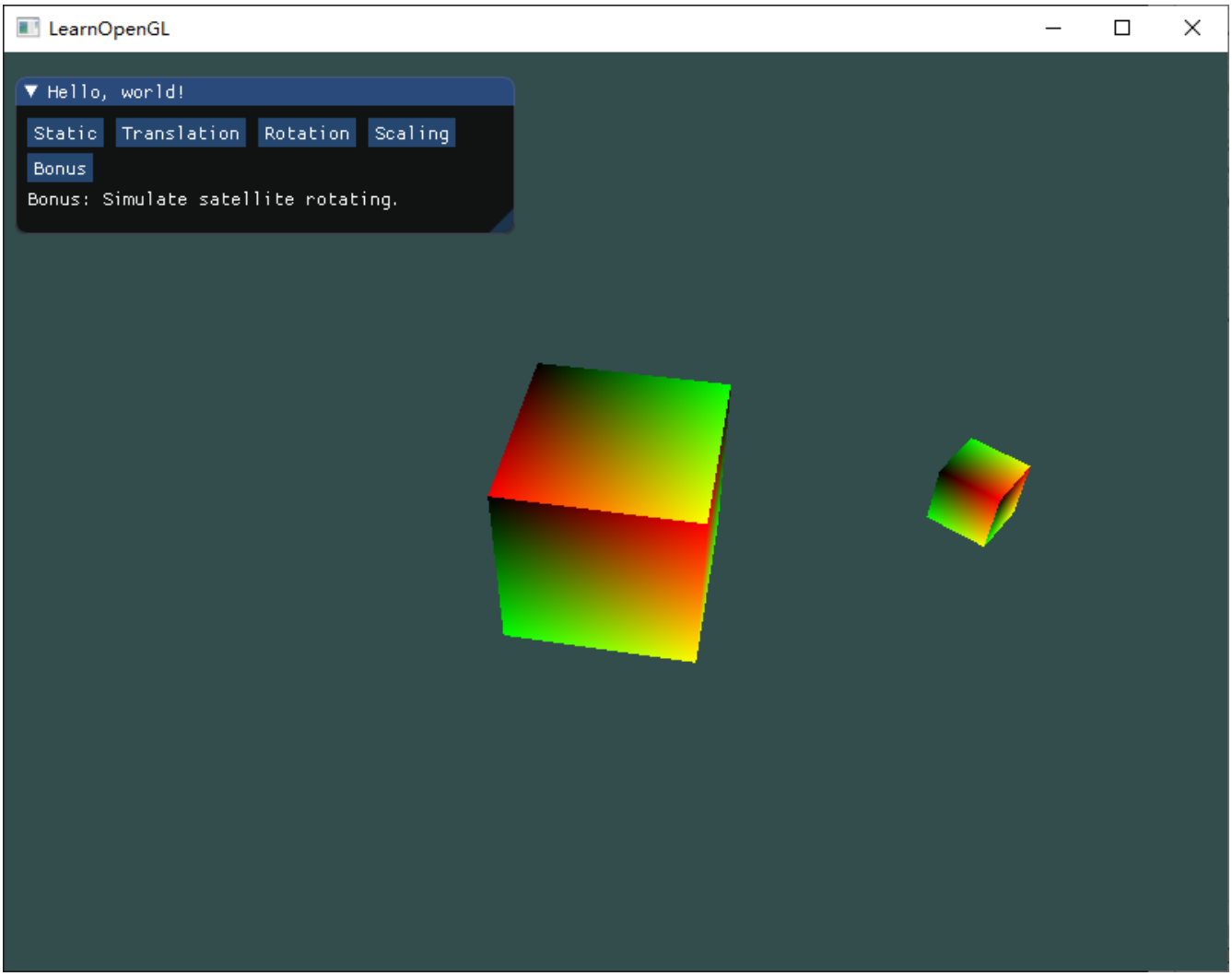
片段着色器的主要目的是计算一个像素的最终颜色，这也是所有OpenGL高级效果产生的地方。通常，片段着色器包含3D场景的数据（比如光照、阴影、光的颜色等等），这些数据可以被用来计算最终像素的颜色。

在所有对应颜色值确定以后，最终的对象将会被传到最后一个阶段，叫做**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)**。所以，即使在片段着色器中计算出来了一个像素输出的颜色，在渲染多个三角形的时候最后的像素颜色也可能完全不同。

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。

实验截图



算法实现

申明一个vector用于保存每个cube的变换矩阵，用一个cube当作太阳，用另一个当作地球，再改变观察矩阵的位置，使用glm::lookAt()，三个参数分别是摄像机观察位置eye，朝向center，Y轴正方向up。

```
vector<glm::mat4> myModel;
myModel.assign(2, glm::mat4(1.0f));
// sum
myModel[0] = glm::scale(myModel[0], glm::vec3(1.5, 1.5, 1.5));
myModel[0] = glm::rotate(myModel[0], (float)glfwGetTime() * 1.0f, glm::vec3(0.0f, 0.0f,
1.0f));

// earth
myModel[1] = glm::scale(myModel[1], glm::vec3(0.5, 0.5, 0.5));
myModel[1] = glm::rotate(myModel[1], (float)glfwGetTime() * 4.0f, glm::vec3(0.0f, 0.0f,
1.0f));
myModel[1] = glm::translate(myModel[1], glm::vec3(4, 4, 0.0));

view = glm::lookAt(glm::vec3(0.0f, -6.0f, 6.0f),glm::vec3(0.0f, 0.0f, 0.0f),
                glm::vec3(0.0f, 1.0f, 0.0f));
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]);
```