Project Report for CG 100433 course

Team member

高康瑞-1751900

汪 政-1753190

涂欣宇-1752952

王志达-1752572

陶研廷-1753555

Project Title

第一人称3D魔方游戏

Abstract

3d交互式游戏，计算机图形学，魔方游戏，模型

Motivation

在学习了一定的图形学知识以后，借助这次课程项目的机会，我们小组同学经过讨论，决定实现一个第一人称的3D魔方游戏。在这个游戏中，我们会使用一系列的图形学知识，比如坐标变换、光照、贴图等。为了使我们的游戏更加的真实，我们将尽可能做一系列的修缮。虽然实现起来有些难度，但我们愿意尽自己的努力完成它。

The Goal of the project

模型部分:

我们将会实现一个尽可能逼真的魔方，以及一双机械手

我们将会用天空盒的方法创造第一人称的环境，使玩家感觉像是在真实场景中玩游戏

我们将会在项目中添加材质、光照等部分，让我们的模型、场景更加真实

交互部分：

玩家可以用键盘操纵手的移动，使魔方旋转；通过鼠标以及特定按键来调整视角

我们将会尽可能从模型和场景的方面让我们的游戏更加真实

The Scope of the project

没有使用光线追踪的方法
没有使用第三人称的游戏

Involved CG techniques

3d建模

坐标变换

光照

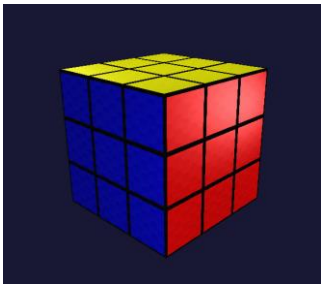
材质贴图

Project contents

建模部分：完成对魔方、人手以及对场景的建模
移动过程：魔方可以自由旋转，程序中，手也可以协调地运转
交互体验：玩家可以通过键盘控制手来旋转魔方的各个层，通过鼠标、键盘来调整视角。我们将尽可能使效果更加逼真。

Implementation

一、魔方实现：

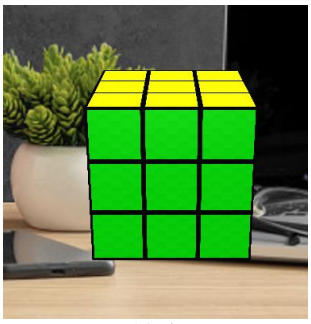


魔方由 27 个可以独立运动的小立方体组成，小立方体的每个面拥有不同的颜色和纹理。魔方应用了塑料材质的反光系数。照明则由 4 个点光源提供。每个小立方体的位置和方向信息都是单独储存的。

```
//方块位置变换矩阵。储存魔方的 27 个小方块的位置和朝向。
    glm::mat4 cubePosChanged[27]{
        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),
        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),
        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),

        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),
```

```
        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),
        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),

        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),
        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f),
        glm::mat4(1.0f),glm::mat4(1.0f),glm::mat4(1.0f)
    };
// 着色器
    Shader cubeShader = Shader("cubeShader.vs", "cubeShader.fs");

    // VBO 和 VAO
    unsigned int cubeVBO, cubeVAO;
// 四盏灯的位置
    glm::vec3 lightPos1 = glm::vec3(0.0f, 5.0f, 0.0f);
    glm::vec3 lightPos2 = glm::vec3(5.0f, 0.0f, 0.0f);
    glm::vec3 lightPos3 = glm::vec3(0.0f, 0.0f, 5.0f);
    glm::vec3 lightPos4 = glm::vec3(-5.0f, 3.0f, -5.0f);
```

在正常视角下，玩家可以观察到魔方的正面和上面。玩家可以左右侧身观察魔方的侧面，也可以直接左右旋转整个魔方，或上下翻转整个魔方来观察其他面。这些功能是通过视角偏移或旋转变换来实现的。



```
    // 翻转魔方
    bool overTurn(GLFWwindow* window, float angle);

    // 魔方整体右转
    bool rightTurn(GLFWwindow* window, float angle);

    // 魔方整体左转
    bool leftTurn(GLFWwindow* window, float angle);
```

玩家可以通过键盘上的 12 个按键实现对 6 个面的旋转顺时针或者逆时针旋转。在程序中实现时，需要根据不同的按键在筛选出需要旋转的那一面的 9 个方块以及旋转方向后进行旋转。

```
//旋转顶层,z==1
    bool z1Turn(GLFWwindow* window, float angle);
```

```
//旋转底层,z==-1
bool z_1Turn(GLFWwindow* window, float angle);

//旋转 x 轴靠近玩家的侧面，x==-1,"左面"。
bool x_1Turn(GLFWwindow* window, float angle);

//旋转 x 轴远离玩家的侧面，x==1,"左面的对面"。
bool x1Turn(GLFWwindow* window, float angle);

//旋转 y 轴靠近玩家的侧面，y==-1,"右面"。
bool y_1Turn(GLFWwindow* window, float angle);

//旋转 y 轴远离玩家的侧面，y==1,"右面的对面"。
bool y1Turn(GLFWwindow* window, float angle);、
```

按下自动复原键之后，魔方会自动复原到初始状态。这个功能由栈实现。

```
// 还原魔方的栈
    std::stack<int> stk;
```

魔方旋转时会有自然的加速和减速过程，模拟了真实的旋转过程。这是因为旋转速度是一个平滑的先增后减的函数。

```
if (timer < 0.5)
        rSpeed = deltaTime * (timer * timer) * 1080;
    else
        rSpeed = deltaTime * ((1 - timer) * (1 - timer)) * 1080;
```

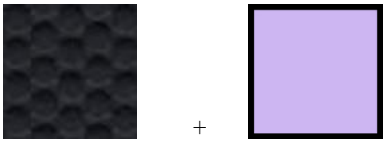
二、魔方的外观优化

我们在材质和纹理上，对魔方的外观进行了优化。
首先是材质上，我们选择了塑料的材质，其参数为：

```
cubeShader.setVec3("material.ambient",  rgb  *  glm::vec3(0.1f));
cubeShader.setVec3("material.diffuse",  rgb  *  glm::vec3(0.5f));
cubeShader.setVec3("material.specular",  0.55f,  0.55f,  0.55f);
cubeShader.setFloat("material.shininess",  16.0f);
```

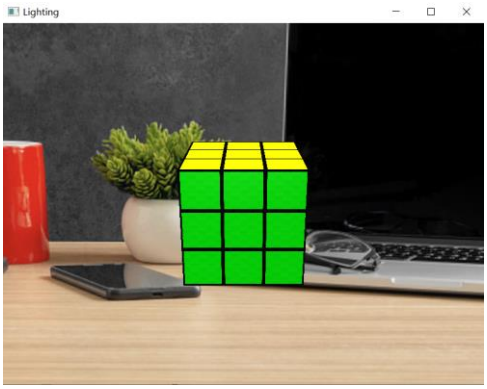
让魔方表面有了一定的反光效果

然后我们对魔方表面添加了纹理贴图，我们采用的是混合贴图的方法，用一张底色和一张纹理图，得到最终的效果：



三、室内场景天空盒的实现

天空盒是一个包含了整个场景的（大）立方体，它包含周围环境的 6 个图像，让玩家以为他处在一个比实际大得多的环境当中。游戏中使用天空盒的例子有群山、白云或星空。
但我们希望能实现一个室内的效果。我们尝试，在盒子内部贴上一个房间的 6 个面，让盒子实现房间的效果。经过不断尝试，我们发现，很难实现一个较为真实的效果。
后来我们改变了思路，利用一张桌面前的照片，作为背景，将魔方靠近 skybox 的边缘，然后调节摄像机的位置和透视参数，可以实现一个较为真实的效果：



四、摄像机控制

我们通过取 4x4 矩阵左上角的 3x3 矩阵来移除变换矩阵的位移部分。我们可以将观察矩阵转换为 3x3 矩阵（移除位移），再将其转换回 4x4 矩阵的方法，将摄像机固定在天空盒的一个固定的位置：

```
glm::mat3  newView  =  glm::mat3(camera.GetViewMatrix());
view  =  glm::mat4(newView);
view  =  glm::translate(view,  glm::vec3(0.0f,  0.0f,  -2.0f));
```

这样可以实现我们在移动视角的时候，背景的变化不会过于剧烈，增加了场景真实感

其次，我们增加了 2 个“侧身”按键，让玩家可以侧向移动，观察模仿的左右 2 个面：

```
if(direction  ==  Q)
{
```

```
3.          if(Yaw<-83.4f)
4.          {
5.                  Position  -=  Right  *  velocity;
6.                  Yaw  +=  velocity  *  rate;
7.          }
8.  }
9.  if(direction  ==  E)
10.   {
11.         if(Yaw>-96.7f)
12.         {
13.                 Position  +=  Right  *  velocity;
14.                 Yaw  -=  velocity  *  rate;
15.         }
16. }
17.
18. if(direction  ==  NONE)
19. {
20.         if(Yaw>-89.5f)
21.         {
22.                 Position  +=  Right  *  velocity;
23.                 Yaw  -=  velocity  *  rate;
24.         }
25.         else  if(Yaw<-90.5f)
26.         {
27.                 Position  -=  Right  *  velocity;
28.                 Yaw  +=  velocity  *  rate;
29.         }
30.         else
31.         {
32.                 Yaw  =  -90.0f;
33.         }
34. }
```

五、手部实现

1. 手有关类的封装与功能

1.1. 指节

1.1.1. 类功能

组成手的最小单位，进行绘制的底层实现部分，无需考虑宏观的运动，只需根据上层传入的信息，进行平移旋转等操作，最终在正确的位置绘制出该最小单位形态。

1.1.2. 重要成员变量

Finger_type f_type------所属手指的类型；

Dactylus_type d_type------指节本身的类型；

glm::vec3 d_position------该指节所在的位置；

glm::mat4 rotate_hudu------该指节在空间中三个轴旋转的弧度；

glm::vec3 d_model------该指节传入 shader 进行绘画的模型空间信息；

1.1.3. 重要成员函数

bool trans_a_dactylus(glm::vec3 pos);

对指节信息进行平移，在指节自己的坐标系中，平移到传入的 pos 位置；

bool rotate_a_dactylus(float angle, glm::vec3 axis);

对指节信息进行旋转，在指节自己的坐标系中，按照传入 axis 轴，顺时针旋转 angle 弧度。

bool fixed_the_anis(glm::vec3 jiaodu);

根据所传入的角度（针对三个维度），将指节旋转固定到那个轴上去，主要是在指节最终位置形成的时候，第一次对指节的角度进行校正，校正到与手指轴向相同的坐标系中。

bool draw_a_dactylus(Shader* s);

在一系列操作后，成员变量 d_model 已经存储了指节所在位置和角度的最终信息，再调用本函

数即可对该指节进行正确的绘制。

1.2. 手指

1.2.1. 类功能

组成手的主要单位，用于接收手类的调用，对一个手指的三个指节进行调配，完成手指的位移，旋转，以及弯曲等操作。同时存储了手指当前的状态(针对于手掌的相对位置角度)，并且能控制手指运动的弧度范围以及速度。

1.2.2. 重要成员变量

Finger_type type------手指的类型
Dactylus part_root------手指连接手掌的指节
Dactylus part_middle------手指中间的指节
Dactylus part_top------手指最顶层的指节
glm::vec3 f_position------手指相对手掌的位置
glm::vec3 hf_position------手掌的位置
float part_len[3]------三个手指的长度
float f_phi------手指的弯曲弧度
float alpha------手指的弯曲角度
bool flag_up------ 允许手指伸直
bool flag_down------允许手指收缩
float speed_coe------加速系数

1.2.3. 重要成员函数

- bool draw_a_finger(Shader* s);
  针对一个手指，调用其三个指节的函数 draw_a_dactylus（），完成一个手指的绘制，当然前提是已经经过操作将手指的各项数据调整至目标位置及角度。
- bool form_a_finger(glm::vec3 rotate_jiaodu, int which_hand,
  glm::vec3 another_movement = glm::vec3(0));
  手指形成最主要的函数，用来根据已有信息在指定位置生成一个初始化的手指。
  针对三个指节依次计算，首先是底部指节，先计算出手指相对于手的相对位置的角度，然后得到在全局坐标中该手指的位置和角度，再在手指坐标系中移动底部指节；之后是中间指节，在之前计算出来的底部指节坐标系中相对于底部指节进行移动，按照类中所保存的该指节的弯曲程度，相对于底部指节进行旋转，这样结构清晰，方便思考，可以将两个指节完美的契合在一起；最后按照之前的规则对顶部指节进行绘制，主要参照的是中间指节的位置及角度。
- bool move_a_finger_with_an_angle(float h_angle, float l_angle);
  用于给上层调用，对手指的弯曲程度按照自己设置的步长和加速因子进行变化，多次调用即可形成手指弯曲伸直的动态小伙，可以自己移动角度的上下限，h_angle 是弯曲程度上限，l_angle 是弯曲程度下限。
- void change_direction(bool dir);
  用于在手指要更改移动状态时对移动状态进行更改，move_a_finger_with_an_angle 函数只会按照类中已有的方向进行移动和判断，并不会自己进行移动状态的改变，所以要靠本函数对手指的移动状态进行设置。（dir 为真时状态是伸展手指，为假时状态是收缩手指）

1.3. 手

1.3.1. 类功能

最上层的类，完成了手的宏观建立、形成、移动、进行特地动作等多项功能。基本不会对指节类进行直接的操作，主要操作是针对自己的五个手指头成员，对手指成员的相应状态进行设置，已达到整个手的协调绘制。

1.3.2. 重要成员变量

int hand_type------手的类型，1 是左手，-1 是右手，因为手是镜面对称所以不能完全不加以区分。其次设置成 1 和-1 是为了在控制角度时免于对手的类型进行判断，直接乘以手的类型就可以达到对两只手镜面角度进行区别。
Finger f_dmz------大拇指成员
Finger f_sz------食指成员
Finger f_zz------中指成员

Finger f_wmz------无名指成员
Finger f_xmz------小拇指成员
Dactylus plate------手掌成员，因为手掌基本上没有什么动作，这里直接按照指节的规则进行调用。
glm::vec3 h_position------手的位置
glm::vec3 h_angle------手的旋转角度
bool in_a_action------ 动作互斥的信号，如果该信号被置为真，则屏蔽其他动作的信号，目的是不允许多个同时执行
int state------调控手的动作所设置的动作状态机的状态

1.3.3. 重要成员函数

bool draw_a_hand(Shader* s, float deltaTime);
与之前的相同，宏观调用各种低层函数，完成整个手的绘制。
bool form_a_hand();
根据已有信息在指定位置生成一个初始化的手，也是通过调用成员的底层函数实现。
bool check_flag(float deltaTime);
检查标志位，是否有动作时间被触发，有则进入相应动作的状态机。
bool start_action();
开始游戏，进行准备动作
bool end_action();
动作结束，如果魔方被打乱则对魔方进行复原
bool sz_play();
底层动作，食指拨动魔方，供上一级动作函数的状态机在某一状态进行调用
bool wmz_play();
底层动作，无名指拨动魔方，供上一级动作函数的状态机在某一状态进行调用
bool hold_rubiks_cube();
底层动作，抓住魔方，供上一级动作函数的状态机在某一状态进行调用
bool release_rubiks_cube();
底层动作，松开魔方，供上一级动作函数的状态机在某一状态进行调用
bool hand_rotate_c(float down_res);
顶层动作，手竖向顺时针转动
bool hand_rotate_a(float up_res);
顶层动作，手竖向逆时针转动
bool hand_dial_c(float down_res);
顶层动作，手横向顺时针转动
bool hand_dial_a(float up_res);
顶层动作，手横向逆时针转动
bool action_move_uc();
顶层动作，上面-顺时针
bool action_move_ua();
顶层动作，上面-逆时针
bool action_move_dc();
顶层动作，下面-顺时针
bool action_move_da();
顶层动作，下面-逆时针
bool action_move_fc();
顶层动作，前面-顺时针
bool action_move_fa();
顶层动作，前面-逆时针
bool action_move_bc();
顶层动作，后面-顺时针
bool action_move_ba();

顶层动作，后面-逆时针
```
bool action_move_lc();
```
顶层动作，左面-顺时针
```
bool action_move_la();
```
顶层动作，左面-逆时针
```
bool action_move_rc();
```
顶层动作，右面-顺时针
```
bool action_move_ra();
```
顶层动作，右面-逆时针
```
bool action_all_up();
```
顶层动作，向上转
```
bool action_all_down();
```
顶层动作，向下转
```
bool action_all_left();
```
顶层动作，向左转
```
bool action_all_right();
```
顶层动作，向右转

2. 模拟手的真实比例，加入金属风格贴图

2.1. 关于手的坐标变换

- $p^n$is the world coordinate of point p after n transformations

$$p^n = M_1 M_2 \cdots M_n p$$

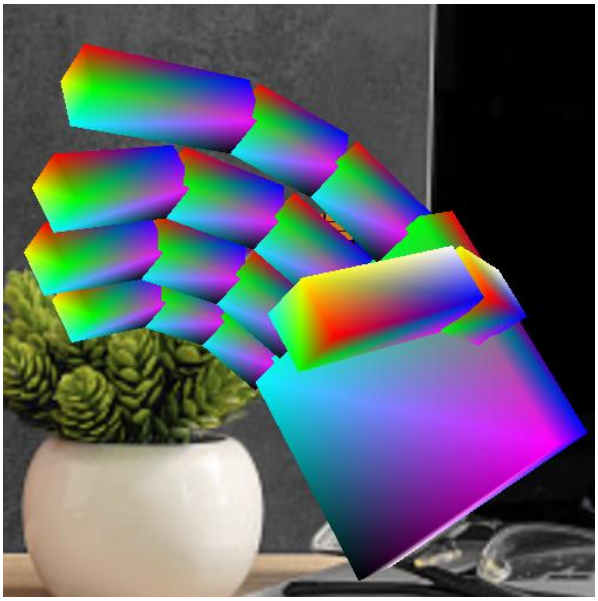
2.2. 通过静态全局变量，控制手指长度与比例

设置每个手指长度

设置每个手指宽度

```
//手指宽度单位
const float finger_relative_pos[15]{
    -1.0f* oppo_size,-3.5f* oppo_size,  5.0f* oppo_size,//大
     0.0f* oppo_size,  0.0f* oppo_size,  3.0f* oppo_size,//食
     0.0f* oppo_size,  0.0f* oppo_size,  0.0f* oppo_size,//中
     0.0f* oppo_size,  0.0f* oppo_size,-3.0f* oppo_size,//无
     0.0f* oppo_size,  0.0f* oppo_size,-5.0f* oppo_size //小
};
```

- 效果



2.3. 加入贴图

3. 动作封装与配合

3.1. 不断调参使手与模仿完美结合

在封装类的时候想到了手与魔方结合会出现不协调的问题，所以设置了很多借口和可以更改的全局静态函数调用，在结合的时候只需要对参数进行调整即可。

3.2. 通过标志位控制动作

每个状态机函数返回值都是 bool 型，正常返回为 false，表示状态机仍在运行，未达到终止状态，如果返回了 true，那么状态机一定是异常退出或者达到目标盘状态了，且上一层的检测函数可能在高一级的状态机里，上一级的状态机在检测到子状态机完成了以后，则会进行状态的改变，在顶层调用状态机也会进行标志位检测，如果发现状态机进入终止状态，则恢复相应标志位，表明此次调用动作在本帧结束。

3.3. 一旦完成退出，则不再进入动作

也就是一但检测到标志位为真，告知调用状态结束，怎会改变调用逻辑，恢复标志位，以保证程序正常运行。

3.4. 通过静态全局变量，控制重要动作幅度

动作的幅度不可能每次在调用的时候临时设置动作幅度，还是要通过静态全局变量提前设置，将动作幅度确定，方便调用也方便更改调试。

3.5. 在指定状态发送信号，控制魔方转动

因为要与魔方进行配合，所以在进行拨动动作之前的状态，在转换到拨动之前也要向魔方发送相应的信号。

4. 复原状态机

4.1. 入栈

每次调用动作函数，进入状态机，在终止状态后进行动作入栈处理。

4.2. 出栈

在进行复原时，先检测栈是否为空，若不为空，则将顶层动作读出，并且进行复原。

4.3. 检查

出栈状态后进入等待状态，每次的等待状态都要进行动作是否完成的检查，若完成则进入下一状态。

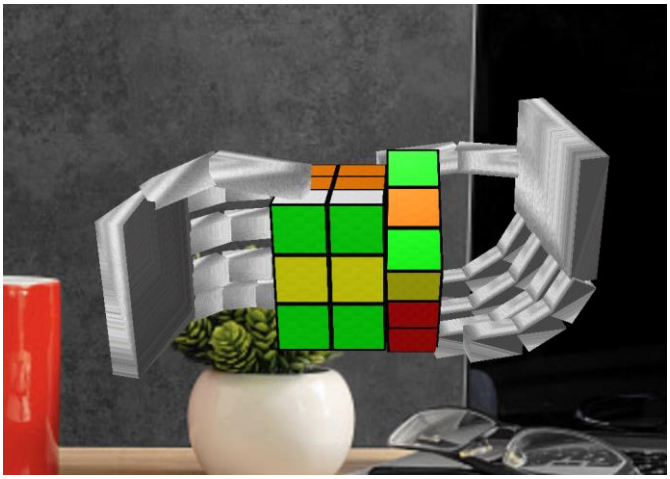
4.4. 等待

等待状态即等待动作进行，当动作进行完则返回判断是否有下一个动作需要进行复原。

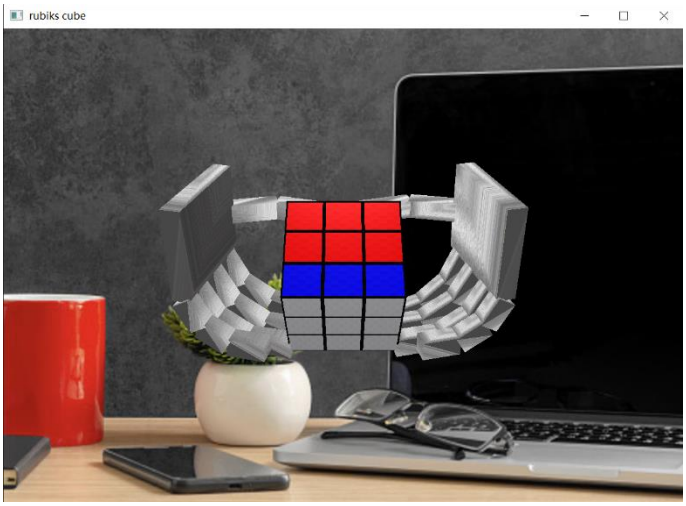
Results

1.手自由移动

2.12 个拨动魔方的动作



3.4 个转动魔方的动作



4. 自动还原
见演示 gif 或者演示视频

六、3d 模型导入

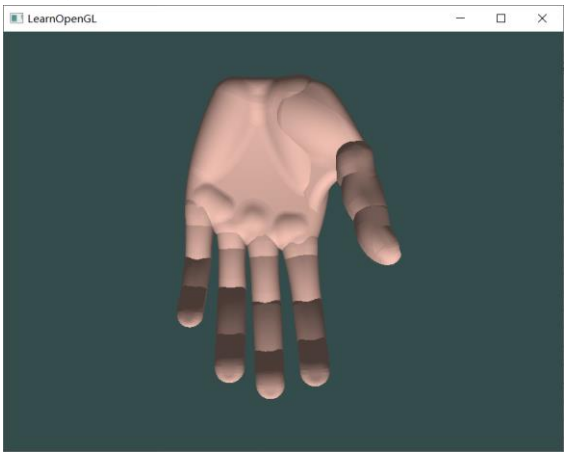
1.通过 3dsmax 软件打开模型，将其导出为.obj 文件

2. 通过 deep exploration 处理模型的 obj 文件得到的 hand_origin.cpp 文件

3. 编写 C++程序 input_data.cpp 读取 hand_origin.cpp 中保存顶点属性的数组，生成可以直接被 opengl 的 shader 直接解析的数组

4. 编写 opengl 程序，设置光照、颜色、动作，展示人手模型

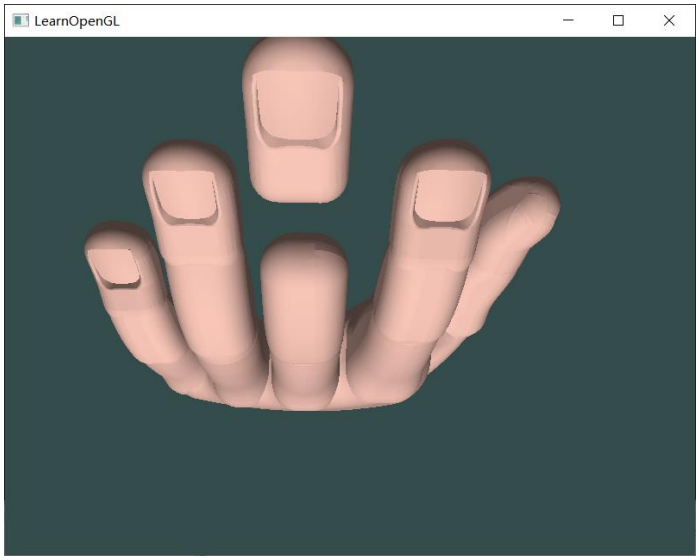
导入模型后的结果：



通过上面的方法，我们可以导入人手模型的基本属性。然而.obj 文件并不支持 3ds MAX 软件中骨骼的导出，因此我们导出的模型中并没有骨骼的信息。

但是由于我们选择的模型将手掌、各个手指划分为不同的对象并且分别建模，最终将这些组件拼接形成完整的人手。在模型导出的.obj 文件中，手指的各个关节和手掌等部位的顶点信息是分开存储的，因此我们可以通过索引来判断不同坐标归属的不同部位，通过整体操纵一个部位的平移、旋转，我们可以制作各种人手动画。

如下图，我们可以单独操控中指的第三关节进行平移运动。这表明了我们可以分别构造人手不同部位的空间变换矩阵来制作人手的动画。但由于时间紧张以及人力短缺（小组成员共计 5 人），要定位各个手指关节的旋转轴需要大量精力（一只手需要定位 14 个旋转轴的），我们并没有实现真实人手模型操纵魔方的动画效果。



不导入骨骼，我们可以通过上述的方式来实现人手模型的导入和动画制作。除此之外，我们研究了骨骼数据导入 opengl 的方法。编写了如下的骨骼类。通过这样的类，我们可以将骨骼数据与顶点坐标联系起来，实现真正的骨骼动画。其中 face_indice 是存放面的数组，由对构成一个三角面的三个顶点的索引组成。

七、骨骼动画理论研究

为了实现更加真实的人手动画，我们对如何从三维建模软件向 opengl 导入三维模型和制作三维模型动画进行了相关的学习研究，并实现了人手模型的导入和简单人手动画。

模型动画就是三维计算机动画，它利用计算机构造三维形体的模型，并通过对模型、虚拟摄像机、虚拟光源运动的控制描述，由计算机自动产生一系列具有真实感的连续动态图像。

目前 3D 游戏制作中动画的实现主要有两种方式：帧动画和骨骼动画。针对化顾名思义，就是在三维建模软件中对一个物体对象进行位移、旋转、缩放、变形，然后把关键帧的信息记录下来，在播放的时候按照关键帧时间对物体对象进行位移、旋转、缩放、变形，并在关键帧和关键帧之间做插值运算。

骨骼动画的特点是，需要做动画的物体对象本身不记录位移、旋转、缩放、变形信息，而是通过第三方的“骨骼”物体记录动画信息，然后物体对象本身只记录受骨骼物体影响的权重。在播放的时候，通过骨骼物体的关键帧和物体对象记录的权重，让动画重现。

在骨骼动画中我们记录的关键帧并不是物体本身的信息，而是物体的不同组成部分依附的不同骨骼的信息。所谓的骨骼本质上就是一个依附于它的顶点进行空间变换的矩阵。在不同关键帧记录每一个骨骼的位置变换的矩阵。在动画实现的过程中，我们在每个关键帧依照骨骼的变换矩阵和影响不同顶点的权重对组成物体的不同顶点的空间坐标进行变换，在关键帧中间通过不同插值算法生成关键帧中间的动画。

相较于帧动画，骨骼动画是较为先进的计术，有如下优点：

1、骨骼动画是影响到顶点级别的动画，而且可以多根骨骼根据权重影响同一个顶点，不论是 2D 或者 3D 使用，都可以让动画做得更丰富。

2、做到了对象和动画分离，我们只需要记录了物体对于骨骼的蒙皮权重，就可以单独的去制作骨骼的动画，在保证蒙皮信息和骨骼信息一致的情况下，还可以多个物体之间共享动画。

3、相对于 2D 的逐帧动画大大的节省资源容量（我们只需要记录在不同的关键帧骨骼的变换矩阵和影响顶点的权重）。

4、骨骼动画能减少贴图体积，减少美术工作量。

条件允许的情况下，我们可以在后续版本中，实现相应效果

Results

经过小组成员的努力，我们成功实现了以一个尽可能逼真的魔方，一双机械手，以及书桌为背景组成的第一人称3D魔方游戏。在过程中我们充分使用了图形学的知识，顺利地建模、实现运动过程，优化了游戏体验。经过这次课程项目的锻炼，小组同学都收获颇丰，希望在以后的学习中，我们可以取得更大进步。

Roles in group

高康瑞：负责模型动画理论研究、3d 模型导入 OpenGL
涂欣宇：负责魔方绘制及其功能实现
汪政：负责天空盒实现、魔方优化、魔方与手的结合
陶研廷：负责手部全部模型实现，魔方与手的结合
王志达：手部模型优化、魔方与手的结合、材料撰写整理

References

learnOpengl教程

https://learnopengl-cn.github.io/