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;` `}`	沿用上次作业使用的正方体，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。
`// 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 projection;\n\` `  uniform mat4 view;\n\` `  uniform mat4 transform;\n\` `  out vec3 outColor;\n\` `  void main() {\n\` `  gl_Position = projection * 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。相比上次作业，添加了 projection 矩阵
`if (modeChange == 1) {` `  transform = glm::translate(transform, glm::vec3(-1.5f, 0.5f, -1.5f));` `  view = glm::lookAt(glm::vec3((cameraRadius - 25.0f) / 5 , 0.0f, -10.0f),` `  glm::vec3(0.0f, 0.0f, 0.0f),` `  glm::vec3(0.0f, 1.0f, 0.0f));` `  if (Orth_or_Pers) {` `  projection = glm::ortho(O_left, O_right, O_bottom, O_top, O_near, O_far);` `  }` `  else {` `  projection = glm::perspective(glm::radians(P_radians),` `  (float)WINDOW_WIDTH / (float)WINDOW_HEIGHT,` `  P_near, P_far);` `  }` `}`	作业第一部分：正交变换和透视变换。首先使用 transform 矩阵将正方体平移到（-1.5，0.5，-1.5）的位置，然后设置照相机的位置（view 矩阵）为（x，0，-10）【为了方便观察，x 设置为可调整】。然后分别输入正交投影变换、透视变换需要的参数值，经过计算得到 peojection 矩阵的真实值。

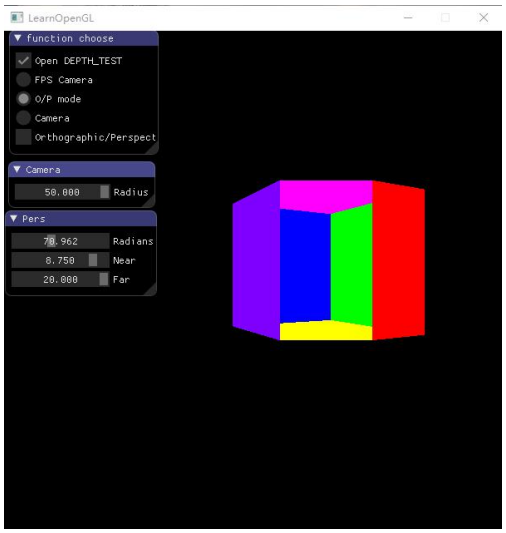
`` else if (modeChange == 2) { float camPosX = sin(glfwGetTime()) * cameraRadius; float camPosZ = cos(glfwGetTime()) * cameraRadius; view = glm::lookAt(glm::vec3(camPosX, 0.0f, camPosZ), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f)); projection = glm::perspective(glm::radians(45.0f), (float)WINDOW_WIDTH / (float)WINDOW_HEIGHT, 0.1f, 100.0f); } ```	第二部分：视角变换。这部分将正方体固定在（0，0，0），因此不需要修改 transfrom 矩阵。view 矩阵使用 glfwGetTime()以及三角函数实现实时位置的计算，使得摄像机在 XoZ 平面上圆周运动。图像使用透视变换进行最后的处理（projection 矩阵）。
`` else{ view = camera.getView(); projection = glm::perspective(glm::radians(45.0f), (float)WINDOW_WIDTH / (float)WINDOW_HEIGHT, 0.1f, 100.0f); } ```	第三部分：Bonus 的 Camera 类。这部分同样将正方体固定在（0，0，0）位置。view 矩阵使用 Camera 类根据键盘输入和鼠标移动实时计算，projection 矩阵依旧使用透视变换。
`` // 将变换矩阵应用到坐标中 unsigned int transformLoc = glGetUniformLocation(shaderProgram, "transform"); unsigned int viewLoc = glGetUniformLocation(shaderProgram, "view"); unsigned int projectionLoc = glGetUniformLocation(shaderProgram, "projection"); glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(transform)); glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]); glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, &projection[0][0]); ```	将 transform 矩阵，view 矩阵，projection 矩阵应用到渲染器上。
`` void key_input(GLFWwindow* window, float deltaTime) { if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) { camera.ProcessKeyboard(UP, deltaTime); } if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) { camera.ProcessKeyboard(DOWN, deltaTime); } if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) { camera.ProcessKeyboard(LEFT, deltaTime); } if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) { camera.ProcessKeyboard(RIGHT, deltaTime); } } ```	处理键盘输入的函数。w 键对应往前移动操作，调用 ProcessKey 函数中的 UP 部分，镜头向前移动；s 键对应往后移动操作，调用 ProcessKey 函数中的 DOWN 部分，镜头向后移动；a 键对应往左移动操作，调用 ProcessKey 函数中的 LEFT 部分，镜头向左移动；d 键对应往右移动操作，调用 ProcessKey 函数中的 RIGHT 操作，镜头向右移动。
`` void moveForward(float cameraSpeed) { cameraPos += cameraSpeed * this->cameraFront; }; void moveBack(float cameraSpeed) { cameraPos -= cameraSpeed * this->cameraFront; }; void moveLeft(float cameraSpeed) { cameraPos -= this->cameraRight * cameraSpeed; }; void moveRight(float cameraSpeed) { cameraPos += this->cameraRight * cameraSpeed; }; void ProcessKeyboard(MOVE_DIRECTION direction, float deltaTime) { float cameraSpeed = this->MovementSpeed * deltaTime; if (direction == UP) { this->moveForward(cameraSpeed); } if (direction == DOWN) { this->moveBack(cameraSpeed); } if (direction == LEFT) { this->moveLeft(cameraSpeed); } if (direction == RIGHT) { this->moveRight(cameraSpeed); } //cameraPos.y = 0.0f; }; ```	Camera 类中处理键盘输入的部分函数。cameraFront 表示视角向前正方向的单位向量，cameraRight 表示视角右面的单位向量，通过 cameraFront 和 cameraUp（表示视角垂直方向的单位向量）的叉乘操作得到。当接收到键盘输入时，根据输入选择视角移动的方向，执行相应的操作。

`void mouse_callback(GLFWwindow* window, double xpos, double ypos)` `{` `  if (firstTimeUsingMouse) {` `  lastX = xpos;` `  lastY = ypos;` `  firstTimeUsingMouse = false;` `  }` `  float xoffset = xpos - lastX,` `  yoffset = ypos - lastY;` `  lastX = xpos;` `  lastY = ypos;` `  camera.ProcessMouseMove(xoffset, yoffset);` `}`	处理鼠标输入的函数。通过检测鼠标当前位置与上一次的位置的差异，求出鼠标的位移值。将这个位移值传入 Camera 类的处理鼠标输入的函数，同时保存新的鼠标坐标值。
`void ProcessMouseMove(double xoffset, double yoffset) {` `  // 设置敏感度` `  float sensitivity = 0.1f;` `  xoffset *= sensitivity;` `  yoffset *= sensitivity;` `  // 改变俯仰角` `  this->yaw += xoffset;` `  this->pitch += yoffset;` `  if (this->constrainPitch) {` `  if (this->pitch > 89.0f) {` `  this->pitch = 89.0f;` `  }` `  if (this->pitch < -89.0f) {` `  this->pitch = -89.0f;` `  }` `  }` `  updateVector();` `};`	Camera 类中处理鼠标输入的函数。首先将输入的鼠标移动距离乘以一个敏感值，敏感值越小镜头晃动越不明显。然后调整镜头的俯仰角和偏航角。最后更新整个 Camera 的坐标向量。
`void updateVector() {` `  glm::vec3 front;` `  front.x = cos(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));` `  front.y = sin(glm::radians(this->pitch));` `  front.z = sin(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));` `  this->cameraFront = glm::normalize(front);` `  this->cameraRight = glm::normalize(glm::cross(cameraFront, cameraUp));` `  this->cameraUp = glm::normalize(glm::cross(cameraRight, cameraFront));` `}`	获取相机的坐标向量。首先通过俯仰角和偏航角求出镜头正向的方向向量。然后根据此向量分别求：正向单位向量（cameraFront），右单位向量（cameraRight）以及垂直单位向量（cameraUp）。
`glm::mat4 getView() const {` `  return glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);` `};`	根据正向单位向量和右单位向量求出 view 矩阵。

第二部分：运行结果截图和相关问题解答

1. 投影(Projection):

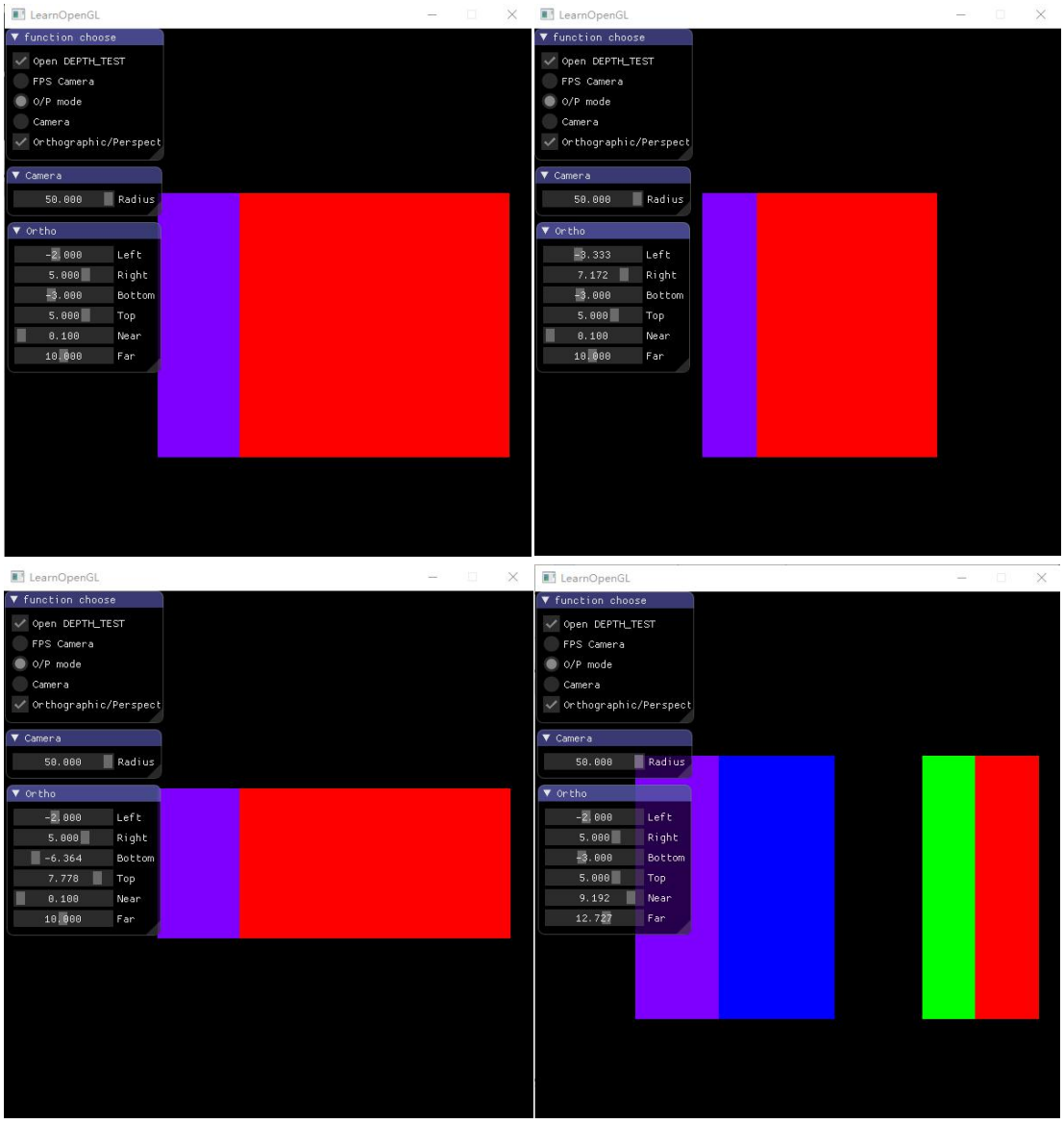
（1）把上次作业绘制的 cube 放置在(-1.5, 0.5, -1.5)位置，要求 6 个面颜色不一致



利用透视变换的近平面对立方体进行切割可以观察到正方体的六个面颜色不相同。

（2）正交投影(orthographic projection)：实现正交投影，使用多组(left, right, bottom, top, near, far)参数，比较结果差异

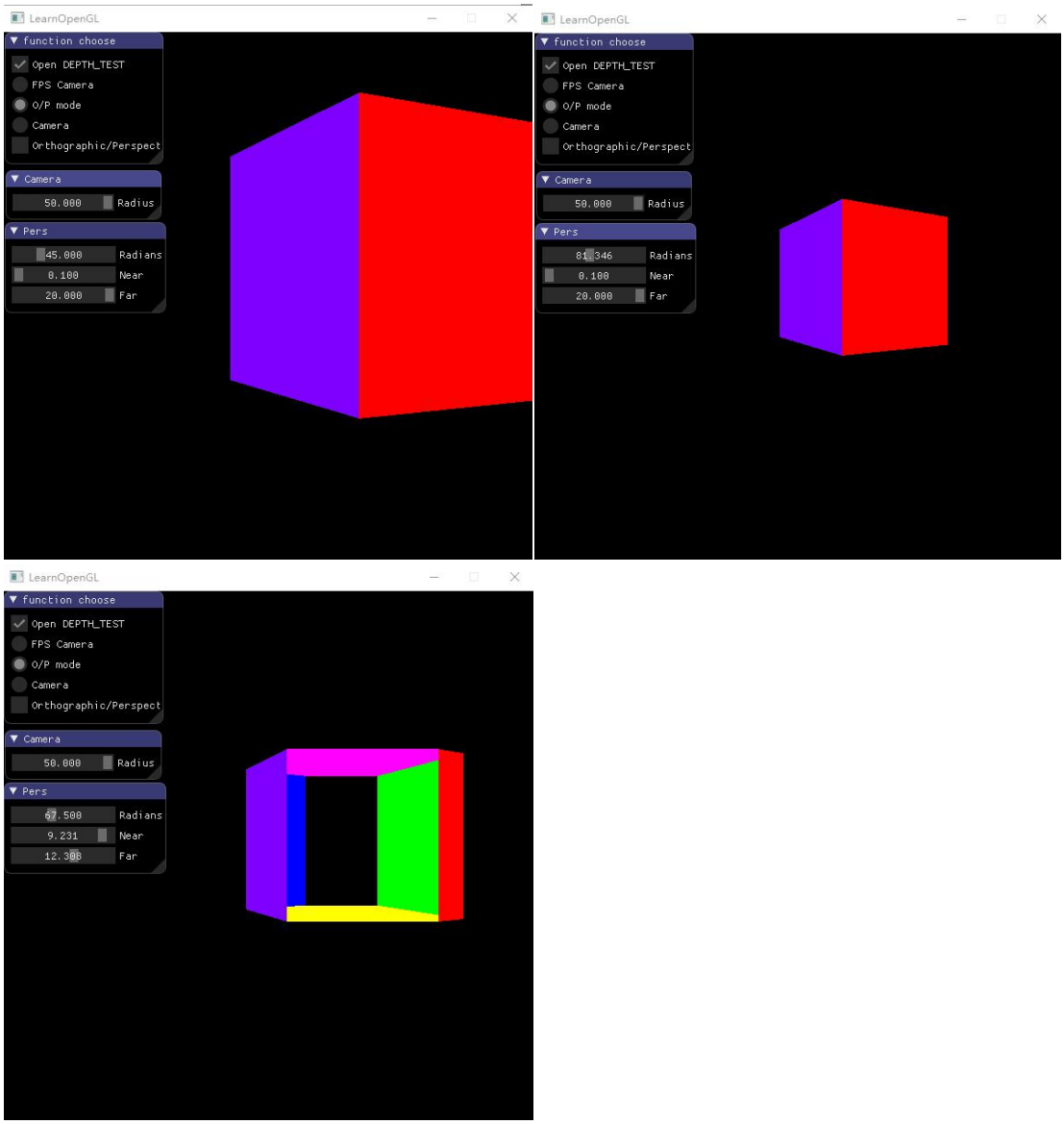
从左到右，从上到下依次对应的操作为：
原图，调整 left、right 参数，调整 bottom、up 参数，调整 near、far 参数



可以观察到，调整 left、right 参数，图像会出现水平方向的形变；调整 bottom，top 参数，图像会出现垂直方向的形变；调整 near、far 参数，部分图像会被剪裁，如图 4 中的蓝色和绿色的出现是由于近平面将红紫平面交界处的图像裁剪后出现空隙，然后 OpenGL 将原本不可见的蓝色面和绿色面渲染出来，而蓝色面和绿色面中间出现的不规则缺面则是因为远平面将蓝绿平面交界处的图像裁剪导致的。

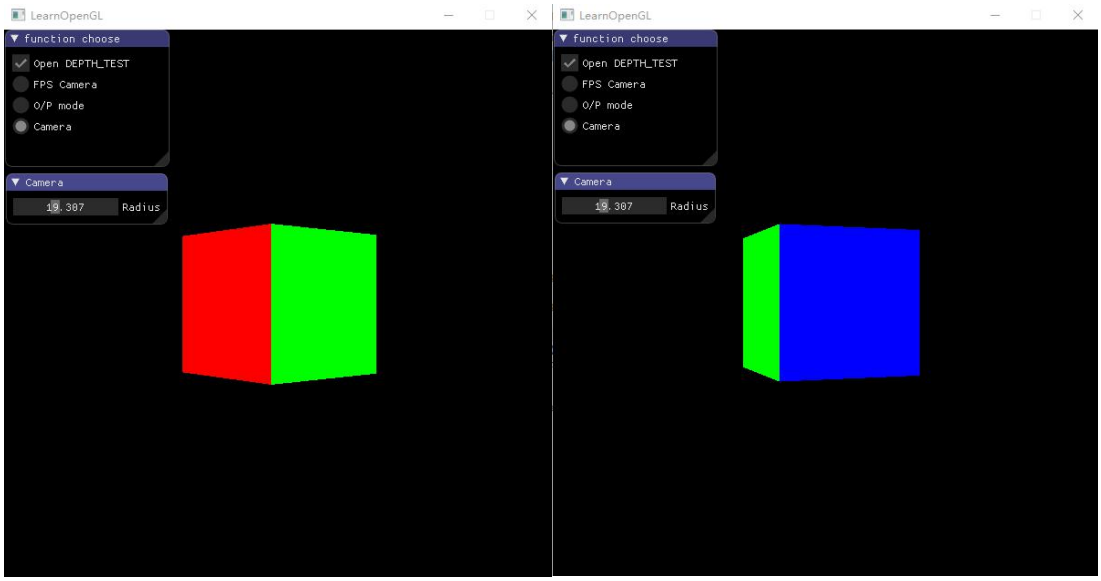
（3）透视投影(perspective projection)：实现透视投影，使用多组参数，比较结果差异

从左到右，从上到下依次对应的操作为：
原图，调整视野（FOV）参数，调整 near、far 参数

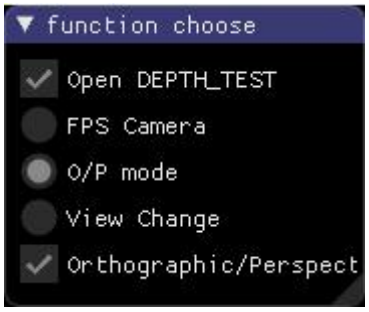


可以观察到，调整 radians 参数，图像会出现缩放，这是因为 randians 是视野角参数，视野越大，视野角越大，视野中的物体距离摄像机越远，体积越小；调整 near、far 参数，部分图像会被剪裁，如图 4 中的蓝色、绿色等平面颜色的出现是由于近平面将红紫平面交界处的图像裁剪后出现空隙，然后 OpenGL 将原本不可见的蓝色面、绿色面等平面渲染出来，而蓝色面和绿色面中间出现的不规则缺面则是因为远平面将蓝绿平面交界处的图像裁剪导致的。

2. 视角变换(View Changing):把 cube 放置在(0, 0, 0)处，做透视投影，使摄像机围绕 cube 旋转，并且时刻看着 cube 中心

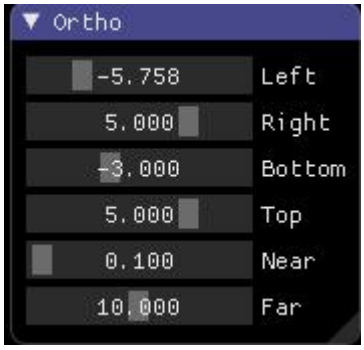


3. 在 GUI 里添加菜单栏，可以选择各种功能。

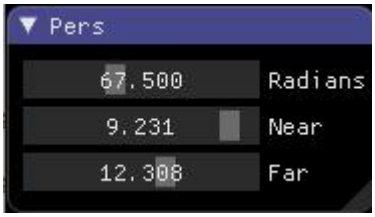


用于选择正方体的显示模式，FPS Camera 是 Bonus 中的摄影机；O/P mode 是显示正交投影和透视投影的摄像机；View Change（即前面图片中的 Camera 选项）是旋转摄影机。

在 O/P mode 模式用于调整摄像机的 X 坐标；在 View Change 模式用于调整摄像机的旋转半径。



在 O/P mode 模式用于调整正交投影的各个参数



在 O/P mode 模式用于调整透视投影的各个参数

4. 在现实生活中，我们一般将摄像机摆放的空间 View matrix 和被拍摄的物体摆设的空间 Model matrix 分开，但是在 OpenGL 中却将两个合二为一设为 ModelView matrix，通过上面的作业启发，你认为是为什么呢？

通过使用 ModelView 矩阵，可以将模型上的顶点从模型空间直接变换到摄像机的视图空间。我认为 OpenGL 这样做有两个好处：

（1）假如在世界空间中有许多模型，并且每个模型包含许多顶点。那么用一个 ModelView 矩阵把单个模型的所有顶点一次性变换到视图空间，比对所有顶点都做两次矩阵变换要高效得多。

（2）将顶点从模型空间变换到世界空间中，所做的变换常常需要根据顶点离世界坐标的原点的距离使用不同的精度去计算。同样的，当再把顶点从世界空间变换到视图空间时，所做变换的精度也需要根据顶点到摄像机的距离。当我们使用多个摄像机时，对那些距离摄像机很近的顶点采用高精度是合适的，但对于距离摄像机很远的顶点同样采用高精度则会产生精度损失。当模型与摄像机离得很近，并且两者又离世界坐标系的原点很远时，变换两次容易产生精度损失。用 ModelView 矩阵可以有效减少精度损失，提高结果的精确性。

Bonus：实现一个 camera 类，当键盘输入 w,a,s,d，能够前后左右移动；当移动鼠标，能够视角移动("look around")，即类似 FPS(First Person Shooting)的游戏场景