练 习 题 报 告

课程名称 计算机图形学

项目名称 三维模型的平移、缩放和旋转

学　　院 计算机与软件学院

专　　业 计算机科学与技术

指导教师 周虹

报 告 人 吴嘉楷 学号 2022150168

一、 练习目的

1. 掌握三维模型顶点与三角面片之间关系。
2. 了解和掌握三维模型的基本变换操作。
3. 掌握在着色器中使用变换矩阵。

二. 练习完成过程及主要代码说明

1. 在 TriMesh.cpp 中完善 generateCube 函数

函数源码：

```
void TriMesh::generateCube() {
    // 创建顶点前要先把那些 vector 清空
    cleanData();
    // @TODO: Task1-修改此函数，存储立方体的各个面信息
    //  vertex_positions 和 vertex_colors 先保存每个顶点的数据
    for (int i = 0; i < 8; i++) {
        vertex_positions.push_back(cube_vertices[i]);
        vertex_colors.push_back(basic_colors[i]);
    }
    //  faces 再记录每个面片上顶点的下标
    faces = {
        vec3i(0, 1, 2), vec3i(1, 3, 2),
        vec3i(4, 6, 5), vec3i(5, 6, 7),
        vec3i(0, 2, 4), vec3i(2, 6, 4),
        vec3i(1, 5, 3), vec3i(3, 5, 7),
        vec3i(0, 4, 1), vec3i(1, 4, 5),
        vec3i(2, 3, 6), vec3i(3, 7, 6)
    };
    // 存储每个面片的点和颜色信息
    storeFacesPoints();
}
```

代码说明：

generateCube 函数生成一个立方体的顶点、颜色和面片数据，以便后续渲染。首先，通过调用 cleanData() 清空之前的顶点、颜色和面片数据，确保生成的新立方体数据不受先前数据的影响。然后，使用一个循环将立方体的顶点位置和颜色数据分别存储到 vertex_positions 和 vertex_colors 中，这些数据来源于 cube_vertices 和 basic_colors 数组。每个顶点位置存入 vertex_positions，对应的颜色数据存入 vertex_colors，最终形成立方体的基本顶点和颜色信息。

接下来，faces 使用初始化列表存储立方体的面片结构，每个面片由三个顶点索引构成的 vec3i 组成，这样的三角形面片定义了立方体的六个面。共 12 个三角形，确保立方体的所有面都完整定义。最后调用 storeFacesPoints() 函数，

将这些面片中的顶点位置和颜色数据组织好，方便后续渲染时传输给 GPU，完成立方体的生成和数据准备。

2. 在 TriMesh.cpp 中完善 storeFacesPoints 函数

函数源码：

```
void TriMesh::storeFacesPoints() {
    // @TODO: Task-2 修改此函数在 points 和 colors 容器中存储每个三角面片的各点和颜色信息
    // 根据每个三角面片的顶点下标存储要传入 GPU 的数据
    for (const auto& face : faces) {
        points.push_back(vertex_positions[face.x]);
        points.push_back(vertex_positions[face.y]);
        points.push_back(vertex_positions[face.z]);
        colors.push_back(vertex_colors[face.x]);
        colors.push_back(vertex_colors[face.y]);
        colors.push_back(vertex_colors[face.z]);
    }
}
```

代码说明：

storeFacesPoints 函数的主要目的是将每个三角形面片的顶点位置和颜色信息提取并存储，以便将这些数据上传到 GPU 进行渲染。函数遍历 faces 向量中的每个三角形面片，每个面片包含了构成该三角形的三个顶点的索引。对于每个面片，函数根据这三个顶点的索引，从 vertex_positions 向量中获取实际的顶点位置信息，并从 vertex_colors 向量中获取对应的颜色信息。

然后，函数将每个三角形顶点的位置信息依次存储到 `points` 向量中，将颜色信息依次存储到 `colors` 向量中。这样一来，`points` 和 `colors` 向量就包含了所有三角形面片的顶点数据，以便在后续的渲染流程中传递给 GPU。

3. 打开 main.cpp 的 bindObjectAndData 函数中的注释代码

函数截图：（红色部分为需要解开注释的代码）

```
void bindObjectAndData(TriMesh* mesh, openGLObject& object, const std::string& vshader, const std::string& fshader) {

    // 创建顶点数组对象
    glGenVertexArrays(1, &object.vao);      // 分配1个顶点数组对象
    glBindVertexArray(object.vao);      // 绑定顶点数组对象

    // 创建并初始化顶点缓存对象
    glGenBuffers(1, &object.vbo);
    glBindBuffer(GL_ARRAY_BUFFER, object.vbo);
    glBufferData(GL_ARRAY_BUFFER,
        mesh->getPoints().size() * sizeof(glm::vec3) + mesh->getColors().size() * sizeof(glm::vec3),
        NULL,
        GL_STATIC_DRAW);

    // @TODO: Task3-修改完TriMesh.cpp的代码成后再打开下面注释，否则程序会报错
    glBufferSubData(GL_ARRAY_BUFFER, 0, mesh->getPoints().size() * sizeof(glm::vec3), &mesh->getPoints()[0]);
    glBufferSubData(GL_ARRAY_BUFFER, mesh->getPoints().size() * sizeof(glm::vec3), mesh->getColors().size() * sizeof(glm::vec3), &mesh->getColors()[0]);
```

图 1 解开注释代码

代码说明：

在为缓存分配了空间后，**glBufferSubData** 将实际的顶点数据和颜色数据分别上传到 GPU。顶点数据从缓存偏移量 0 开始存储，颜色数据则从顶点数据的末尾开始存储。

4. 在 main.cpp 中完善 display 函数

函数截图：（红色部分为需要补充的代码）

```
void display()
{
    // 清理窗口
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram(cube_object.program);

    glBindVertexArray(cube_object.vao);

    // 初始化变换矩阵
    glm::mat4 m(1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0);

    // @TODO: Task4-在此处修改函数，计算最终的变换矩阵
    // 调用函数传入三种变化的变化量，累加得到变化矩阵
    // 注意三种变化累加的顺序
    // 平移变换矩阵
    glm::mat4 translateM = glm::translate(glm::mat4(1.0), translateTheta);
    // 旋转变换矩阵
    glm::mat4 rotateM = glm::rotate(glm::mat4(1.0), rotateTheta.x, glm::vec3(1.0, 0.0, 0.0))
        * glm::rotate(glm::mat4(1.0), rotateTheta.y, glm::vec3(0.0, 1.0, 0.0))
        * glm::rotate(glm::mat4(1.0), rotateTheta.z, glm::vec3(0.0, 0.0, 1.0));
    // 缩放变换矩阵
    glm::mat4 scaleM = glm::scale(glm::mat4(1.0), scaleTheta);
    m = translateM * rotateM * scaleM * m;

    // 从指定位置matrixLocation中传入变换矩阵m
    glUniformMatrix4fv(cube_object.matrixLocation, 1, GL_FALSE, glm::value_ptr(m));

    // 绘制立方体中的各个三角形
    glDrawArrays(GL_TRIANGLES, 0, cube->getPoints().size());
}
```

图 2　display 函数

代码说明：（矩阵变换部分）

初始化变换矩阵 m 为单位矩阵，接着依次定义平移、旋转和缩放矩阵。平移矩阵 translateM 是通过 glm::translate 创建的，以 translateTheta 作为平移向量进行平移操作；旋转矩阵 rotateM 结合了三次旋转操作，分别围绕 x、y、z 轴旋转，旋转量由 rotateTheta 的分量提供；缩放矩阵 scaleM 则使用 scaleTheta 进行缩放操作。

translateTheta、rotateTheta、scaleTheta 三者的值是由 DEFAULT_DELTA 初始化的。当我们键盘按下 r 和 f 键时，三个量的值都会相应的改变，从而实现对变化率的控制。

为了实现正确的变换顺序，需要将 scaleM、rotateM、translateM 按顺序右乘上矩阵 m，从而计算出最终的变换矩阵。这是因为：

首先，缩放操作通常是对物体的尺寸进行改变，这种改变是相对于物体的中心点进行的。如果先进行平移或旋转，物体的中心点会发生变化，这可能会导致缩放效果不符合预期。例如，如果一个物体先被平移，然后再进行缩放，缩放的效果会基于平移后的位置进行，这可能导致缩放效果不符合设计意图。

其次，平移操作是移动物体的位置。如果平移在缩放和旋转之后进行，物体

的位置会基于已经旋转和缩放后的位置，这样可以确保平移是基于已经调整好方向和尺寸的物体进行的。如果平移在缩放和旋转之前进行，物体的位置会基于原始位置进行移动，这可能导致物体的相对位置不符合设计意图。

因此，在变换时的顺序是：**缩放 → 旋转 → 平移**，以保证最终变换的效果符合设计需求。

5. 修改程序运行窗口标题、大小

```
// 配置窗口属性
GLFWwindow* window = glfwCreateWindow(1000, 1000, "2022150168_吴嘉楷_实验2.3", NULL, NULL);
```

图 3 修改程序运行窗口配置

6. 部分运行结果图

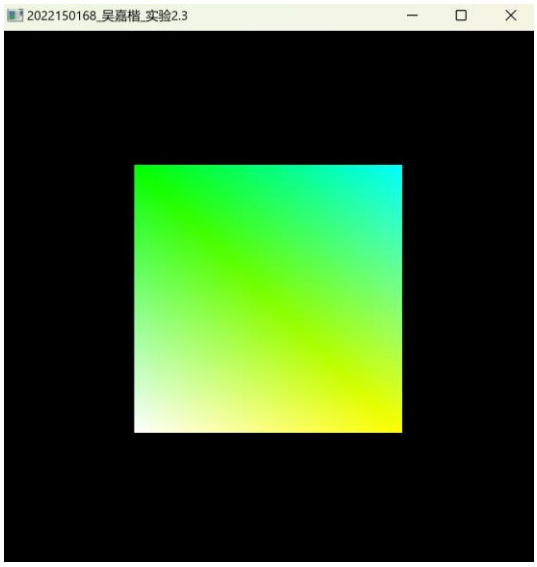

图 4 不做任何操作的图形

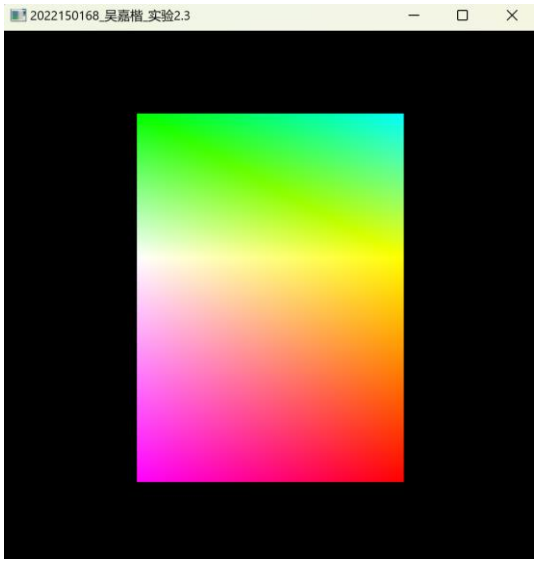

图 5 沿 X 轴放大后的图形

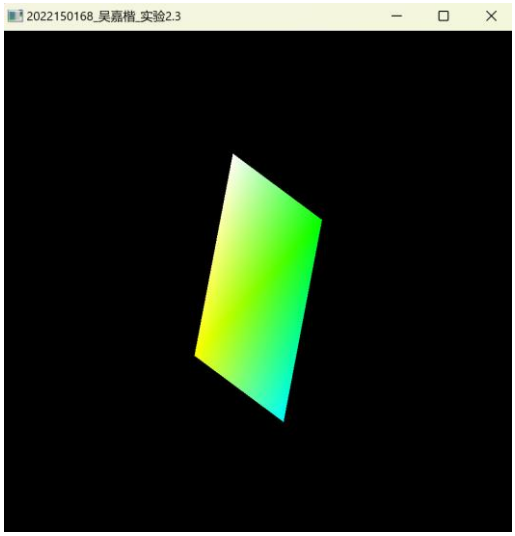

图 6 沿某个轴缩小到 0 后的图形

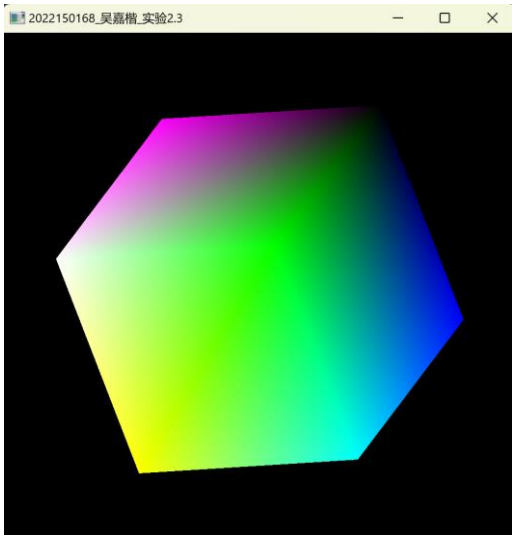

图 7 经过旋转后的图形

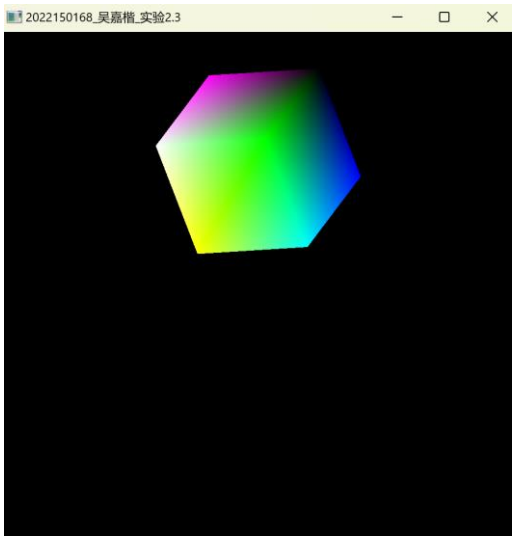

图 8 经过缩小即平移后的图形

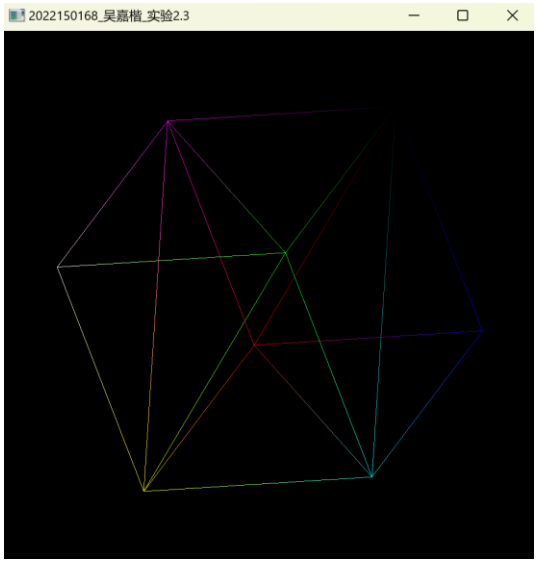

图 9 经过旋转和线条填充后的图形

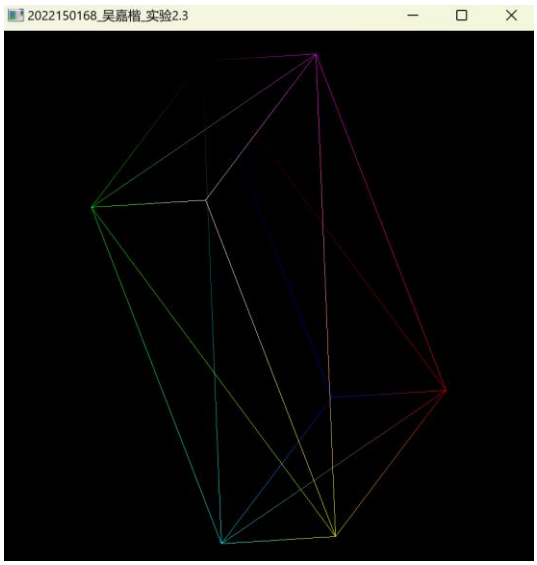

图 10 经过旋转、线条填充和缩放后的图形