计算机图形学实验报告

第一部分：实现思路和代码分析

代码截图	代码说明
`std::vector<GLfloat> getVertices() {` `  std::vector<GLfloat> result{` `  // 面1，红色` `  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,//` `  // 面2，蓝色` `  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,//` `  // 面3，绿色` `  -2.0f,  2.0f, -2.0f, 0.0f, 1.0f, 0.0f,` `  -2.0f, -2.0f, -2.0f, 0.0f, 1.0f, 0.0f,//` `  -2.0f, -2.0f,  2.0f, 0.0f, 1.0f, 0.0f,` `  -2.0f,  2.0f,  2.0f, 0.0f, 1.0f, 0.0f,//` `  // 面4，紫色` `  2.0f,  2.0f, -2.0f, 0.5f, 0.0f, 1.0f,` `  2.0f, -2.0f, -2.0f, 0.5f, 0.0f, 1.0f,//` `  2.0f, -2.0f,  2.0f, 0.5f, 0.0f, 1.0f,` `  2.0f,  2.0f,  2.0f, 0.5f, 0.0f, 1.0f,//` `  // 面5，黄色` `  2.0f, -2.0f, -2.0f, 1.0f, 1.0f, 0.0f,` `  2.0f, -2.0f,  2.0f, 1.0f, 1.0f, 0.0f,//` `  -2.0f, -2.0f,  2.0f, 1.0f, 1.0f, 0.0f,` `  -2.0f, -2.0f, -2.0f, 1.0f, 1.0f, 0.0f,//` `  // 面6，粉红色` `  2.0f,  2.0f, -2.0f, 1.0f, 0.0f, 1.0f,` `  2.0f,  2.0f,  2.0f, 1.0f, 0.0f, 1.0f,//` `  -2.0f,  2.0f,  2.0f, 1.0f, 0.0f, 1.0f,` `  -2.0f,  2.0f, -2.0f, 1.0f, 0.0f, 1.0f //` `  };` `  return result;` `}`	创建一个边长为 4 的立方体。将立方体每个面切分成两个三角形，为了代码的可阅读性，每个面都使用了 4 个点进行表示。因此表示一个立方体共需要 12 个三角形，共 24 个点（同一面上的两个三角形共用位于对角线上的点，后面标有”\\”的即为公共点）。
`std::vector<unsigned int> getIndices() {` `  std::vector<unsigned int> indices{` `  // 面1` `  0,  1,  3,` `  2,  1,  3,` `  // 面2` `  4,  5,  7,` `  6,  5,  7,` `  // 面3` `  8,  9, 11,` `  10,  9, 11,` `  // 面4` `  12, 13, 15,` `  14, 13, 15,` `  // 面5` `  16, 17, 19,` `  18, 17, 19,` `  // 面6` `  20, 21, 23,` `  22, 21, 23` `  };` `  return indices;` `}`	使用 EBO 进行图形的渲染，这里是 index 数组，用于记录每个面上组成每个小三角形的三个点对应的顶点坐标。
`void Translation(glm::mat4& transform, glm::vec3 shaft) {` `  transform = glm::translate(transform, shaft);` `}`	使用 GLM 库的 translate 函数进行平移变换，传入变换矩阵和平移距离。
`void Rotation(glm::mat4& transform, float speed, glm::vec3 shaft) {` `  transform = glm::rotate(transform, speed, shaft);` `}`	使用 GLM 库的 rotate 函数进行旋转变换，传入变换矩阵、旋转速度和旋转轴。
`void Scale(glm::mat4& transform, glm::vec3 shaft) {` `  transform = glm::scale(transform, shaft);` `}`	使用 GLM 库的 scale 函数进行缩放变换，传入变换矩阵和缩放比例。
`// 创建顶点着色器` `const char* shader = "#version 330 core\n\` `  layout(location = 0) in vec3 pos;\n\` `  layout(location = 1) in vec3 color;\n\` `  uniform mat4 view;\n\` `  uniform mat4 transform;\n\` `  out vec3 outColor;\n\` `  void main() {\n\` `  gl_Position = view * transform * vec4(pos, 1.0);\n\` `  outColor = color;\n\` `  }\0";`	修改顶点着色器，添加 uniform 变量，用于允许外部函数将变换矩阵传入 GLSL 中实现相应的图形变换。

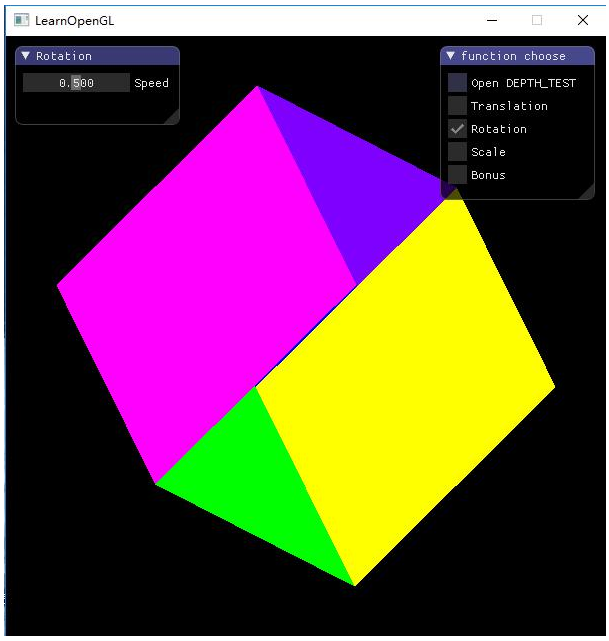
`` // 顶点数组 GLfloat* vertices = NULL; unsigned int* indices = NULL; int total_v = 0, total_ind = 0; std::vector<GLfloat> v; std::vector<unsigned int> ind; // 作图 v = hw4.getVertices(); ind = hw4.getIndices(); total_v = v.size(); total_ind = ind.size(); vertices = new GLfloat[total_v]; indices = new unsigned int[total_ind]; for (int i = 0; i < total_v; i++) { vertices[i] = v[i]; } for (int i = 0; i < total_ind; i++) { indices[i] = ind[i]; } ```	用于创建立方体的顶点数组以及相应的索引数组。
`` if (enableDT) { // 开启深度测试 glEnable(GL_DEPTH_TEST); // 开启深度测试，清空深度测试缓存 glClear(GL_DEPTH_BUFFER_BIT); } else { // 关闭深度测试 glDisable(GL_DEPTH_TEST); } ```	通过 ImGui 设置变量 enableDT，选择是否开启深度测试，观察两种模式的差别。
`` if (Translation_func) { if (leftright) { t_x = 2 * sin(glfwGetTime()); hw4.Translation(transform, glm::vec3(t_x, 0.0f, 0.0f)); } else { t_y = 2 * sin(glfwGetTime()); hw4.Translation(transform, glm::vec3(0.0f, t_y, 0.0f)); } } ```	通过 ImGui 设置变量 Translation_func，进行平移变换。由于 glfwGetTime()的值随时间递增，而为了避免超出视窗，需要将其控制在(0,1)之间。这里我选择了 sin 函数进行转化，确保了横向移动时 x 坐标满足要求，纵向移动时，y 坐标满足要求。
`` if (Rotation_func) { if (Bonus_func) { hw4.Scale(transform, glm::vec3(0.3, 0.3, 0.3)); } hw4.Rotation(transform, speed * (float)glfwGetTime(), glm::vec3(1.0f, 0.0f, 1.0f)); } ```	通过 ImGui 设置变量 Rotation_func，进行旋转变换。这里需要满足围绕 XoZ 平面的 x=z 轴进行旋转，因此将 vec3 的 x 和 z 设置为 1。glfwGetTime()用于持续旋转，speed 控制旋转的速度。
`` if (Scale_func) { scale_size = (sin(glfwGetTime()) + 1.5) * 0.5; hw4.Scale(transform, glm::vec3(scale_size, scale_size, scale_size)); } ```	通过 ImGui 设置变量 Scale_func，进行缩放变换。由于 glfwGetTime()的值随时间递增，而为了避免缩放过大，我同样选择了 sin 函数进行范围的控制。
`` // 将变换矩阵应用到坐标中 unsigned int transformLoc = glGetUniformLocation(shaderProgram, "transform"); unsigned int viewLoc = glGetUniformLocation(shaderProgram, "view"); glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(transform)); glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]); // 绘制物体 glBindVertexArray(VAO); // 绘制立方体 glDrawElements(GL_TRIANGLES, total_ind * sizeof(unsigned int), GL_UNSIGNED_INT, (void*)0); ```	将得到的变换矩阵应用到顶点着色器中，实现相应的变换。
`` if (Bonus_func) { transform = glm::mat4(1.0f); hw4.Scale(transform, glm::vec3(0.2, 0.2, 0.2)); t_y = 16 * sin((float)glfwGetTime()); t_x = 16 * cos((float)glfwGetTime()); hw4.Translation(transform, glm::vec3(t_x, t_y, 0.0f)); hw4.Rotation(transform, 5 * (float)glfwGetTime(), glm::vec3(1.0f, 0.0f, 1.0f)); glUniformMatrix4fv(transformLoc, 1, GL_FALSE, &transform[0][0]); // 绘制物体 glBindVertexArray(VAO); // 绘制立方体 glDrawElements(GL_TRIANGLES, total_ind * sizeof(unsigned int), GL_UNSIGNED_INT, (void*)0); } ```	重新设计一个变换矩阵（用于进行模拟天体的公转运动和自转运动），替换前面的变换矩阵，应用到顶点着色器中，实现 Bonus 效果（类似月球围绕地球进行公转和自转）。

第二部分：运行结果截图和相关问题解答

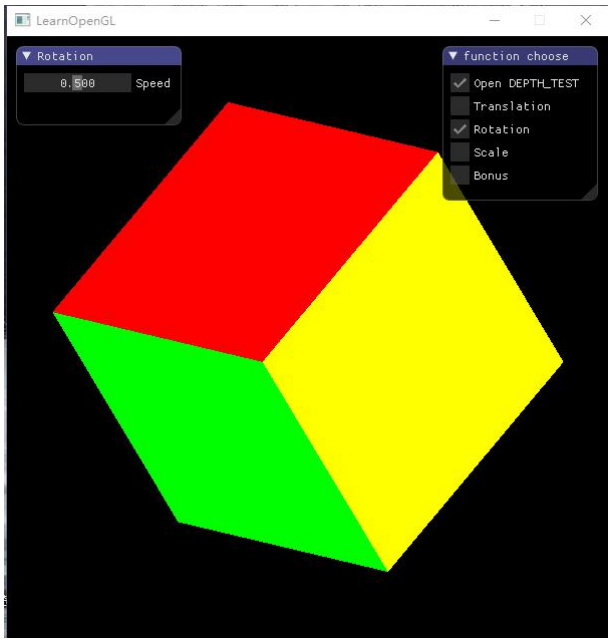
1. 画一个立方体(cube)：边长为 4， 中心位置为(0, 0, 0)。分别启动和关闭深度测试

glEnable(GL_DEPTH_TEST) 、 glDisable(GL_DEPTH_TEST) ，查看区别，并分析原因。

关闭深度测试：



启动深度测试：

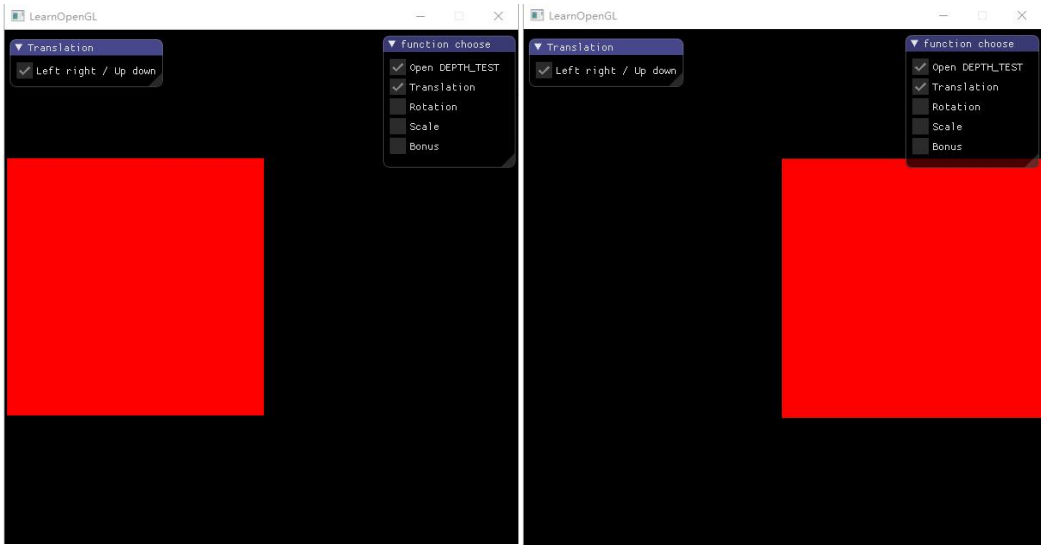


区别：不启用深度测试时立方体的表面不能正确显示，有些原本属于可以观察到的面被错误地丢弃，而原本不能被观测到的面则被错误地渲染出来。

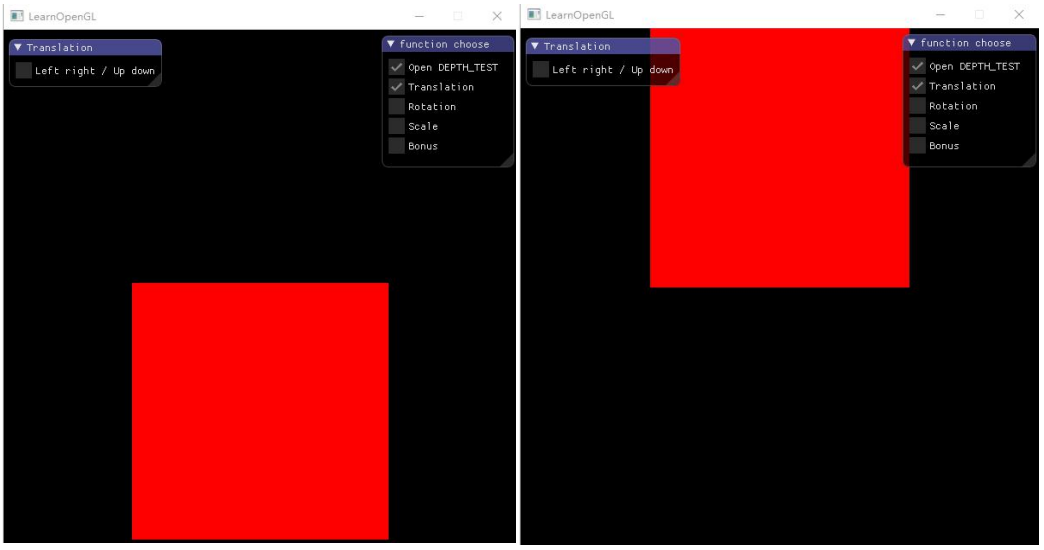
原因：在不启用深度测试的时候，如果先绘制一个距离较近的物体，再绘制距离较远的物体，则距离远的物体会因为绘制顺序靠后而把距离近的物体覆盖掉。而开启深度测试后，OpenGL 提供深度缓冲区用于存储像素的深度信息。在绘制一个新片段时，OpenGL 会将该片段的的深度值与深度缓冲的内容进行对比。OpenGL 会执行一个深度测试，如果这个测试通过了的话，深度缓冲将会更新为新的深度值。如果深度测试失败了，片段将会被丢弃。因此绘制的物体能按照离观察者距离的远近正常显示。

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

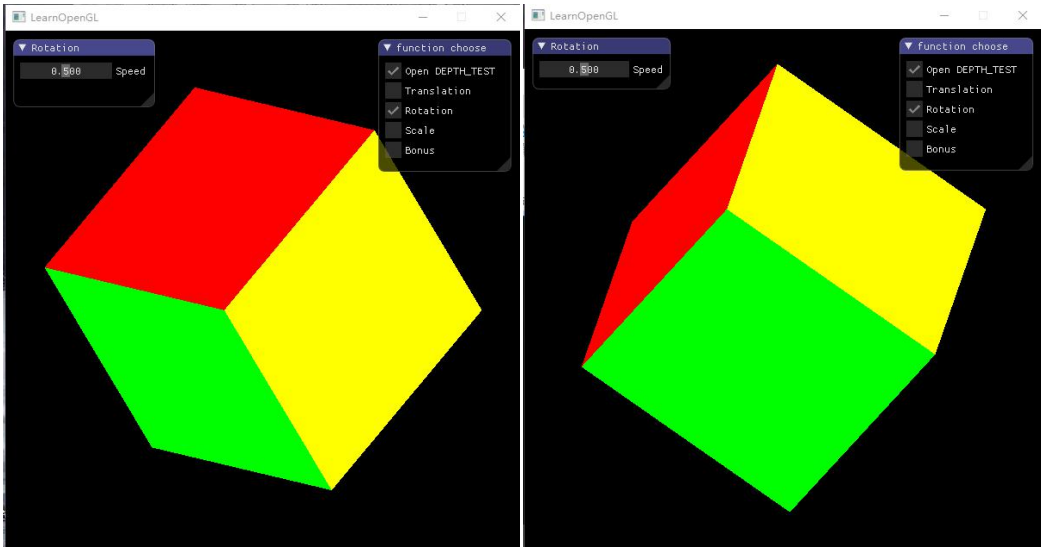
水平移动：



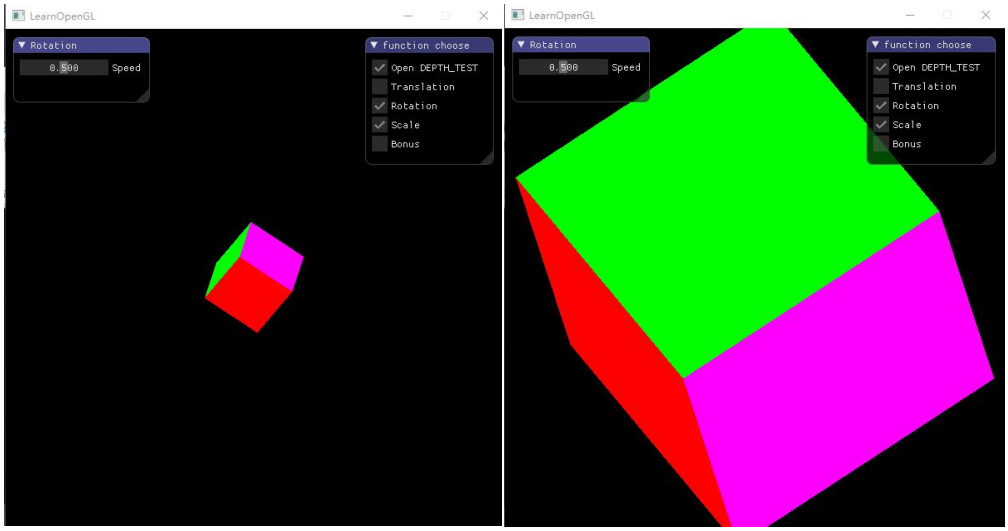
垂直移动：



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

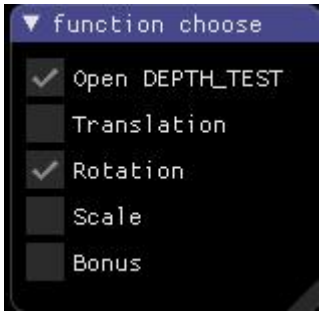


4. 放缩(Scaling)：使画好的 cube 持续放大缩小。



5. 在 GUI 里添加菜单栏，可以选择各种变换。

主菜单：选择是否开启深度测试，平移，旋转，缩放以及 bonus 变换



平移菜单：选择水平或垂直移动

旋转菜单：选择旋转速度

6. 结合 Shader 谈谈对渲染管线的理解

OpenGL 在渲染处理过程中会顺序执行一系列操作，这一系列相关的处理阶段就被称为 OpenGL 的渲染管线。Shader 是可编程管线，这部分管线可以动态编程，实现动态修改渲染效果而不必重写编译代码的目标，主要分为 Vertex Shader（顶点着色器）和 Fragment Shader（片段着色器）。

工作流程：

渲染管线在处理和显示图像数据时，需要首先将图形的顶点数据保存到缓存对象中，然后调用 Vertex Shader 对所有的顶点数据进行变换（如：计算定点的位置变换，光照以及纹理坐标的变换等）。渲染管线将变换后的顶点信息及其构成的相关几何图元的所有信息在图元装配阶段组织起来，准备进行剪切和光栅化操作。裁剪和剪切阶段会对几何图元进行修改，确保相关像素不会出现在窗口以外。光栅化阶段是判断某一部分的几何体所覆盖的屏幕空间。

光栅化后进入片元处理阶段，在这个阶段可以使用 Fragment Shader 对光栅化后的片元进行纹理映射、着色等操作，计算出每个像素的最终颜色。经过片段测试（如：Alpha Test，Blending 等）得到最终图像后执行帧缓冲的写入等操作，渲染出最后的图像。

Bonus：模拟月球围绕地球进行公转运动，同时包括了地球的自转和月球的自转。