Homework 5 - Camera

Basic:

1. 投影（Projection）：

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

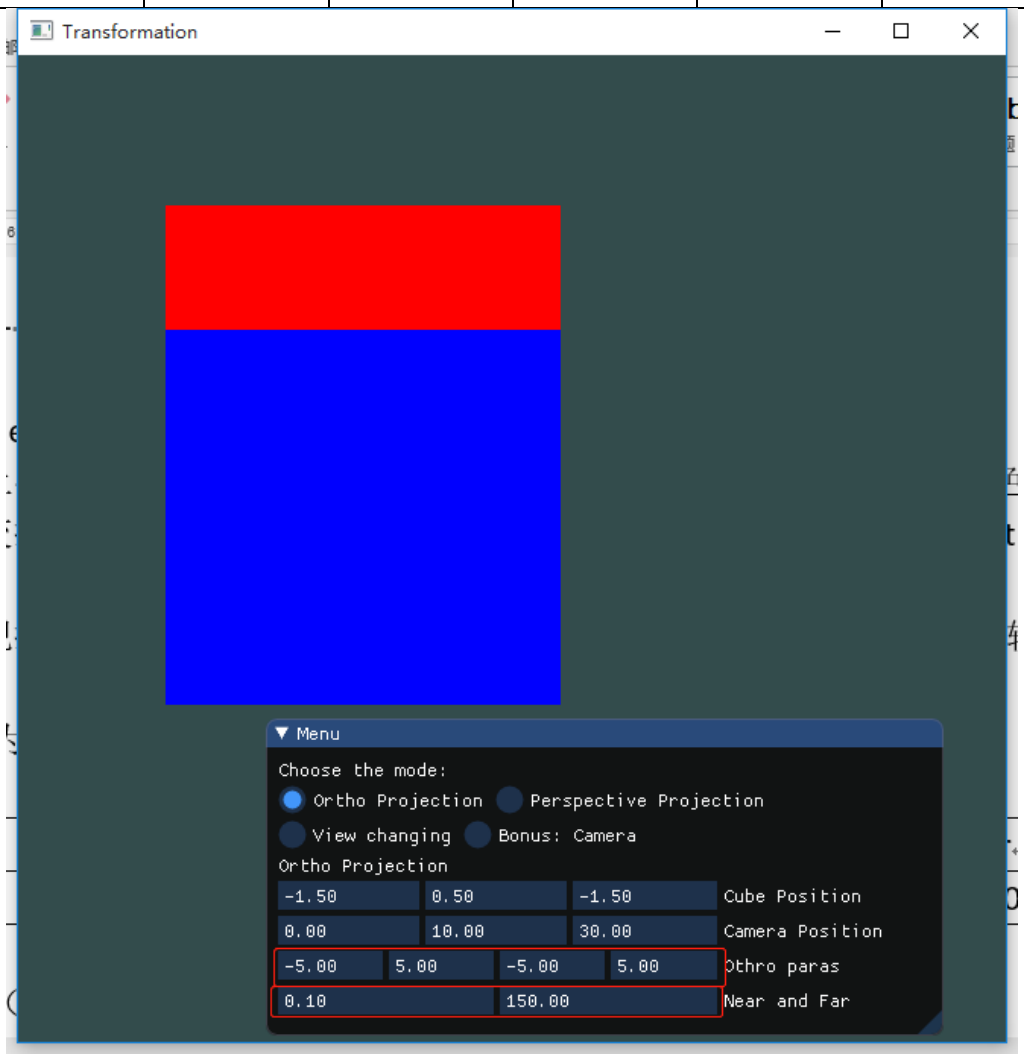
实现结果：

Cube 为变长为 4，位置在（-1.5,0.5，-1.5）的正方体。

正交投影：

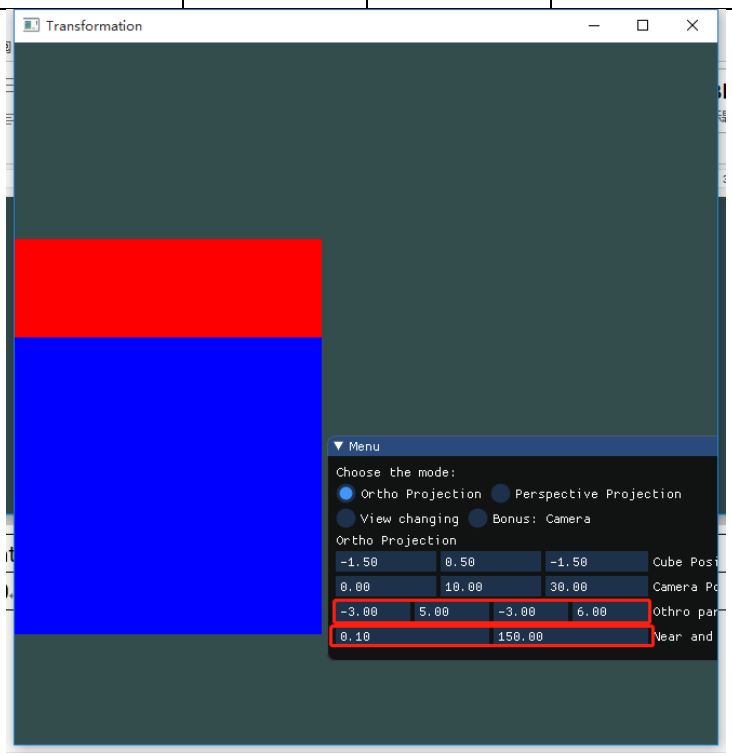
参数为：

Left	Right	Bottom	Top	Near	Far
-5.00	5.00	-5.00	5.00	0.10	150.00



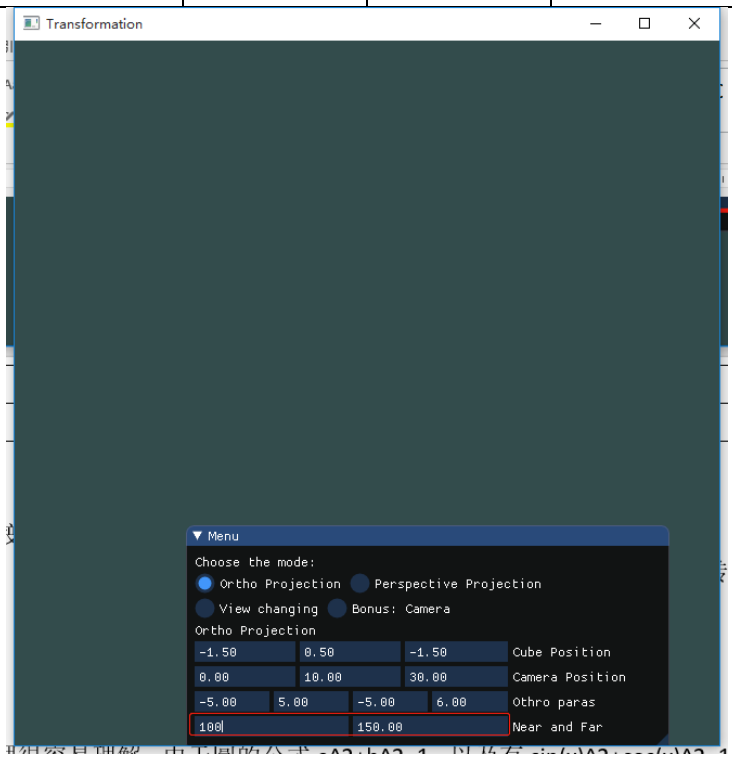
参数为：

Left	Right	Bottom	Top	Near	Far
-3.00	5.00	-3.00	6.00	0.10	150.00



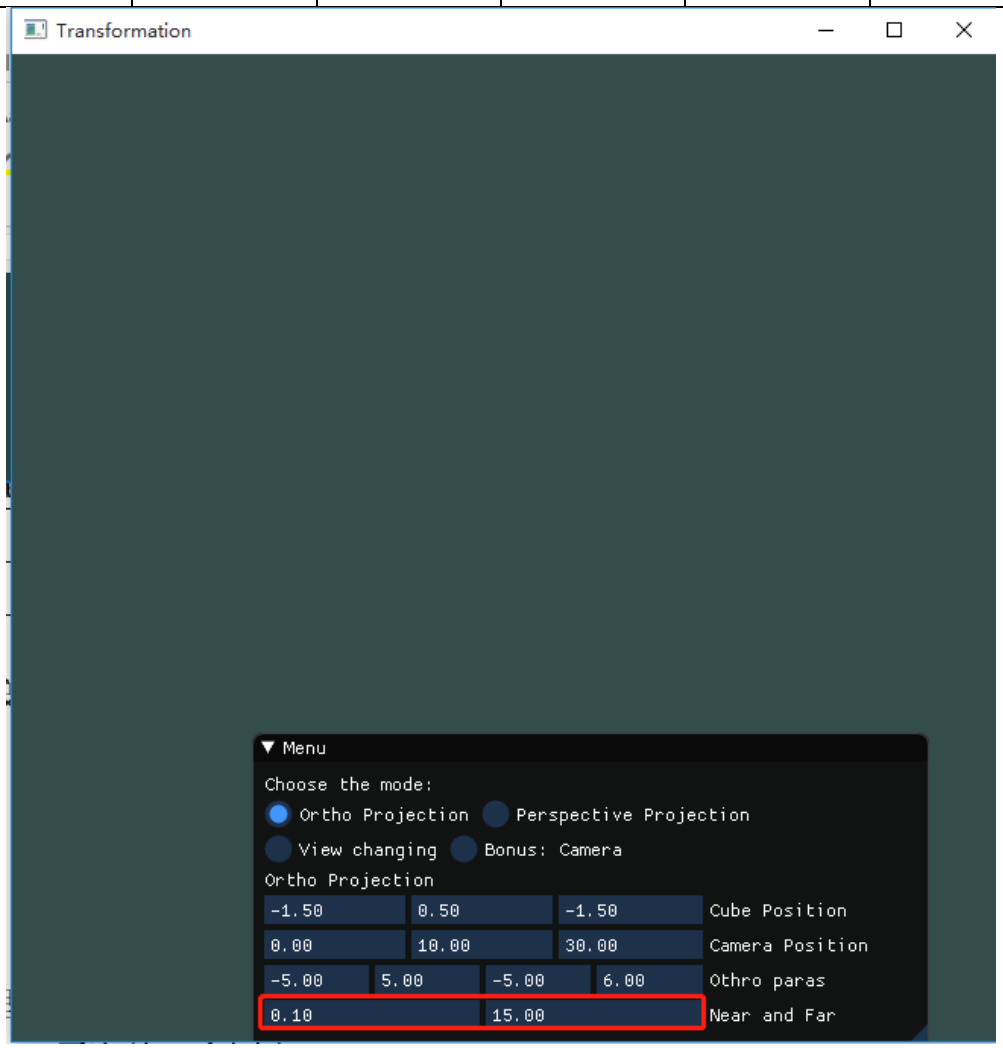
参数为：

Left	Right	Bottom	Top	Near	Far
-5.00	5.00	-5.00	5.00	100.00	150.00



参数为：

Left	Right	Bottom	Top	Near	Far
-5.00	5.00	-5.00	5.00	0.10	15.00



实现语句：

```
projection = glm::ortho(ortho[0], ortho[1], ortho[2], ortho[3], eye_space[0], eye_space[1]);
```

在 GUI 里设计接口改变变量的值，然后将相应变量传进 model，view，projection 的矩阵即可。

差异：

正交投影矩阵直接将坐标映射到 2D 平面（屏幕）中，没有将透视考虑进来，失去了真实世界中近大远小这一性质。而且，将 cube 进行位置的平移后，只有其处于创建的投影立方体的内部部分能够显示出来。

当修改参数 left，right，bottom，top 时，可以发现：

- 如果投影立方体的高度和宽度不同的话，cube 的长宽压缩程度也会不同，在视图中则会变成长方体。

- 随着影立方体的高度和宽度的增加，视野中的立方体会越来越小。

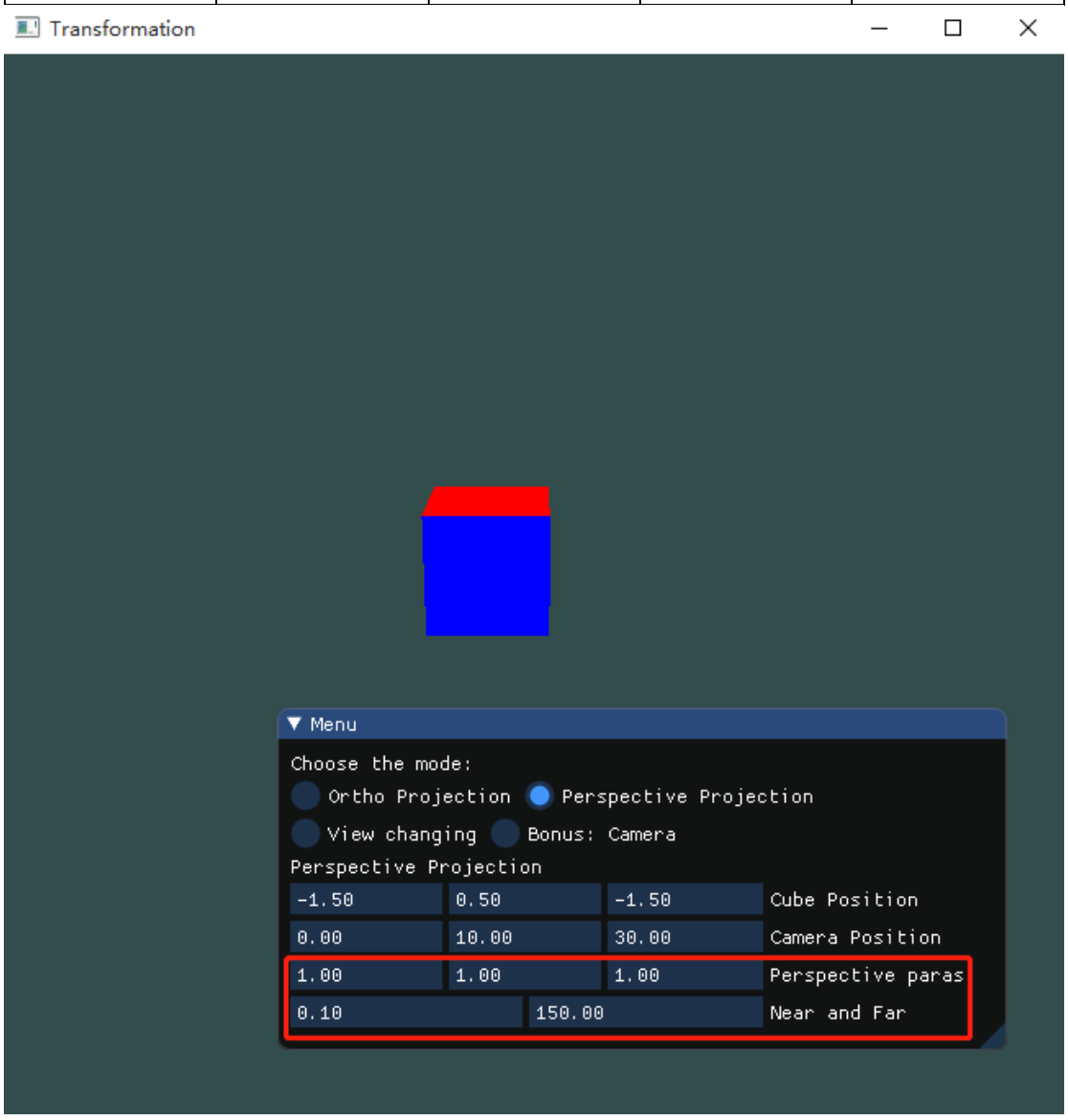
当修改参数 near，far 时，可以发现：

- 当 far 太小或 near 太大时，使得投影立方体没有将 cube 包含在内，在视图中就会看不见 cube。

透视投影：

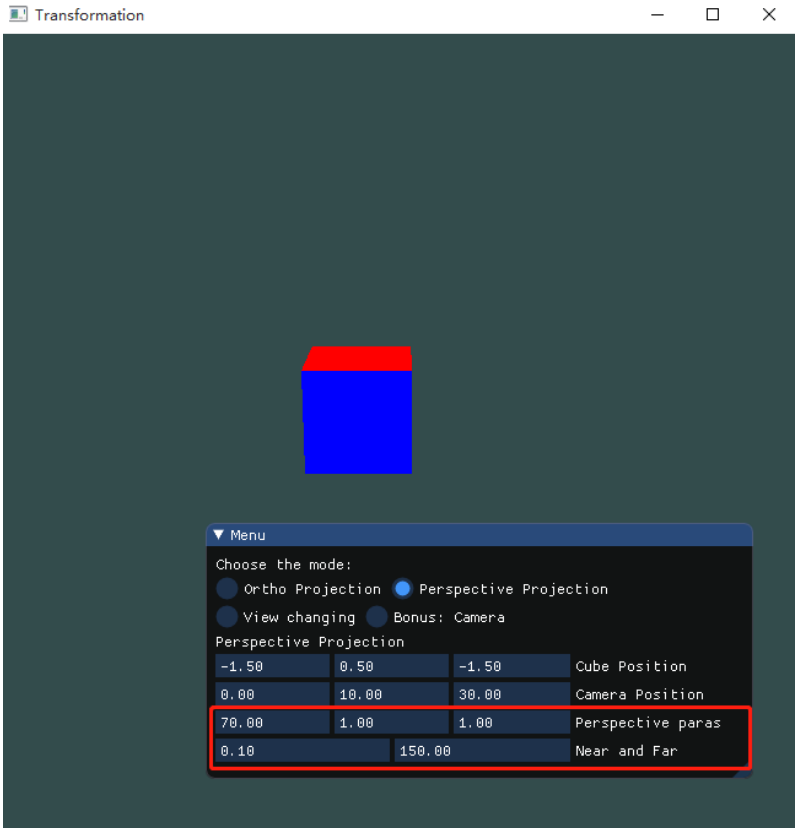
参数为：

Angle	Width	Height	Near	Far
1.00	1.00	1.00	0.10	150.0



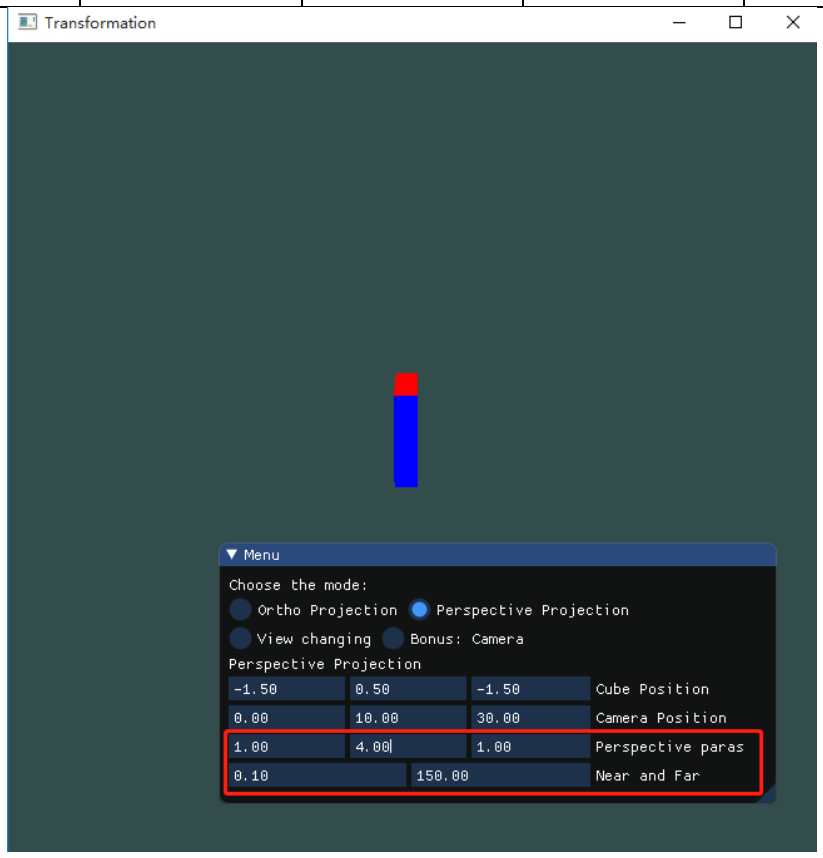
参数为:

Angle	Width	Height	Near	Far
70.00	1.00	1.00	0.10	150.0



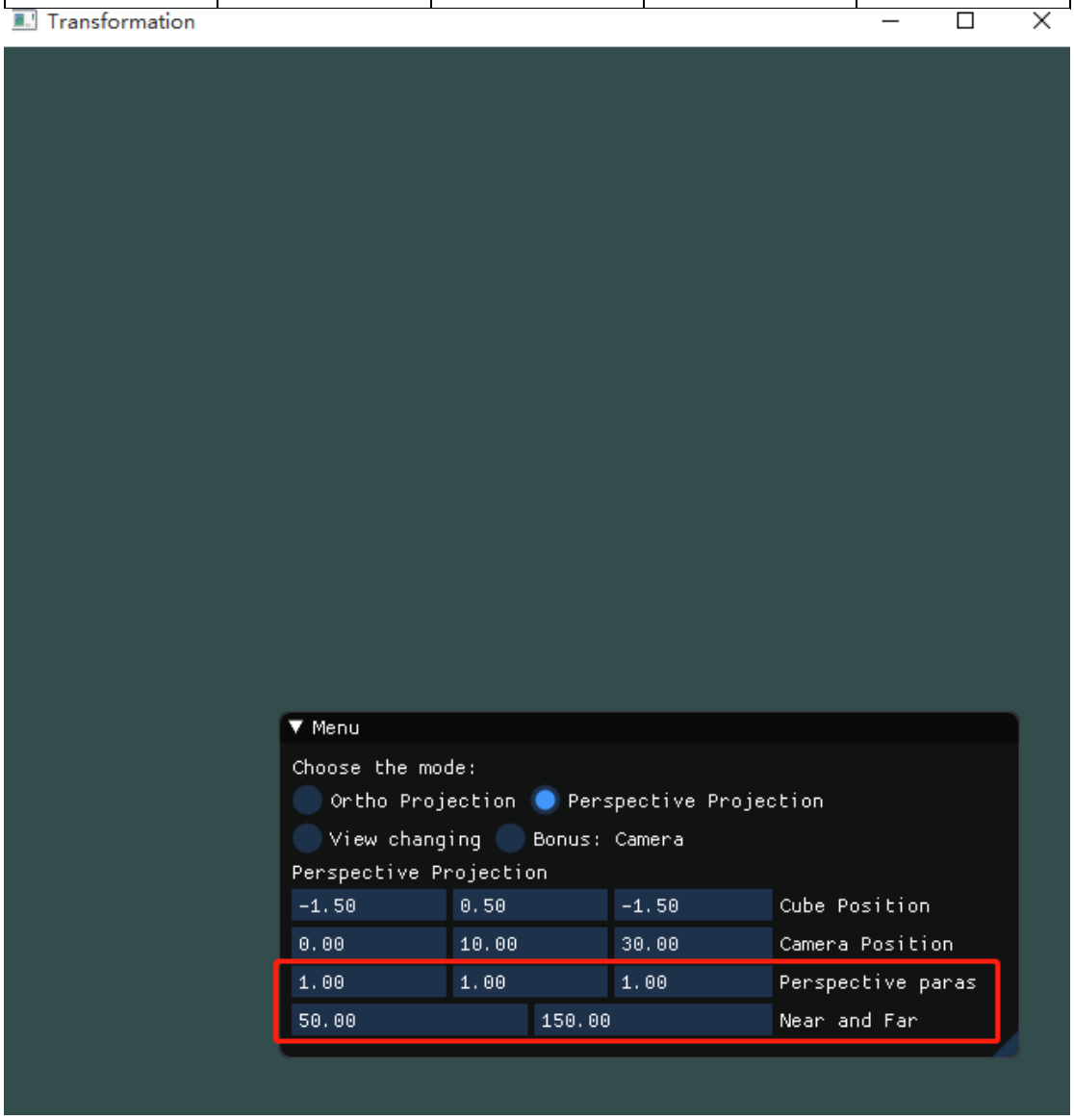
参数为：

Angle	Width	Height	Near	Far
1.00	4.00	1.00	0.10	150.0



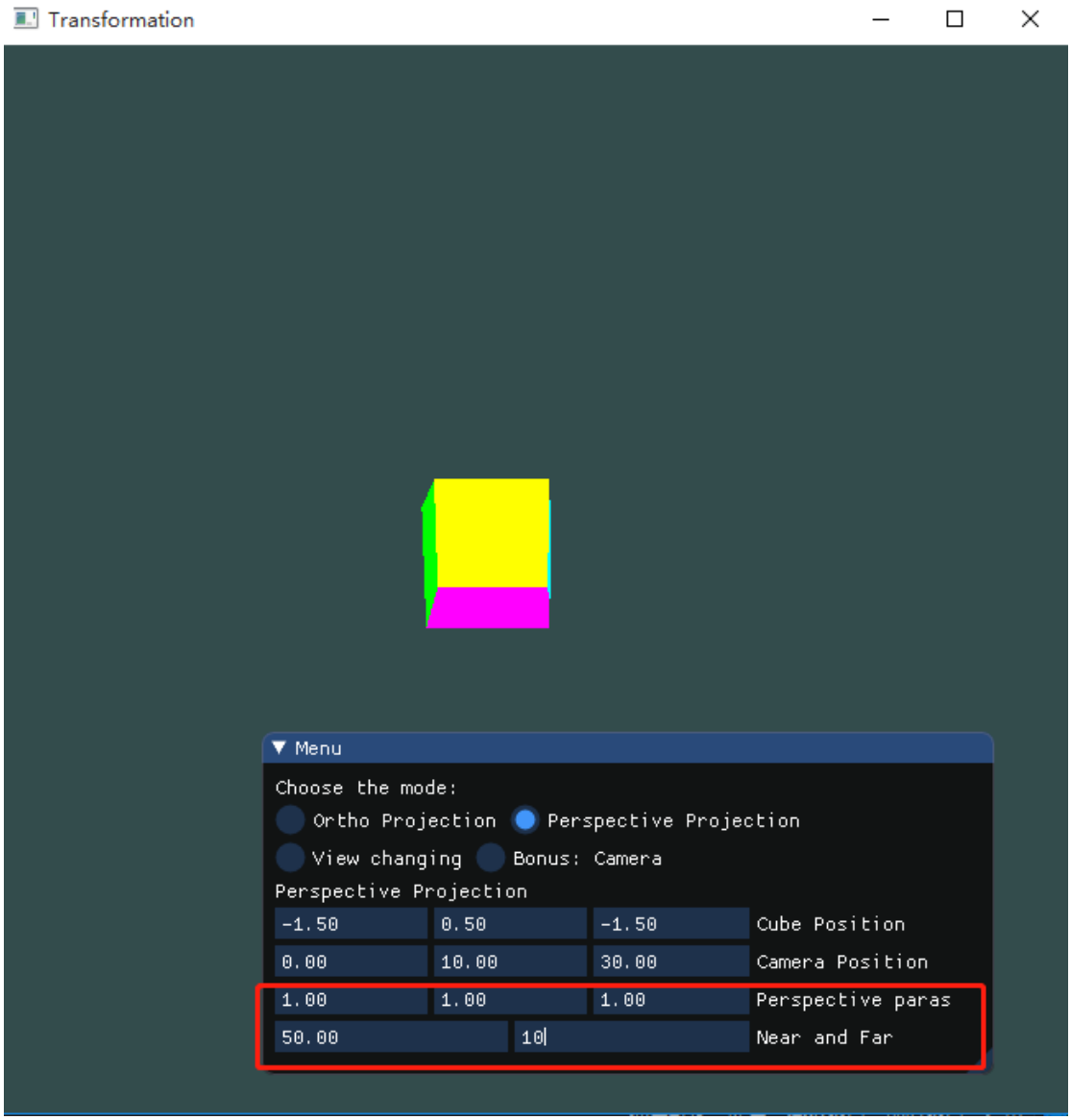
参数为：

Angle	Width	Height	Near	Far
1.00	1.00	1.00	50	150.0



参数为：

Angle	Width	Height	Near	Far
1.00	1.00	1.00	50	10



算法实现：

使用如下语句进行透视投影：

```
projection = glm::perspective(perspective[0], perspective[1] / perspective[2], eye_space[0], eye_space[1]);
```

在 GUI 里设计接口改变变量的值，然后将相应变量传进 model，view，projection 的矩阵即可。

参数比较：

- Width 和 height 不一样的话，cube 的长宽压缩比例不同，可以显示出一个长方体，同时长方体的位置也会因为映射范围的改变而改变
- Angle 增大，视图能看到的范围越大，cube 在视野中越小。Angle 理论上的范围为 [0， π]，若传入的 angle 超出该范围，与 angle mod π 作为值传入的效果相同。
- Near 和 far 同正交投影。但是，当 near > far 时，视图会变成从后面往前看。

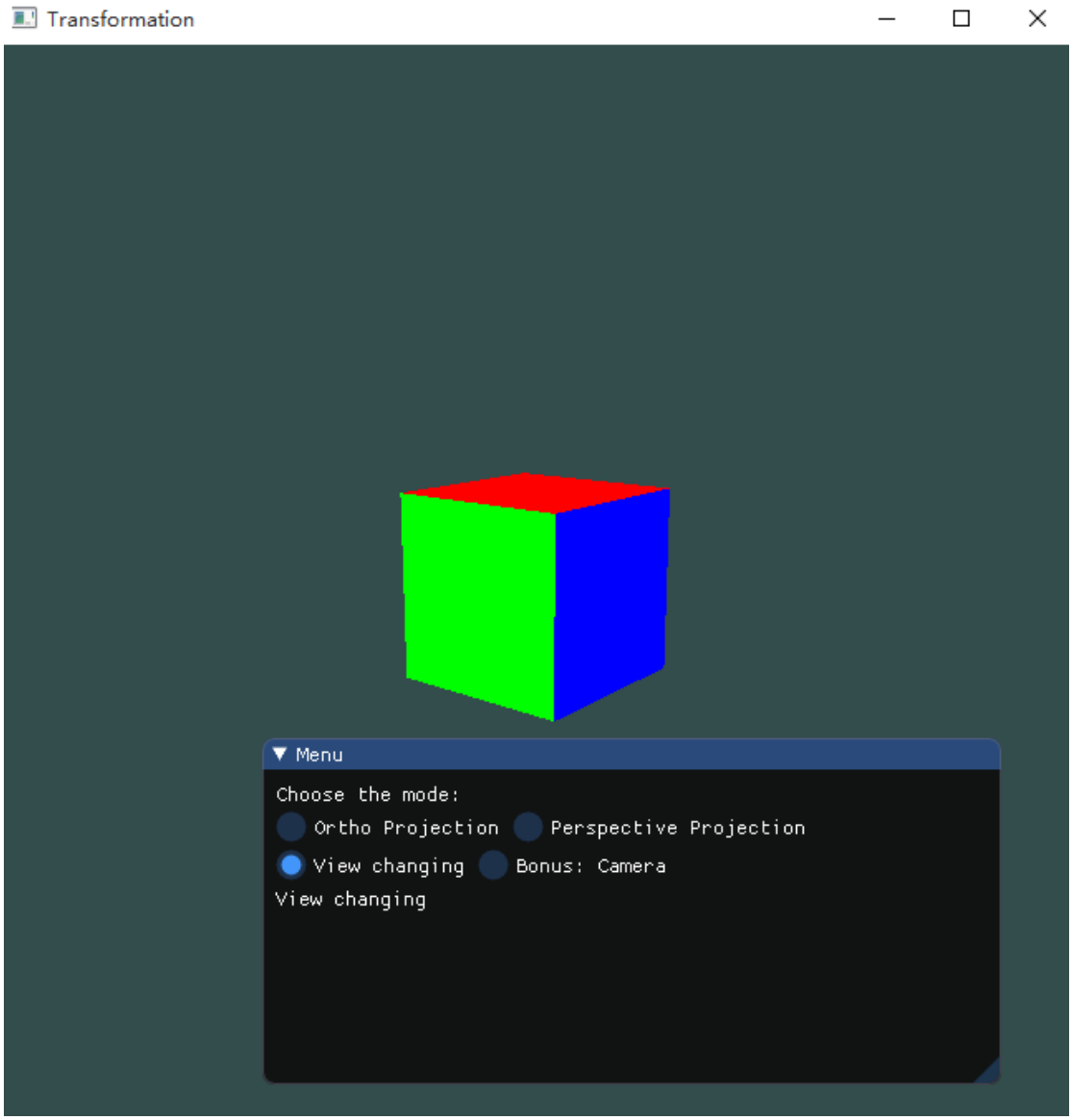
2. 视角变换（View Changing）：

- 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
- Hint：是摄像机一直处于一个圆的位置，可以参考以下公式：

```
camPosX = sin(clock()/1000.0) * Radius;
camPosZ = cos(clock()/1000.0) * Radius;
```

原理很容易理解，由于圆的公式 a^2+b^2=1，以及有 sin(x)^2+cos(x)^2=1，所以能保证摄像机在 XOZ 平面的一个圆上。

实验结果（具体结果见演示视频.mp4）



算法实现：

利用时间函数修改 camera 的位置，然后在每一帧的渲染中改变 camera 的 view 矩阵。

```
float Radius = 20.0f;
float camPosX = (float)sin(glfwGetTime()) * Radius;
float camPosZ = (float)cos(glfwGetTime()) * Radius;
```

```
view = glm::lookAt(glm::vec3(camPosX, 5.0f, camPosZ), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 5.0f, 0.0f));
projection = glm::perspective(45.0f, (float)WIN_WIDTH / (float)WIN_WIDTH, 0.1f, 100.0f);
```

3. 在 GUI 里添加菜单栏，可以选择各种功能。

使用 ImGui::RadioButton 函数来改变选择的功能：

```
ImGui::Begin("Menu");
ImGui::Text("Choose the mode:");
ImGui::RadioButton("Ortho Projection", &mode, 0);
ImGui::SameLine();
ImGui::RadioButton("Perspective Projection", &mode, 1);
ImGui::RadioButton("View changing", &mode, 2);
ImGui::SameLine();
ImGui::RadioButton("Bonus: Camera", &mode, 3);
```

在每个不同的功能中，利用 ImGui::InputFloat 函数来实现参数的更改：

```
//设置不同模式下的参数处理
switch (mode)
{
case 0:
    ImGui::Text("Ortho Projection");
    ImGui::InputFloat3("Cube Position", cubePosition, 2);
    ImGui::InputFloat3("Camera Position", cameraPos, 2);
    ImGui::InputFloat4("Othro paras", ortho, 2);
    ImGui::InputFloat2("Near and Far", eye_space, 2);
    break;
case 1:
    ImGui::Text("Perspective Projection");
    ImGui::InputFloat3("Cube Position", cubePosition, 2);
    ImGui::InputFloat3("Camera Position", cameraPos, 2);
    ImGui::InputFloat3("Perspective paras", perspective, 2);
    ImGui::InputFloat2("Near and Far", eye_space, 2);
    break;
case 2:
    ImGui::Text("View changing");
    break;
case 3:
    ImGui::Text("Using WASD to change view");
    isFirstFPS = true;
    firstMouse = true;
    break;;
default:
    break;
}
ImGui::End();
```

4. 在现实生活中，我们一般将摄像机摆放的空间 View matrix 和被拍摄的物体摆放的空间 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？在报告中写入。（Hints：你可能有不止一个摄像机）

对于每一个顶点，如果先做 model 变换在做 view 变换，则进行了两次矩阵预算，此时 n 个点需要的运算是 2n。如果先将 view 和 model 合并后再进行变换，则需要的矩阵运算次数为 n+1，而图形学中 n 一般很大，这时候运算开销是明显减少的。对于 projection，因为在 view 和 projection 之间有时候需要插入 lighting 操作，故不把 projection 也进行合并。

Bonus：

1. 实现一个 camera 类，当键入 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.

实验结果见演示视频.mp4

实现处理：

- 在 Camera 类中包含必要的摄像机成员变量，以及封装实现一些必要的操作方法进行按键实现移动镜头和鼠标移动镜头。使用 GetViewMatrix 方法作为接口生成当前摄像机的位置信息并返回一个 view 矩阵。
- 使用一个全局变量 deltaTime 来保存两帧之间的时间间隔，从而控制各部分时间的一致性。
- 通过处理输入的回调函数来监听键盘的输入事件。根据输入的按键实现对应操作，如监听到键盘输入 WASD 时改变摄像机的位置。

```
//处理键盘输入
void processInput(GLFWwindow* window) {
    //Esc输入
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, true);
    }
    //输入q时，退出Bonus模式
    if (glfwGetKey(window, GLFW_KEY_Q) == GLFW_PRESS) {
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
        isFirstFPS = false;
        mode = 0;
    }

    //WASD方向输入处理
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
        camera.moveForward(deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
        camera.moveLeft(deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
        camera.moveBack(deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
        camera.moveRight(deltaTime);
    }

}
```

- 通过回调函数监听鼠标输入事件。每次监听到时间后改变摄像机的位置，通过改变 xoffset 和 yoffset 来改变 pitch（俯仰角）和 yaw（偏航角）。

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
}
```