实验二　跑跑卡丁车

严铖　517021910823

1　实验目的与基本要求

1. 熟练掌握和综合运用 OpenGL 编程技术来开发简单的三维交互式游戏

2　实验内容

1. 设计并绘制一辆汽车模型（该模型可以是导入的 OBJ 模型）以及一个简单的直线跑道；缺省视图是从外面一个固定的视点观察汽车和跑道
2. 在缺省视图下，绘制汽车在跑道上的阴影（自定义一个假想的点光源）
3. 利用鼠标和键盘控制汽车前进、后退、转弯、加速和减速
4. 定义对应于 ReShape 事件的回调函数，使得当用户改变窗口的大小时，显示的汽车不会变形
5. 采用弯曲的封闭的跑道
6. 制作一个弹出菜单，上面的菜单项用来自定义小车的部分部件如车轮、车体颜色等以及退出程序
7. 在路边设置一些路标，对于地面、跑道、天空进行纹理映射等以增强逼真度
8. 除了缺省视图之外，支持第二种视图：坐在车内从驾驶座位向前看的视图。两种视图之间用“t”键进行切换

3 实验步骤

本次实验基于 LearnOpenGL CN 教程完成,教程网址为:https://learnopengl-cn.github.io。在此向教程的作者表示由衷的感谢。

实验中使用的是 GLFW 环境以及开源库 GLAD，均使用的是最新版本。此外，OpenGL 的数学库 glm 也会在实验中被使用。

以下，根据实验内容，我们分别讨论各项所需的实验步骤

3.1 汽车模型、跑道、路标的绘制

在上一次实验绘制龙模型的基础上，我们使用自己写的模型导入库进行模型、纹理等的导入。具体代码可参见源代码 model.h 以及 mesh.h，其中 mesh.h 构造了绘图的一个基本单元 mesh，model.h 在 mesh 库的基础上，通过外部库 assimp 对 obj 以及相关联的 mtl 文件进行读取，再将读取到的 assimp 格式的数据转换成库内置的一个 mesh 向量。

```
50  private:
51      /*  Functions   */
52      // loads a model with supported ASSIMP extensions from file and stores the result
53      void loadModel(string const& path)
54      {
55          // read file via ASSIMP
56          Assimp::Importer importer;
57          const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProc
58          // check for errors
59          if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
60          {
61              cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
62              return;
63          }
64          // retrieve the directory path of the filepath
65          directory = path.substr(0, path.find_last_of('\\'));
66
67          // process ASSIMP's root node recursively
68          processNode(scene->mRootNode, scene);
69      }
```

图 1: Model 类构造函数代码截图

我在网络上找到了汽车模型、跑道以及路标等的 obj、mtl 文件以及其附带的纹理。因此我们构造三个 model 对象分别对应三个模型，构造函数的参数为 obj 文件所在的路径。（助教如希望在自己的电脑上运行程序，请现将构造函数的路径改为自己电脑相关文件所在的路径）再根据各个模型的大小、所在的位置、朝向等调整每个模型相应的 model 矩阵，便可以将三者组合在合适的位置上。

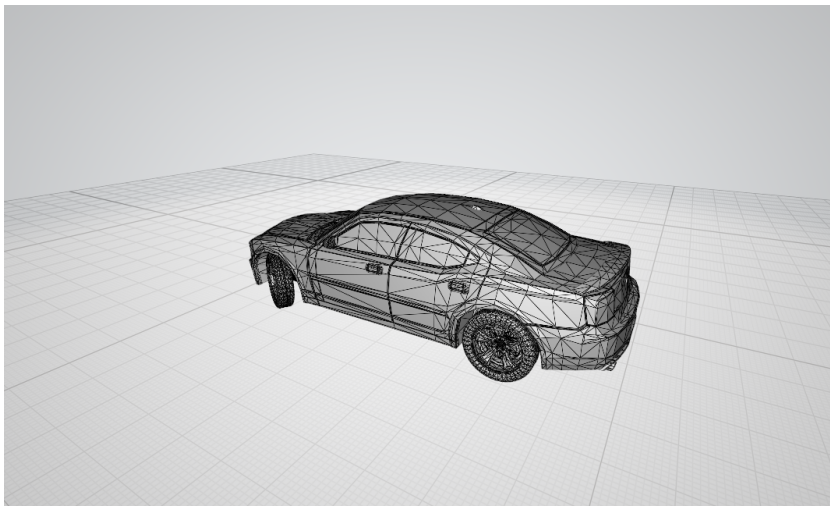

图 2: 实验中使用的汽车模型

3.2 ReShape 事件

ReShape 事件对应着 OpenGL 窗口大小的改变。因此我们需要注册一个相对应的回调函数。

GLFW 的`glfwSetWindowSizeCallback()` 函数便能够对窗口大小的改变做出反应。我们将自己定义的`reshape` 函数作为其参数，其中`reshape` 函数将实时变化的窗口的长、宽大小复制到我们的全局变量`WIDTH, HEIGHT` 中，以进行画面及时的调整。

3.3 汽车的运动

我们的汽车应当能够做到如下运动：前进、后退、转弯、加速以及减速。以下我们分别讨论各个运动方式的实现。

3.3.1 前进与后退

为了实现汽车的运动，我们定义一个全局变量`v` 表示汽车前后运动的速度。在`processInput()` 函数内，如果按键 W 被触发，则在`deltaTime` 时间内，我们定义：速度将自增加 10。反之，若 S 被触发，则速度自减 10。

当然，速度不能无限地增加或减少下去，因此我们定义变量`max_v`，来决定速度的最大上限。若速度大于了该值，则让其等于这个最大值（负值也

同理）。

此外，我们增加了一个摩擦力系统。在每个单位时间，汽车都会受到一定的与速度方向相反的阻力。该阻力使速度的绝对值减少 0.5，若速度为 0 或较小时则不改变。

因此有了速度变量`v`，我们便可以在处理完按键事件更新了变量后，对汽车的`model` 矩阵进行相对应的 translate 调整了

3.3.2 加速与减速

在上述的运动系统中，加速与减速的解决就十分简单了。

对于加速，我们将`max_v` 变量调大以获得更高的最大速度，同时将单位时间自增的速度增大以获得更高的加速度，从而实现汽车的加速。

对于减速，我们做的是增大摩擦力的大小，其作用甚至大于汽车的动力对速度的影响从而实现刹车。

3.3.3 转弯

转弯的实现较上述两种情况就相对更加复杂一些了。我们用一个稍微简单的模型以实现汽车的转弯。以下以按下 W 与 A 键向左转弯为例。

在转弯时视为小车模型在绕着车后轮轴所在直线上的某一点进行圆周运动。如小车要向左转弯时，则小车绕着距离后轴中心`r` 处左侧的轴进行旋转。

基于这种想法，当 W 与 A 被同时按下时，我们会通过几何关系自动计算出这个轴在世界坐标系下的位置，再通过 translate 和 rotate 调整使汽车进行相应的转动。具体的实现请参见源代码文件 myFunction.h。

```
188     // turn left
189     if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS && glfwGetKey(window, GLFW_KEY_W
190     {
191         float x = r * (head_n.z - rear_n.z) / glm::length(t) + rear_n.x;
192         float z = -r * (head_n.x - rear_n.x) / glm::length(t) + rear_n.z;
193         glm::vec3 axis(x, 0.0f, z);
194 
195         // cout << x << " " << z << " " << rear_n.x << " " << rear_n.z << endl;
196 
197         model = glm::translate(model, -axis);
198         model = glm::rotate(model, glm::radians(d * deltaTime), glm::vec3(0.0f, 1.0f,
199         model = glm::translate(model, axis);
200     }
201 
202     // turn right
203     if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS && glfwGetKey(window, GLFW_KEY_W
204     {
205         float x = -r * (head_n.z - rear_n.z) / glm::length(t) + rear_n.x;
206         float z = r * (head_n.x - rear_n.x) / glm::length(t) + rear_n.z;
```

图 3: 转弯代码部分截图

3.4 阴影的绘制

首先我们应当绘制深度贴图，其本质是一种由光渲染的深度纹理。因此与纹理的初始构造类似，我们生成我们的深度贴图depthmap，但是一些相关函数的参数可能会根据深度贴图的需要发生少许的变化。

```
106     // Configure depth map FBO
107     const GLuint SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
108
109     GLuint depthMapFBO;
110     glGenFramebuffers(1, &depthMapFBO);
111
112     // - Create depth texture
113     GLuint depthMap;
114     glGenTextures(1, &depthMap);
115     glBindTexture(GL_TEXTURE_2D, depthMap);
116
117     glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0
118     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
119     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
120     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
121     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
122
123     GLfloat borderColor[] = {1.0, 1.0, 1.0, 1.0};
124     glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
125
126     glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
127     glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthM
```

图 4: depthmap 的初始化

在真正绘制我们的几个物体之前，我们应当以光的透视图进行场景的渲染。因此我们构建一个简单的着色器 simpleDepthShader，在顶点着色器中通过传入的光变换矩阵将视图转换，而片段着色器并不做其他作用。

```
1   #version 330 core
2   layout (location = 0) in vec3 aPos;
3
4   uniform mat4 lightSpaceMatrix;
5   uniform mat4 model;
6
7   void main()
8   {
9       gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
10  }
```

图 5: 深度着色器中的顶点着色器内容

在渲染循环的开始，我们先对光变换矩阵进行计算。其满足公式 $lightSpaceMatrix = lightProjection \times lightView$。其中 $lightView$ 为以光源看向原点的 lookAt 矩阵，而 lightProjection 为透视变换矩阵。将计算得出的光变换矩阵传入深度着色器中，最后我们用这个着色器依次渲染各个物体，并存储深度贴图。

有了深度贴图纹理之后，我们便可以将其导入我们之后真正绘制物体的着色器之中进行阴影的计算了。绘制过程不再赘述，在绘制之前应用深度贴图即可。在着色器内计算阴影的函数如下图所示：

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range fragPosLight
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // calculate bias (based on depth map resolution and slope)
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
    // check whether current frag pos is in shadow
    // float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
    // PCF
    float shadow = 0.0;
    vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
    for(int x = -1; x <= 1; ++x)
```

图 6: 阴影计算代码部分截图

绘制出的阴影效果如图中所示：

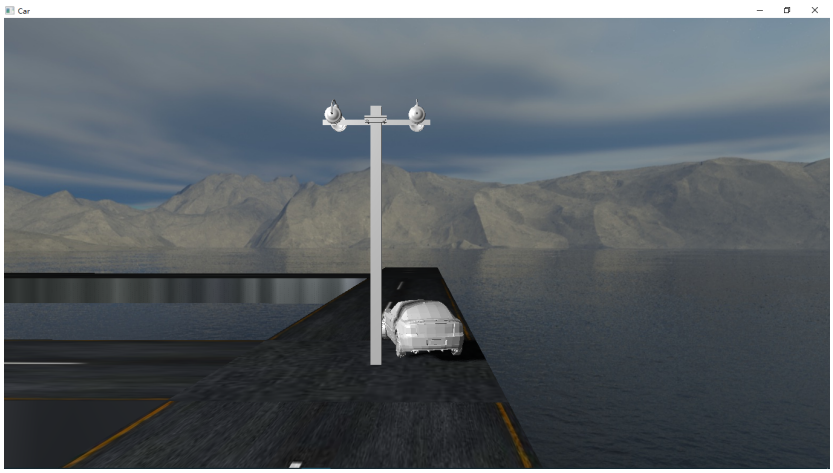

图 7: 小车及路面上的阴影

3.5 驾驶座视图

为了实现第二种视图，我们应当再定义一个新的摄像机camera_fol，与之前缺省视图不同的是，这个摄像机的位置与朝向都是随着车的运动时时刻刻在改变的。因此在这个相机初始化的时候，其位置应当等同于汽车驾驶位上的某一点在世界坐标系下的位置，其朝向应当与车头的正向相同。这些在经过一系列实验后便可以得到正确的参数。

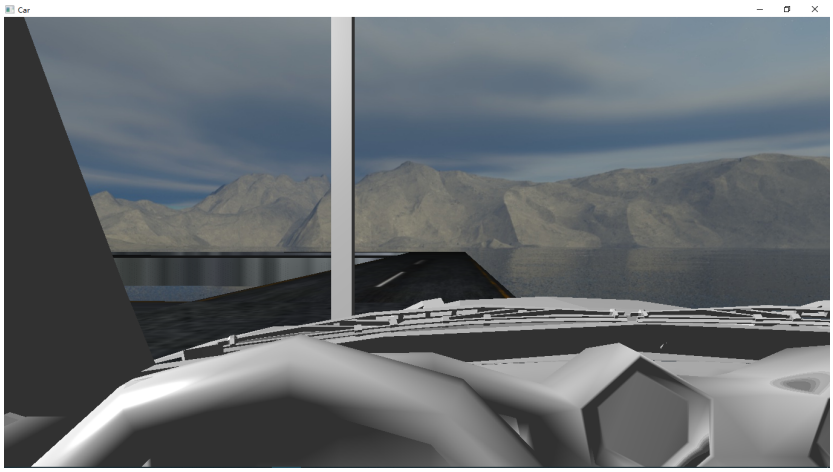

图 8: 驾驶座视图上的视野

与此同时，我们通过processInput 函数内对键盘上 t 键的检测进行模式上的变换。如模式改变,则在绘制图形的着色器中上传的view，projection，viewPos 等矩阵就由另一个摄像机所提供，从而进行视角上的变换。

3.6 天空盒的绘制

天空盒本质上是一种立体贴图。立方体贴图由六个正方形的二维贴图组成，在给出了方向向量之后，只要正方体的中心在世界坐标轴的原点，便可以将贴图上的一点投射到远处的平面中。

与其他纹理的创建类似，立方体贴图的构建如图中的代码所示：

其中函数loadCubemap() 使用stb_image 库提取路径下的纹理，并返回一个纹理 ID。

在创建了纹理之后，便需要在场景中进行渲染。我们在绘制物体之后进行这个步骤，其使用一个新的着色器进行渲染。具体的着色器内容可参见源

```
134     // skybox
135     float skyboxVertices[] = { ... }
179     // skybox VAO
180     unsigned int skyboxVAO, skyboxVBO;
181     glGenVertexArrays(1, &skyboxVAO);
182     glGenBuffers(1, &skyboxVBO);
183     glBindVertexArray(skyboxVAO);
184     glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
185     glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices, GL_STATIC_
186     glEnableVertexAttribArray(0);
187     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
188
189     vector<std::string> skybox_path
190     { ... }
198
199     unsigned int cubemapTexture = loadCubemap(skybox_path);
```

图 9: 立方体贴图构造代码

代码 shadow_fs.txt, shadow_vs.txt。绘制出的天空盒如下图所示：

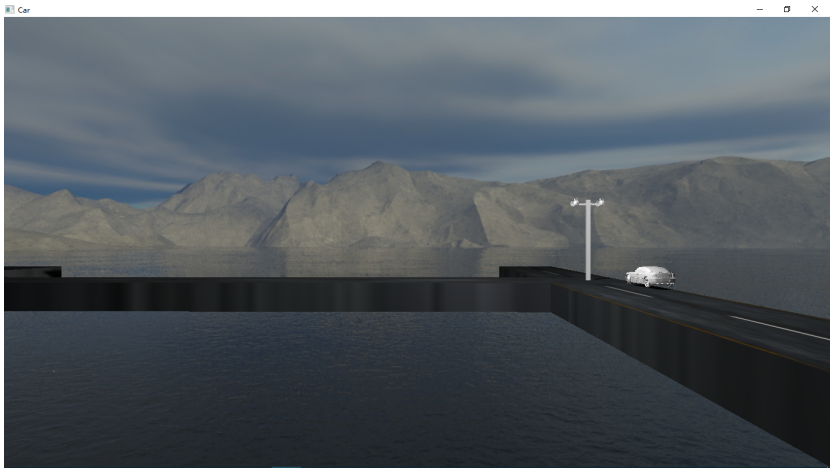

图 10: 天空盒的一面

4 实验小结

本次实验我对 OpenGL 类库以及模型从构建到渲染等步骤都有了一个很深刻的认识。相较于第一次实验，我了解了更多高级的光照、渲染技巧，明白了其之中蕴含的数学及计算机原理。

实验让我对 c++ 编程以及面向对象的思想有了更好的掌握，有了 model 及 mesh 类，对于未来对数据模型的导入便只需提供文件所在的路径即可，十分方便。

但是可能仍然有一些选做的要求未能完成。例如弹出式菜单的制作在 GLFW 内并不提供支持，而在 OpenGL 的另外一个类库 GLUT 有提供相对应的接口。因此由于使用的库的局限性，无法完成弹出式菜单的制作，如果未来有机会，我也希望对 GLUT 库也能有一些更加深刻的了解。