计算机图形学期末大作业-小组报告

18340012 陈晨 18340056 胡邱诗雨 18340195 杨智博

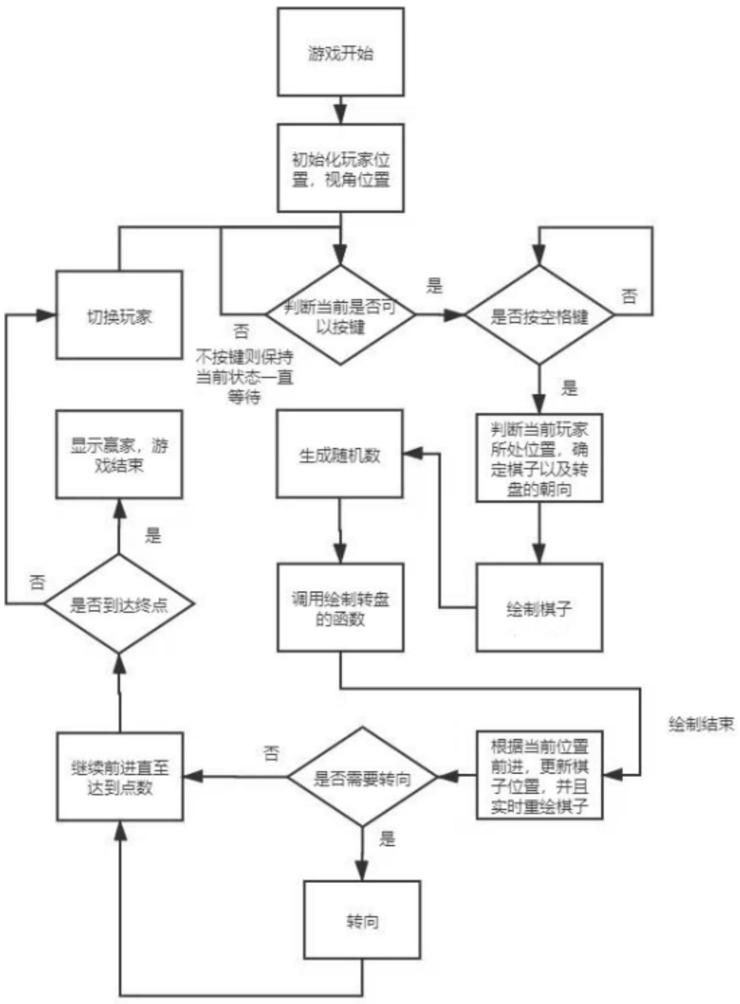
1 作业内容

- 实现一个第三人称跟随视角的回合制棋盘游戏，每次按键得到随机点数，相应玩家前进对应点数的格子，先到达终点的玩家获胜。
- 使用了openGL，通过变换、纹理、光照和视角变化等手段完成所需效果的实现。

2 思路与代码实现

2.1 游戏逻辑

2.1.1 整体逻辑



2.1.2 视角跟随

2.1.2.1 初始化

利用`initialGame()`给棋子和camera进行初始化，并且记录下当前的位置,以及标记终点的位置，以便最后的终点判断

```cpp
void MyGLWidget::initialGame()
{
//初始化玩家位置
    for (int i = 0; i < 3; i++)
    {
        playerPos[i][0] = -1.0;
        playerPos[i][1] = 1.5;
        playerPos[i][2] = -7.0;
    }
//初始化camera的位置
    for (int i = 0; i < 3; i++)
    {
        camera_x[i] = playerPos[i][0] + 3;
        camera_z[i] = playerPos[i][2];
        camera_x_at[i] = playerPos[i][0];
        camera_z_at[i] = playerPos[i][2];
    }
//初始化结束的位置
    endPos[0] = 1.0;
    endPos[2] = -7.0;
    endPos[1] = 1.5;

}
```

2.1.2.2 前进

设置了一个全局变量分别记录玩家1和玩家2的位置，以及camera位置的记录。每次根据玩家前进，对camera的位置进行对应的调整，保证camera始终保持在玩家后3格，并且聚焦在玩家身上。每次切换玩家即会切换视角，保证camera始终聚焦在当前玩家的身上。并且`count_step`变量记录目前的步数是否达到了转盘出现的点数

```cpp
if (direction == 1)
{
    camera_x[i] -= 0.2;
    playerPos[i][0] -= 0.2;
    camera_x_at[i] = playerPos[i][0];
    count_step += 0.2;
}
...
//后面省略其他方向的操作
```

2.1.2.3 转向

在每次更改位置前，先判断当前位置是否到达转角处，若需要转角，则先原地等待camera转换方向，然后再继续前进

```cpp
else if (direction == 2)
{
//若此处为转角处
    if ((abs(playerPos[i][0] - (-7.0)) < 0.0000001 && camera_x[i] >
-6.9 && abs(playerPos[i][2] + 7.0) < 0.0000001))
```

```
    {
    //camera开始转方向
        camera_x[i] -= 0.6;
        camera_z[i] -= 0.6;
    }
    else
    {//若不是转角处，则继续前进
        camera_z[i] += 0.2;
        playerPos[i][2] += 0.2;
        camera_z_at[i] = playerPos[i][2];
        count_step += 0.2;
    }
}
```

2.1.2.4 视角切换

当`count_step`计数满转盘点数时，即切换到下一个玩家，将全局变量`player_now`记录为另一个玩家，实现视角的切换.

```
if(count_step < turntable_number)
{
    ...
}
else
{
    count_step = 0.0;
    press_flag = true;
    is_moving = false;
    step[player_now - 1] = 0;
    if (i == 1)
    {
        player_now = 2;
    }
    else
    {
        player_now = 1;
    }
}
```

2.1.3 动画调用控制

2.1.3.1 棋子

棋子的绘制以及跳跃由陈晨实现，我负责调用其函数来完成棋子的前进以及转向。在转盘动作结束前，以及键盘无操作时，棋子应当保持当前状态原地等待。当前的前进方向，以及存储的实时棋子位置进行绘制。并且由`is_moving`变量来控制当前棋子是否进行棋子跳跃的动作。

```
if (direction == 1 || direction == 5)
{
    obj_dir = 1;
    //绘制1号玩家

    move_p(1, obj_dir, playerPos[1][0], playerPos[1][2], (!press_flag
&& player_now == 1 && is_moving));
    //绘制2号玩家
    move_p(2, obj_dir, playerPos[2][0], playerPos[2][2], (!press_flag
&& player_now == 2 && is_moving));
}
else
{
    obj_dir = direction;
    move_p(1, obj_dir, playerPos[1][0], playerPos[1][2], (!press_flag
&& player_now == 1 && is_moving));
    move_p(2, obj_dir, playerPos[2][0], playerPos[2][2], (!press_flag
&& player_now == 2 && is_moving));
}
```

2.1.3.2 转盘

转盘的绘制以及旋转由杨智博实现，我负责转盘位置的确定，以及生成随机数传递给转盘，并且控制转盘的绘制时间。

首先由一个读键盘的函数来得到是否玩家按下了空格键来换取随机点数。并且在读取到空格键后将布尔值变量`ISDraw_turntable`置为真值，这样表示可以绘制转盘。

```
void MyGLWidget::keyPressEvent(QKeyEvent* e) {
    if (press_flag == true)
    {
        if (e->key() == Qt::Key_Space) {
            IsDraw_turntable = true;
            press_flag = false;
            is_running = true;
            number_flag = true;
        }
    }

}
```

在Game函数里读取到绘制转盘，并且在`is_running`为false时，结束转盘绘制，将`ISDraw_turntable`置为false;

```
if (number_flag)
{
//生成随机数
srand(time(NULL));
turntable_number = rand() % 6 + 1;
number_flag = false;
}
if (IsDraw_turntable == true)
{

//转盘
draw_plate(turntable_number, 0.2 * playerPos[i][0] + 0.8 * 
camera_x[i], 1.2f, 0.2 * playerPos[i][2] + 0.8 * camera_z[i], 
obj_dir);
//转结束了
if (is_running == false)
{
    plate_angle = 30.0f;
    IsDraw_turntable = false;
    is_running = true;
    is_moving = true;
}

}
```

2.1.3.3 闸机打开

根据当前点的位置，当棋子到达相应的位置后，即可将对应的flag置为true,即可开门

```
if (abs(playerPos[player_now][2] - 7.0) < 1e-6 && 
playerPos[player_now][0] > -3.0 && close_flag2 == true)
{
    openning_flag2 = true;
    close_flag2 = false;
}
if (abs(playerPos[player_now][2] - 7.0) < 1e-6 && 
playerPos[player_now][0] > -0.5 && close_flag1 == true)
{
    openning_flag1 = true;
    close_flag1 = false;
}
```

2.2 场景

2.2.1 着色器

在本次作业中，所有物体的绘制都基于VBO，VAO，EBO和自定义着色器实现。着色器主要要实现三个功能：纹理贴图，光照，不透明度设置。因此，VBO中需要存储三种属性：顶点坐标（3位），法向量（3位），2D纹理坐标（2位）。

基于此，实现着色器如下：

顶点着色器中，需要通过传入的modelview矩阵和projection矩阵计算顶点的位置、计算标准化的法向量传入片段着色器，并且处理纹理坐标，传入片段着色器。

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

out vec3 frag_pos;
out vec3 frag_normal;
out vec2 TexCoord;
```

```

uniform mat4 modelview;
uniform mat4 projection;

void main()
{
    gl_Position = projection * modelview * vec4(aPos, 1.0f);
    frag_pos = vec3(modelview * vec4(aPos, 1.0f));
    frag_normal = normalize(mat3(transpose(inverse(modelview))) *
aNormal);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

片段着色器中，需要传入纹理，不透明度，视角位置，分别用于计算纹理对应的片段颜色，不透明度和镜面反射。这里采用了冯氏光照，最终将光照的计算结果与通过纹理坐标得到的结果相乘，并且设置不透明度（alpha）。

```
#version 330 core
out vec4 FragColor;

in vec3 frag_pos;
in vec3 frag_normal;
in vec2 TexCoord;

uniform sampler2D ourTexture;
uniform float alpha;
uniform vec3 view_pos;

void main()
{
    vec3 light_color = vec3(1.0f, 1.0f, 1.0f);
    vec3 light_pos = vec3(15.0f, 5.0f, -2.5f);

    float ambient_rate = 0.5f;
    vec3 ambient = ambient_rate * light_color;

    float diffuse_rate = 1.2f;
    vec3 light_dir = normalize(light_pos - frag_pos);
    float diff = max(dot(frag_normal, light_dir), 0.0f);
    vec3 diffuse = diffuse_rate * diff * light_color;

    float specular_rate = 0.2f;
    vec3 view_dir = normalize(view_pos - frag_pos);
    vec3 reflect_dir = reflect(-light_dir, frag_normal);
    float spec = pow(max(dot(view_dir, reflect_dir), 0.0f), 8);
    vec3 specular = specular_rate * spec * light_color;

    vec3 result = ambient + diffuse + specular;
    FragColor = texture(ourTexture, TexCoord);
    FragColor = FragColor * vec4(result, alpha);
}
```

着色器的初始化如下所示。从文件中读取着色器，并且进行创建、编译、链接，最终得到着色器程序。

```cpp
void MyGLWidget::init_shaders(const char* v_path, const char* f_path) {
    //读取着色器
    string v_code, f_code;
    ifstream v_shader_file, f_shader_file;
    v_shader_file.open(v_path);
    f_shader_file.open(f_path);
    stringstream v_shader_stream, f_shader_stream;
    v_shader_stream << v_shader_file.rdbuf();
    f_shader_stream << f_shader_file.rdbuf();
    v_shader_file.close();
    f_shader_file.close();
    v_code = v_shader_stream.str();
    f_code = f_shader_stream.str();
    const char* v_shader_source = v_code.c_str();
    const char* f_shader_source = f_code.c_str();

    // 顶点着色器
    GLuint v_shader;
    v_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(v_shader, 1, &v_shader_source, NULL);
    glCompileShader(v_shader);

    //片段着色器
    GLuint f_shader;
    f_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(f_shader, 1, &f_shader_source, NULL);
    glCompileShader(f_shader);

    //着色器程序
    shader_program = glCreateProgram();
    glAttachShader(shader_program, v_shader);
    glAttachShader(shader_program, f_shader);
    glLinkProgram(shader_program);
    glDeleteShader(v_shader);
    glDeleteShader(f_shader);
    glUseProgram(shader_program);
}
```

在初始化函数中，使用glUniform设置projection矩阵如下。

```
GLfloat projection_matrix[16];
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0f, (GLfloat)width() / (GLfloat)height(), 0.1f,
200.0f);
glGetFloatv(GL_PROJECTION_MATRIX, projection_matrix);
glUniformMatrix4fv(glGetUniformLocation(shader_program, "projection"),
1, GL_FALSE, projection_matrix);
```

设置modelview矩阵，不透明度等同理。

2.2.2 物体类

由于这里默认所有的物体都是通过VBO，VAO，EBO画图，并且都贴图。为了方便起见，实现一个object类，其中包含了物体的VBO，VAO，EBO，纹理的标识属性，和内容/路径等绘制时可能需要重复使用的内容，以及绘制物体的函数。

```
class object : public QOpenGLWidget, protected QOpenGLExtraFunctions {
public:
    object(GLfloat* vertices, GLuint vertices_size, GLuint* indices,
GLuint indices_size, char* path);
    ~object();
    void set_texture();
    void draw();

private:
    GLuint VBO, VAO, EBO;
    GLuint texture;
    char* path;

    GLfloat* vertices;
    GLuint* indices;
    GLuint vertices_size, indices_size;
};
```

类的构造函数实现如下，需要初始化成员变量，并且给VAO和VBO生成标识（因为考虑到可能会有不用EBO的模型，所以这里加了条件判断）。

```
object::object(GLfloat* vertices, GLuint vertices_size, GLuint*
indices, GLuint indices_size, char* path)
{
    initializeOpenGLFunctions();
    this->vertices = vertices;
    this->indices = indices;
    this->vertices_size = vertices_size, this->indices_size =
indices_size;
    this->path = path;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenTextures(1, &texture);

    if (indices_size != 0) {
        glGenBuffers(1, &EBO);
    }
}
```

析构函数则需要删除缓冲区内容，需要注意纹理的缓存一定要delete，否则会导致进程内存的不断增大。

```
object::~object()
{
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    glDeleteTextures(1,&texture);
}
```

set_texture函数是对纹理的初始化。其中设置了纹理贴图的相关参数和对当前纹理的初始化。需要注意的是，由于导入的都是png文件，所以glTexImage2D的参数需要设置成GL_RGBA，如果是jpg，则需要设置成GL_RGB。

```
void object::set_texture()
{
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    int width, height, nrChannels;
    stbi_set_flip_vertically_on_load(true);
    unsigned char* data = stbi_load(path, &width, &height,
&nrChannels, 0);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
    stbi_image_free(data);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
}
```

绘制函数的实现如下。每次绘制一个物体时，需要将其VBO，VAO，EBO和纹理存入缓冲区中。这里考虑到可能会物体不用EBO，因此如果索引数组长度不为0，则用glDrawElements画，如果为0，则用glDrawArrays画。

```
void object::draw()
{
    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, vertices_size, vertices,
GL_STATIC_DRAW);
```

```
    // position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)0);
    glEnableVertexAttribArray(0);
    // color attribute
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(1);
    // texture coord attribute
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void*)(6 * sizeof(GLfloat)));
    glEnableVertexAttribArray(2);

    set_texture();

    if (indices_size != 0) {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices_size, indices, GL_STATIC_DRAW);
        glDrawElements(GL_TRIANGLES, indices_size, GL_UNSIGNED_INT, 0);
    }
    else {
        glDrawArrays(GL_TRIANGLES, 0, vertices_size);
    }
}
```

以上便实现了物体的类。

2.2.3 功能函数

由于物体大部分都为圆柱或者长方体的形状，因此这里实现了几个常用的功能函数。

首先是输入柱体的细分程度，高度，半径，生成柱体侧面对应的VBO和EBO的函数。需要注意的是，由于侧面的贴图是首尾相接的，因此有两个顶点需要声明两次，其中的纹理坐标分别是左上，左下和右上，右下。也因此需要注意传入的指针所申请的大小需要比细分程度所对应的大小要多。

```
void MyGLWidget::get_cylinder_v_i(GLfloat r, GLfloat h, GLuint u_num, GLfloat* temp_v, GLuint* temp_i) {
    for (int i = 0; i < 2 * u_num + 2; i++) {
        if (i <= u_num) {
            temp_v[i * 8] = r * cos(PI * i * 2 / u_num);
            temp_v[i * 8 + 1] = h;
            temp_v[i * 8 + 2] = -r * sin(PI * i * 2 / u_num);
            temp_v[i * 8 + 6] = 1.0f * (double)i / u_num;
            temp_v[i * 8 + 7] = 1.0f;
            temp_v[i * 8 + 3] = cos(PI * i * 2 / u_num);
            temp_v[i * 8 + 4] = 0.0f;
            temp_v[i * 8 + 5] = -sin(PI * i * 2 / u_num);
        }
        else {
            temp_v[i * 8] = r * cos(PI * (i - u_num - 1) * 2 / u_num);
            temp_v[i * 8 + 1] = 0.0f;
            temp_v[i * 8 + 2] = -r * sin(PI * (i - u_num - 1) * 2 / u_num);
            temp_v[i * 8 + 6] = 1.0f * (double)(i - u_num - 1) / u_num;
            temp_v[i * 8 + 7] = 0.0f;
            temp_v[i * 8 + 3] = cos(PI * (i - u_num - 1) * 2 / u_num);
            temp_v[i * 8 + 4] = 0.0f;
            temp_v[i * 8 + 5] = -sin(PI * (i - u_num - 1) * 2 / u_num);
        }
    }
    for (int i = 0; i < 6 * u_num; i += 6) {
        temp_i[i] = i / 6;
        temp_i[i + 1] = i / 6 + 1;
        temp_i[i + 2] = temp_i[i] + u_num + 1;
        temp_i[i + 3] = temp_i[i + 1];
        temp_i[i + 4] = temp_i[i + 1] + u_num + 1;
        temp_i[i + 5] = temp_i[i + 2];
    }
}
```

然后是生成多边形对应的VBO和EBO的函数，通常是作为柱体的上下两面。由于多边形是由边缘的顶点和中心的点形成的三角形组合画成的，因此也需要注意额外申请空间。

```
void MyGLWidget::get_cycle_v_i(GLfloat r, GLuint u_num, GLfloat* temp_v, GLuint* temp_i) {
    for (int i = 0; i <= u_num; i++) {
        temp_v[i * 8] = r * cos(PI * i * 2 / u_num);
        temp_v[i * 8 + 1] = 0.0f;
        temp_v[i * 8 + 2] = -r * sin(PI * i * 2 / u_num);
        temp_v[i * 8 + 3] = 0.0f;
        temp_v[i * 8 + 4] = 1.0f;
        temp_v[i * 8 + 5] = 0.0f;
        temp_v[i * 8 + 6] = cos(PI * i * 2 / u_num) / 2 + 0.5f;
        temp_v[i * 8 + 7] = sin(PI * i * 2 / u_num) / 2 + 0.5f;
    }
    //圆心
    temp_v[(u_num + 1) * 8] = 0.0f;
    temp_v[(u_num + 1) * 8 + 1] = 0.0f;
    temp_v[(u_num + 1) * 8 + 2] = 0.0f;
    temp_v[(u_num + 1) * 8 + 3] = 0.0f;
    temp_v[(u_num + 1) * 8 + 4] = 1.0f;
    temp_v[(u_num + 1) * 8 + 5] = 0.0f;
    temp_v[(u_num + 1) * 8 + 6] = 0.5f;
    temp_v[(u_num + 1) * 8 + 7] = 0.5f;

    for (int i = 0; i < u_num; i++) {
        temp_i[i * 3] = i;
        temp_i[i * 3 + 1] = i + 1;
```

```
            temp_i[i * 3 + 2] = u_num + 1;
        }
    }
```

除此之外还有快捷设置modelview矩阵的函数。

```
void MyGLWidget::s_r_t_set_matrix(GLfloat s1, GLfloat s2, GLfloat s3, GLfloat r0, GLfloat r1, GLfloat r2, GLfloat r3, GLfloat t1, GLfloat t2, GLfloat t3) {
    GLfloat modelview_matrix[16];
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glTranslatef(t1, t2, t3);
    glRotatef(r0, r1, r2, r3);
    glScalef(s1, s2, s3);
    glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix);
    glUniformMatrix4fv(glGetUniformLocation(shader_program, "modelview"), 1, GL_FALSE, modelview_matrix);
    glPopMatrix();
}
```

由于采用细分程度为4的情况画长方体，因此它应当绕y轴旋转45度，再进行缩放，才能保持直角。而绘制了较多长方体，因此针对长方体也实现了一个类似的函数。

```
void MyGLWidget::r_s_r_set_matrix(GLfloat s1, GLfloat s2, GLfloat s3, GLfloat r0, GLfloat r1, GLfloat r2, GLfloat r3) {
    GLfloat modelview_matrix[16];
    ...
    glRotatef(45.0f, 0.0f, 1.0f, 0.0f);
    glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix);
    glUniformMatrix4fv(glGetUniformLocation(shader_program, "modelview"), 1, GL_FALSE, modelview_matrix);
    glPopMatrix();
}
```

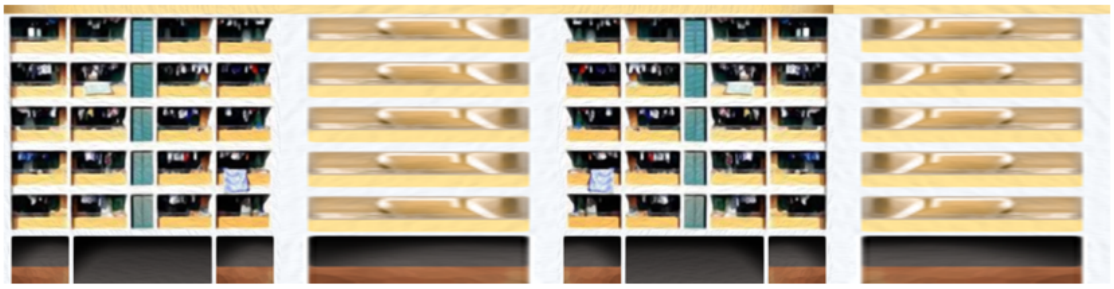
2.2.4 绘图实例

通过上面实现的类和函数，绘图的方式如下。这里以绘制长方体的楼为例。

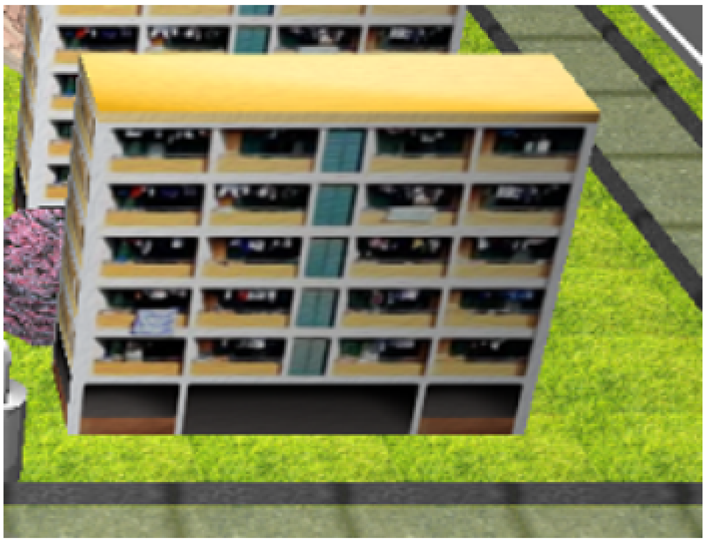
```
//获取VBO，EBO数组
int u_num = 4;
GLfloat* building_v = new GLfloat[2 * (u_num + 1) * 8];
GLuint* building_i = new GLuint[6 * u_num];
get_cylinder_v_i(1.0f, 1.0f, u_num, building_v, building_i);
GLfloat* top_v = new GLfloat[(u_num + 2) * 8];
GLuint* top_i = new GLuint[3 * u_num];
get_cycle_v_i(1.0f, u_num, top_v, top_i);

//创建object
char path_zhishan[] = "pics\\至善.png";
```

```
object building_zhishan(building_v, sizeof(GLfloat) * 2 * (u_num + 1) * 8, building_i, sizeof(GLuint) * 6 * u_num, path_zhishan);
char path_zhishantop[] = "pics\\至善楼顶.png";
object top_zhishan(top_v, sizeof(GLfloat) * (u_num + 2) * 8, top_i, sizeof(GLuint) * 3 * u_num, path_zhishantop);

//绘制
glPushMatrix();
glTranslatef(3.4f, 0.0f, 5.5f);
glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
r_s_r_set_matrix(2.0f, 2.3f, 0.5f, -90.0f, 0.0f, 1.0f, 0.0f);
building_zhishan.draw();
glTranslatef(0.0f, 2.3f, 0.0f);
r_s_r_set_matrix(2.0f, 2.3f, 0.5f, -90.0f, 0.0f, 1.0f, 0.0f);
top_zhishan.draw();
glPopMatrix();
```

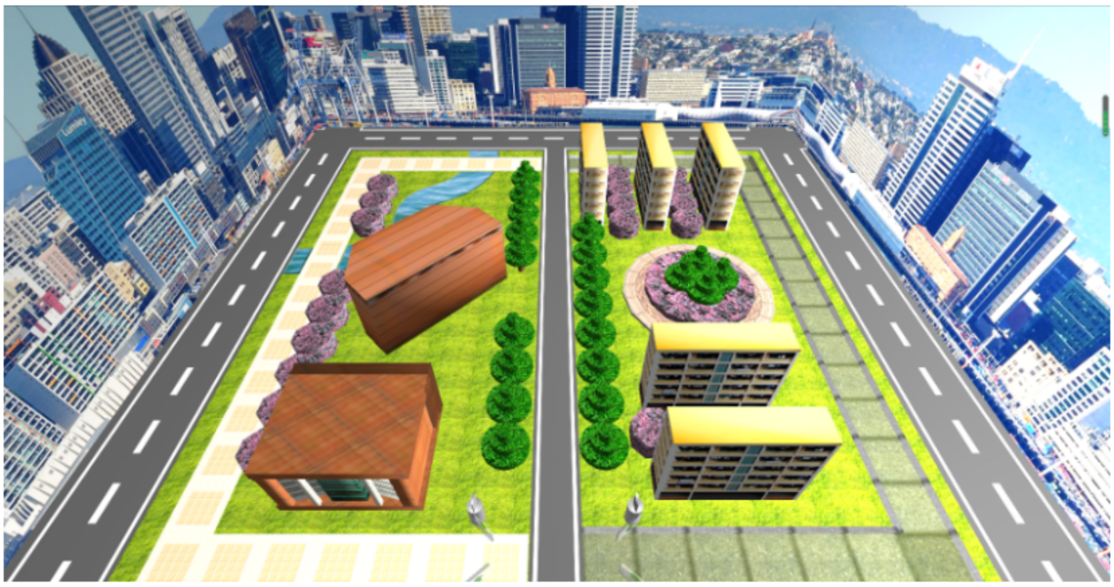
其中的贴图如下：



最后实现的效果如下。



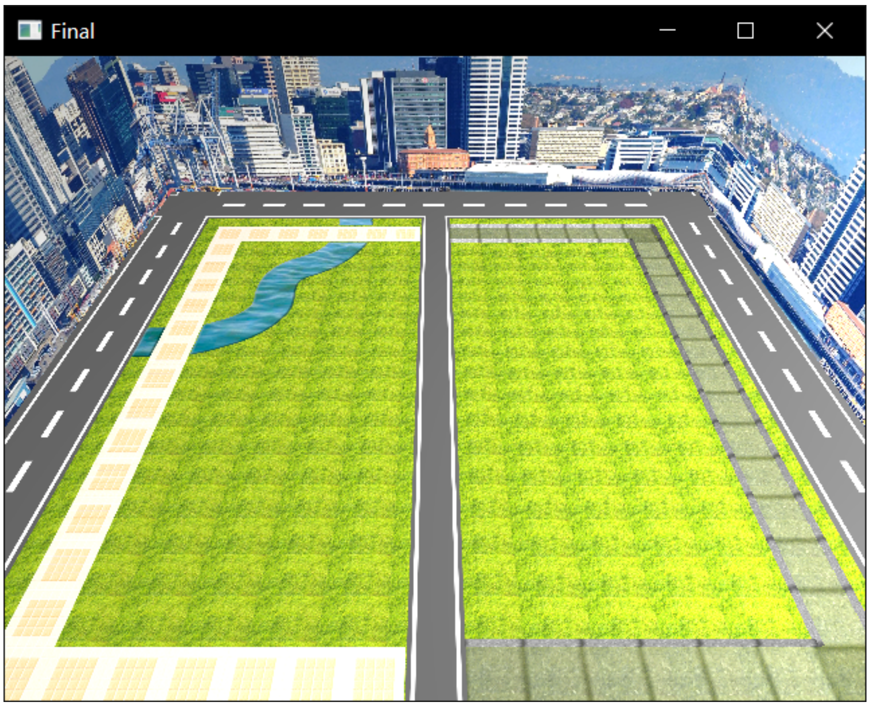
整个场景的最终效果如下：



2.2.5 天空盒

天空盒是环绕在地图周围及顶部的静态图片，它能够营造场景周围的环境，让场景更加的丰富和完整。天空盒实际上是有五个固定位置的带有对应纹理矩形拼接而成。使用我们自己编写的 `void MyGLWidget::get_cycle_v_i` 函数，调整细分程度 `n_nums` 为4，实现一个正方形的绘制，然后通过纹理、平移、旋转、放大等操作来使对应的静态图片置于相应的位置。

```
void MyGLWidget::draw_skybox() {
    GLfloat skybox_top_v[] = {
         10.0f,  15.0f,-10.0f,  0.0f, 1.0f, 0.0f,   1.0f, 1.0f, // top right
         10.0f,  15.0f,10.0f,  0.0f, 1.0f, 0.0f,   0.0f, 1.0f, // bottom right
        -10.0f,  15.0f,10.0f,  0.0f, 1.0f, 0.0f,   0.0f, 0.0f, // bottom left
        -10.0f,  15.0f,-10.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f  // top left
    };
    GLuint skybox_top_i[] = {
       3, 1, 0, // first triangle
       3, 2, 1  // second triangle
    };
    char path_top[] = "pics\\skybox_top.png";
    object skybox_top(skybox_top_v, sizeof(skybox_top_v), skybox_top_i, sizeof(skybox_top_i), path_top);
    r_t_set_matrix(0.0f, 0.0f, 10.0f, 0.0f, 0.0f, 0.0f, 0.0f);
    skybox_top.draw();
    GLfloat skybox_side_v[] = {
        -10.0f,  15.0f,-10.0f,  0.0f, 1.0f, 0.0f,   1.0f, 1.0f, // top right
        -10.0f,  0.0f,-10.0f,  0.0f, 1.0f, 0.0f,   1.0f, 0.0f, // bottom right
        -10.0f,  0.0f,10.0f,  0.0f, 1.0f, 0.0f,   0.0f, 0.0f, // bottom left
        -10.0f,  15.0f,10.0f,  0.0f, 1.0f, 0.0f,   0.0f, 1.0f  // top left
    };
    GLuint skybox_side_i[] = {
       0, 1, 3, // first triangle
       1, 2, 3  // second triangle
    };
    char path_front[] = "pics\\skybox_front.png";
    object skybox_front(skybox_side_v, sizeof(skybox_side_v),
skybox_side_i, sizeof(skybox_side_i), path_front);
    r_t_set_matrix(0.0f, 0.0f, 10.0f, 0.0f, 0.0f, 0.0f, 0.0f);
    skybox_front.draw();

    char path_right[] = "pics\\skybox_right.png";
    object skybox_right(skybox_side_v, sizeof(skybox_side_v),
skybox_side_i, sizeof(skybox_side_i), path_right);
    r_t_set_matrix(-90.0f, 0.0f, 10.0f, 0.0f, 0.0f, 0.0f, 0.0f);
    skybox_right.draw();

    char path_back[] = "pics\\skybox_back.png";
    object skybox_back(skybox_side_v, sizeof(skybox_side_v),
skybox_side_i, sizeof(skybox_side_i), path_back);
    r_t_set_matrix(180.0f, 0.0f, 10.0f, 0.0f, 0.0f, 0.0f, 0.0f);
    skybox_back.draw();

    char path_left[] = "pics\\skybox_left.png";
    object skybox_left(skybox_side_v, sizeof(skybox_side_v),
skybox_side_i, sizeof(skybox_side_i), path_left);
    r_t_set_matrix(90.0f, 0.0f, 10.0f, 0.0f, 0.0f, 0.0f, 0.0f);
    skybox_left.draw();
}
```



2.3 动画

2.3.1 棋子的移动

棋子的移动动画函数从游戏主要逻辑的函数中调用。根据传入的位置和方向绘制棋子跳动到下一个格子的动画。高度通过一个二次函数根据当前步长进行计算，而x，z轴的位移通过对应方向和当前的步长进行修改。为了避免棋子在同一格导致位置覆盖，这里对于两个棋子，在最后进行反方向的位移调整。

```cpp
void MyGLWidget::move_p(GLuint number, GLint dir, GLfloat x, GLfloat z, bool Ismove) {

    GLfloat angle = (dir - 2) * 90.0f;
    GLfloat height = -step[number - 1] * step[number - 1] + step[number - 1];
    GLfloat offset = 0.20f;
    GLfloat x_dir[5] = { 0.0f, -1.0f, 0.0f, 1.0f, 0.0f };
    GLfloat z_dir[5] = { 0.0f,  0.0f, 1.0f, 0.0f, -1.0f };
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    if (number == 1) {
        glTranslatef(offset, 0.0f, offset);
    }
    else {
        glTranslatef(-offset, 0.0f, -offset);
    }
    glPushMatrix();
    glTranslatef(x + x_dir[dir] * step[number - 1], height, z + z_dir[dir] * step[number - 1]);
    glRotatef(angle, 0.0f, 1.0f, 0.0f);
    draw_p(number);
    glPopMatrix();
    glPopMatrix();
    //printf("number:%d,move:%d,step:%.3f\n", number, Ismove, step[number - 1]);
    if (Ismove == true)
    {
        step[number - 1] += 0.20f;
        if (abs(step[number - 1] - 1.0f) < 1e-3) {
            step[number - 1] = 0.0f;
        }
    }
    else
    {
        height = 0.0;
    }
}
```

2.3.2 闸机

闸机的部分是由两边的圆柱体和长方体（闸机门）组成。其中动画实现了开门与关门。采用bool变量记录门的状态（打开，关闭，正在打开，正在关闭），并且每次调用时改变angle，以实现动画效果。其大致实现如下：

```cpp
...//闸机两侧的绘制
```

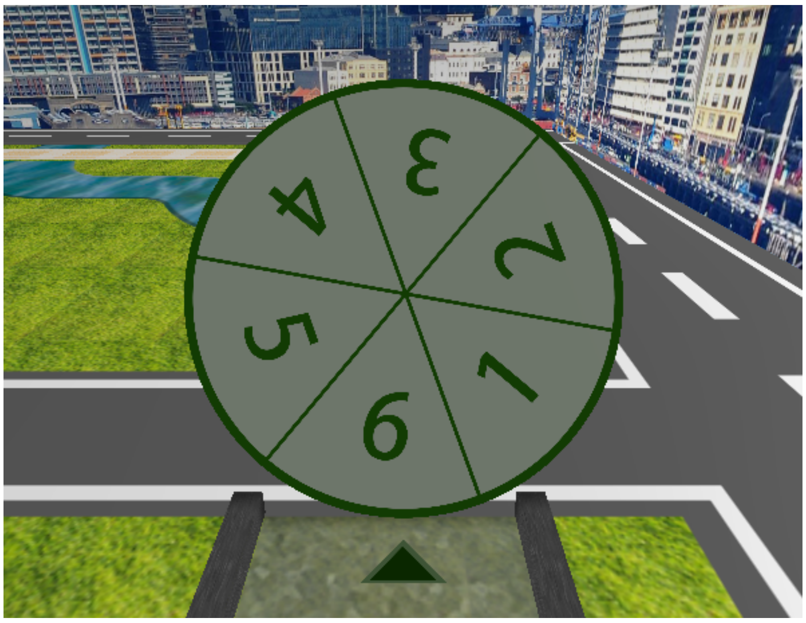
```
if (close_flag1) {
    glPushMatrix();
    glTranslatef(1.5f, 0.2f, 6.6f);
    r_s_r_set_matrix(1.0f, 1.0f, 0.1f, -90.0f, 0.0f, 1.0f, 0.0f);
    gate.draw();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(1.5f, 0.2f, 7.3f);
    r_s_r_set_matrix(1.0f, 1.0f, 0.1f, -90.0f, 0.0f, 1.0f, 0.0f);
    gate.draw();
    glPopMatrix();
}
else if (openning_flag1) {
    //打开中，绘制旋转后的两侧的门
    angle1 += 10.0f;
    glPushMatrix();
    glTranslatef(1.5f, 0.2f, 6.6f);
    glTranslatef(0.02f, 0.0f, -0.2f);
    glRotatef(-angle1, 0.0f, 1.0f, 0.0f);
    glTranslatef(-0.02f, 0.0f, 0.2f);
    r_s_r_set_matrix(1.0f, 1.0f, 0.1f, -90.0f, 0.0f, 1.0f, 0.0f);
    gate.draw();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(1.5f, 0.2f, 7.3f);
    glTranslatef(0.02f, 0.0f, 0.2f);
    glRotatef(angle1, 0.0f, 1.0f, 0.0f);
    glTranslatef(-0.02f, 0.0f, -0.2f);
    r_s_r_set_matrix(1.0f, 1.0f, 0.1f, -90.0f, 0.0f, 1.0f, 0.0f);
    gate.draw();
    glPopMatrix();

    if (abs(angle1 - 90.0f) < 1e-4) {
        openning_flag1 = FALSE;
        open_flag1 = TRUE;
    }
}
else if (open_flag1) {
    ...
}
else if (closing_flag1) {
    ...
    angle1 -= 10.0f;
    gate.draw();
    ...
    if (abs(angle1 - 0.0f) < 1e-4) {
        closing_flag1 = FALSE;
        close_flag1 = TRSUE;
    }
}
```

2.3.3 转盘

使用`void MyGLWidget::get_cycle_v_i`函数，调整细分程度`n_nums`为16，画出一个圆形，然后将转盘的贴图加入。另外绘制一个指针，置于转盘的最下方，作为当前转到点数的的指示。将每次初始时指向的点数设置为6，然后根据输入的目标点数（target）算出旋转的角度，开始顺时针旋转，旋转角度递减。当旋转的角度小于目标角度的时候，停止旋转。

由于需要将转盘移入camera视角内，因此需要调整转盘在xoz平面投影的位置以及仰角，将这些变量作为输入参数，以供其他模块调用：

```
void MyGLWidget::draw_plate(GLint target_num, GLfloat x, GLfloat y, GLfloat z, GLint direction)
{
    GLfloat d[] = {0.0f,0.0f,90.0f,180.0f,270.0f};
    GLfloat cam_angle = d[direction];
    GLfloat target_angle = 30.0f-target_num*60.0f;
    if (is_running)
    {
        printf("is_running:%d",is_running);
        if (plate_angle <= target_angle)
        {
            is_running = false;
        }
        plate_angle-=10.0f;
    }
    GLint u_num = 180;
    GLfloat r = 0.3f;
    GLfloat angle=30.0f;
    GLfloat* plate_v = (GLfloat*)malloc(sizeof(GLfloat) * (u_num + 2) * 8);
    GLuint* plate_i = (GLuint*)malloc(sizeof(GLuint) * u_num * 3);
    get_cycle_v_i(r, u_num, plate_v, plate_i);
    char path_plate[] = "pics\\plate.png";
    object plate(plate_v, sizeof(GLfloat) * (u_num + 2) * 8, plate_i, sizeof(GLuint) * u_num * 3, path_plate);

    GLfloat modelview_matrix[16];
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glTranslatef(x, y, z);
    glRotatef(cam_angle, 0.0f, 10.0f, 0.0f);
    glTranslatef(-0.1f, 0.0f, 0.0f);
    glRotatef(-90.0f, 0.0f, 0.0f, 10.0f);
    glRotatef(angle,0.0f,0.0f,10.0f);
    glRotatef(plate_angle, 0.0f, 10.0f, 0.0f);

    glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix);
    glUniformMatrix4fv(glGetUniformLocation(shader_program, "modelview"), 1, GL_FALSE, modelview_matrix);
    glPopMatrix();
    plate.draw();

    GLfloat arrow_v[] = {
         0.0f,  -r, r / 6.0f,  1.0f, 1.0f, 1.0f,   0.0f, 0.0f,
         0.0f,  -r, r / -6.0f,  1.0f, 1.0f, 1.0f,   1.0f, 0.0f,
         0.0f,  r / 6.0f - r , 0.0f,  1.0f, 1.0f, 1.0f,   0.5f, 1.0f,
    };

    GLuint arrow_i[] = {
        0,1,2
    };

    char path_arrow[] = "pics\\arrow.png";
    object arrow(arrow_v, sizeof(arrow_v), arrow_i, sizeof(arrow_i), path_arrow);

    GLfloat modelview_matrix_arrow[16];
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glTranslatef(x, y, z);
    glRotatef(cam_angle, 0.0f, 10.0f, 0.0f);
    glRotatef(angle,0.0f,0.0f,10.0f);
    glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix_arrow);
    glUniformMatrix4fv(glGetUniformLocation(shader_program, "modelview"), 1, GL_FALSE,
                       modelview_matrix_arrow);
    glPopMatrix();
    arrow.draw();
}
```



3 整体结果展示

见录屏及ppt

4 分工

18340012 陈晨：着色器实现，物体类的的设计，部分动画的设计、大部分贴图制作，地图上建筑、闸机、棋子的绘制

18340056 胡邱诗雨：游戏逻辑实现，跟随棋子的视角控制，棋子移动控制、物体动画调用控制、按键交互实现

18340195 杨智博：地图上灌木丛、树木、花坛的建模，转盘的绘制，天空盒以及终点贴图的绘制