RaceCar 游戏程序设计

> <韩艺轩 3210105354> <赵宇天 3210105567>

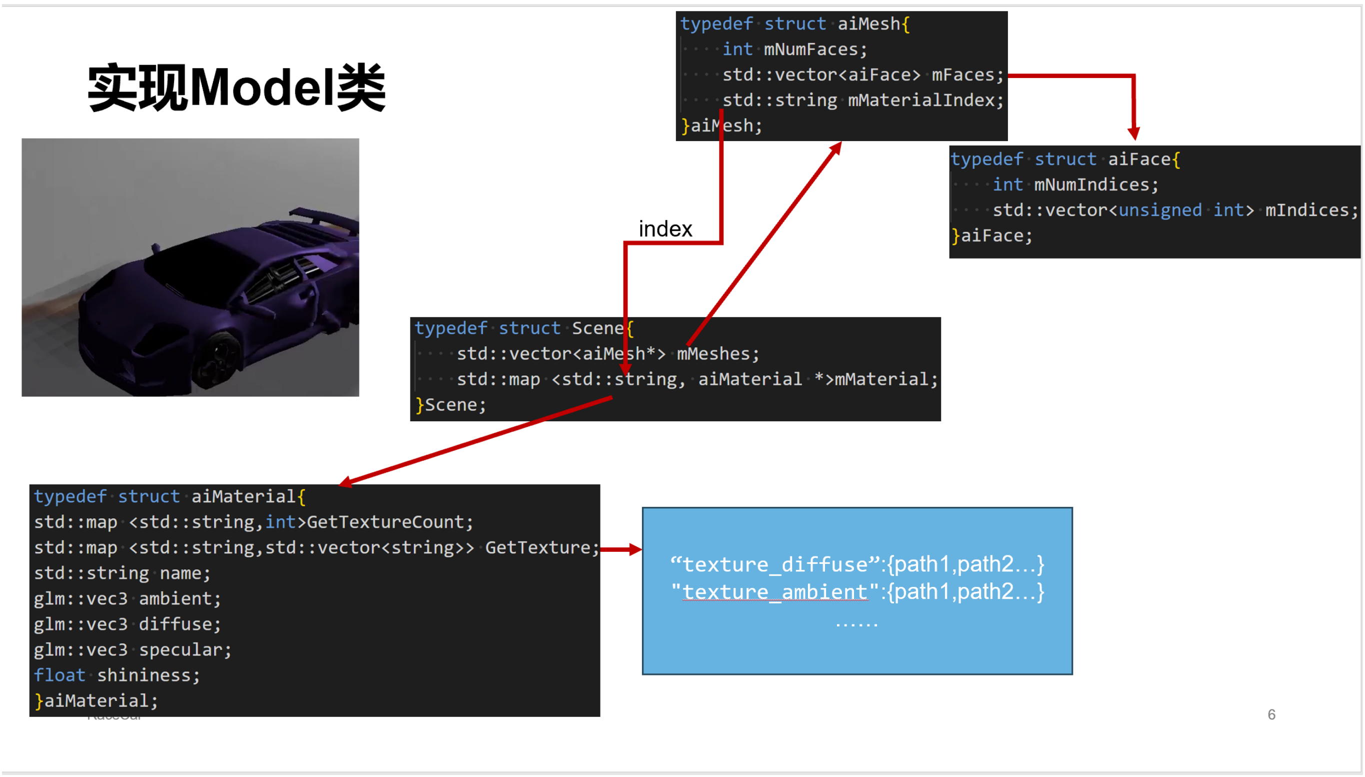
模型读取

实现 my_assimp.h

在读取模型时，我们并没有使用`Assimp`库，而是使用了自己实现的`my_assimp.h`实现了相关功能。

```
#pragma once
#include <vector>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <Mesh.h>
#include <map>
typedef struct aiNode{
    int mNumMeshes;
    int mNumChildren;
    std::vector<int> mMeshes;
    std::vector<aiNode*> mChildren;
}aiNode;

typedef struct aiFace{
    int mNumIndices;
    std::vector<unsigned int> mIndices;
}aiFace;

typedef struct aiMesh{
    int mNumFaces;
    std::vector<aiFace> mFaces;
    std::string mMaterialIndex;
}aiMesh;

typedef struct aiMaterial{
std::map <std::string,int>GetTextureCount;
std::map <std::string,std::vector<string>> GetTexture;
std::string name;
glm::vec3 ambient;
glm::vec3 diffuse;
glm::vec3 specular;
float shininess;
}aiMaterial;

typedef struct Scene{
    aiNode *mRootNode;
    std::vector<aiMesh*> mMeshes;
    // std::vector<aiMaterial *>mMaterial;
    std::map <std::string, aiMaterial *>mMaterial;
}Scene;
```

用这种方法定义了构建场景的重要结构体，他们链接关系如下：



实现Model类

```
typedef struct aiMesh{
    int mNumFaces;
    std::vector<aiFace> mFaces;
    std::string mMaterialIndex;
}aiMesh;
```

```
typedef struct aiFace{
    int mNumIndices;
    std::vector<unsigned int> mIndices;
}aiFace;
```

```
typedef struct Scene{
    std::vector<aiMesh*> mMeshes;
    std::map <std::string, aiMaterial *>mMaterial;
}Scene;
```

```
typedef struct aiMaterial{
std::map <std::string,int>GetTextureCount;
std::map <std::string,std::vector<string>> GetTexture;
std::string name;
glm::vec3 ambient;
glm::vec3 diffuse;
glm::vec3 specular;
float shininess;
}aiMaterial;
```

"texture_diffuse":{path1,path2...}
"texture_ambient":{path1,path2...}
......

每个obj模型都被加载入一个Scene中，其中包含一系列的Mesh和Material，只需要分开绘制每个Mesh，并使用Material的名字索引对应的Material完成渲染。

实现 Model.h

有了模型载入的 my_assimp 中的结构体之后，我们需要一个能操作该文件的类，命名为 Model.h,基本架构如下，实现一个 Model 的实例时需要传给他一个obj文件路径，在构造函数中调用loadModel()函数

```
class Model {
public:

    vector<Texture> textures_loaded;
    vector<Mesh>    meshes;
    string directory;
    std::vector<glm::vec3> mNormals;
    std::vector<glm::vec2> mTextureCoords;
    std::vector<Vertex> Vertices;
    int HasNormals;
    int HasTexture;
    Model(std::string path) {
        HasNormals = 0;
        HasTexture = 0;
        loadModel(path);
    }
    void Draw(Shader &shader) {
        for (unsigned int i = 0; i < meshes.size(); i++){
            // std::cout<<"Draw!"<<std::endl;
            meshes[i].Draw(shader);
        }
    }
private:
    ......
}
```

loadModel() 用于读入obj文件(下面也展示了loadMaterial()要调用的读入mtl文件的函数),按行读取，对于每个以 o 开始的Mesh，将v,vt,vn分别存进三个vector中，然后按照face的对应关系从三个vector中提取相应的属性，存入Vertex结构体列表中，利用这些信息构建一个个aiMesh，存入Scene中，Vertex结构体如下：

```
struct Vertex {
    // position
    glm::vec3 Position;
    // normal
    glm::vec3 Normal;
    // texCoords
    glm::vec2 TexCoords;
};
```

*Vetex结构体将会在后续用到。

```
    void loadModel(const std::string& path) {
        std::ifstream file(path);
        // std::cout<< "begin loadmodel!"<<std::endl;
        if (!file.is_open()) {
            std::cerr << "Failed to open file: " << path << std::endl;
            return;
        }
        Scene scene;
        scene.mRootNode = new aiNode;
        scene.mRootNode->mNumMeshes=0;
        scene.mRootNode->mNumChildren=0;
        aiMesh *current_aiMesh;
        std::string line;
        while (std::getline(file, line)) {
            std::istringstream iss(line);
            std::string type;
            iss >> type;
            // std::cout<<type<<std::endl;
            if (type == "v") {
                glm::vec3 vertex;
                float x,y,z;
                iss >> x >> y >> z;
                // cout<< vertex.x<< " " <<vertex.y<<" "<<vertex.z<<endl;
                vertex = glm::vec3(x, y, z);
                Vertex v;
                v.Position = vertex;
                Vertices.push_back(v);
            } else if (type == "vt") {
                HasTexture = 1;
                glm::vec2 texture;
                float x, y;
                iss >> x >> y;
                texture = glm::vec2(x, y);
                mTextureCoords.push_back(texture);
            } else if (type == "vn") {
                HasNormals = 1;
                glm::vec3 normal;
                float x, y, z;
                iss >> x >> y >> z;
                normal = glm::vec3(x, y, z);
                mNormals.push_back(normal);
            } else if (type == "f") {
                std::istringstream s(line.substr(2));
                std::string splitted;
                std::vector<unsigned int> temp_indice;
                std::vector<unsigned int> temp_vt;
                std::vector<unsigned int> temp_vn;
                while(std::getline(s, splitted, ' ')){
                    unsigned int index,vt,vn;
                    char zifu;
                    std::istringstream temp_iss(splitted);
                    temp_iss >> index >> zifu >> vt >>zifu >>vn;
                    // std::cout<<index<<" "<<vt<<" "<<vn<<std::endl;
                    temp_indice.push_back(index- 1);
                    temp_vt.push_back(vt - 1);
                    temp_vn.push_back(vn - 1);
                }
                for(int i = 2;i<temp_indice.size();i++){
                    aiFace face;
                    // std::cout<<&face<<std::endl;
                    face.mNumIndices = 0;
                    // cout << "temp"<<temp_indice[0] << " "<<temp_indice[i-1]<<"
"<<temp_indice[i]<<endl;
                    // exit(1);
                    face.mIndices.push_back(temp_indice[0]);
                    face.mIndices.push_back(temp_indice[i-1]);
                    face.mIndices.push_back(temp_indice[i]);
                    face.mNumIndices += 3;
                    current_aiMesh->mFaces.push_back(face);
                    current_aiMesh->mNumFaces ++;
                    if(HasTexture==1){
                        Vertices[temp_indice[0]].TexCoords =
mTextureCoords[temp_vt[0]];
                    }else{
                        Vertices[temp_indice[0]].TexCoords = glm::vec2(0.0f,
0.0f);
                    }
                    if(HasNormals==1)
                        Vertices[temp_indice[0]].Normal = mNormals[temp_vn[0]];
                    else
                        Vertices[temp_indice[0]].Normal = glm::vec3(0.0f, 0.0f,
0.0f);
```

```
                    if(HasTexture)
                        Vertices[temp_indice[i-1]].TexCoords =
mTextureCoords[temp_vt[i-1]];
                    else
                        Vertices[temp_indice[i-1]].TexCoords = glm::vec2(0.0f,
0.0f);
                    if(HasNormals)
                        Vertices[temp_indice[i-1]].Normal = mNormals[temp_vn[i-
1]];
                    else
                        Vertices[temp_indice[i-1]].Normal = glm::vec3(0.0f, 0.0f,
0.0f);

                    if(HasTexture)
                        Vertices[temp_indice[i]].TexCoords =
mTextureCoords[temp_vt[i]];
                    else
                        Vertices[temp_indice[i]].TexCoords = glm::vec2(0.0f,
0.0f);
                    if(HasNormals)
                        Vertices[temp_indice[i]].Normal = mNormals[temp_vn[i]];
                    else
                        Vertices[temp_indice[i]].Normal = glm::vec3(0.0f, 0.0f,
0.0f);
                }
                // std::cout<<faces.size()<<std::endl;
            }else if (type == "mtllib") {
                // cout<<"there is mtllib"<<endl;
                std::string mtlPath;
                iss >> mtlPath;
                // cout<<mtlPath<<endl;
                std::string temp = path.substr(0, path.find_last_of('/'));
                // cout<<temp<<endl;
                LoadMaterials(temp +'/' +mtlPath, &scene);
            }else if(type == "usemtl"){
                // cout<<"there is o"<<endl;
                if(scene.mRootNode->mNumMeshes > 0){
                    // current_aiMesh->mMaterialIndex = scene.mMeshes.size();
                    scene.mMeshes.push_back(current_aiMesh);
                    scene.mRootNode->mMeshes.push_back(scene.mMeshes.size()-1);
                }
                current_aiMesh = new aiMesh;
                current_aiMesh->mNumFaces = 0;
                scene.mRootNode->mNumMeshes ++;
                std::string materialindex;
                iss >> materialindex;
                // std::cout<<"index = "<<materialindex <<std::endl;
                current_aiMesh->mMaterialIndex = materialindex ;
            }
        }

        // current_aiMesh->mMaterialIndex = scene.mMeshes.size();
        scene.mMeshes.push_back(current_aiMesh);
        scene.mRootNode->mMeshes.push_back(scene.mMeshes.size()-1);
        file.close();
        // cout<<"current_aiMesh->mMaterialIndex ="<<current_aiMesh-
>mMaterialIndex<<endl;
        std::cout<<"Read Finished!"<<std::endl;
        this->directory = path.substr(0, path.find_last_of('/'));

        processNode(scene.mRootNode, &scene);
        cout<<"finish process Node"<<endl;
    }
    void LoadMaterials(const std::string& mtlPath, Scene *scene) {
        std::ifstream file(mtlPath);
        if (!file.is_open()) {
            std::cerr << "Failed to open file: " << mtlPath << std::endl;
            return;
        }
        std::string line;
        aiMaterial *currentMaterial;
        int cnt = 0;
        while (std::getline(file, line)) {
            std::istringstream iss(line);
            std::string type;
            iss >> type;

            // cout<<type<<endl;

            if (type == "newmtl") {
                if(cnt > 0){
                    scene->mMaterial[currentMaterial->name] = currentMaterial;
                }
                currentMaterial = new aiMaterial;
                string n;
                iss >> n;
                currentMaterial->name = n;
                cnt ++;
            } else if (type == "Ka") {
                iss >> currentMaterial->ambient.x >> currentMaterial->ambient.y >>
currentMaterial->ambient.z;
            } else if (type == "Kd") {
                iss >> currentMaterial->diffuse.x >> currentMaterial->diffuse.y >>
currentMaterial->diffuse.z;
            } else if (type == "Ks") {
                iss >> currentMaterial->specular.x >> currentMaterial->specular.y
>> currentMaterial->specular.z;
            } else if (type == "Ns") {
                iss >> currentMaterial->shininess;
            }else if(type == "map_Kd"){
                string path,path1;
                iss >> path;
                currentMaterial->GetTextureCount["texture_diffuse"] = 1;
                currentMaterial->GetTexture["texture_diffuse"].push_back(path);
                while(1){
                    iss >>path1;
                    if(path==path1 | path1 == "") break;
                    currentMaterial->GetTextureCount["texture_diffuse"] ++;
```

```
                        currentMaterial-
>GetTexture["texture_diffuse"].push_back(path1);
                    }
                }else if(type == "map_Ka"){
                    string path,path1;
                    iss >> path;
                    string text = "texture_ambient";
                    currentMaterial->GetTextureCount[text] = 1;
                    currentMaterial->GetTexture[text].push_back(path);
                    while(1){
                        iss >>path1;
                        if(path==path1 | path1 == "") break;
                        currentMaterial->GetTextureCount[text] ++;
                        currentMaterial->GetTexture[text].push_back(path1);
                    }
                }else if(type == "map_Ks"){
                    string path,path1;
                    iss >> path;
                    currentMaterial->GetTextureCount["texture_specular"] = 1;
                    currentMaterial->GetTexture["texture_specular"].push_back(path);
                    while(1){
                        iss >>path1;
                        if(path==path1 | path1 == "") break;
                        currentMaterial->GetTextureCount["texture_specular"] ++;
                        currentMaterial-
>GetTexture["texture_specular"].push_back(path1);
                    }
                }
            }
            scene->mMaterial[currentMaterial->name] = currentMaterial;
            cout<<"LoadMaterials Done!"<<endl;
            // Add the last material to the materials vector
            file.close();
        }
```

在上述函数中调用了processNode()函数，对整个Scene进行进一步操作。遍历场景中的每一个aiMesh，将其转换成Mesh类的实例（Mesh类是一个可直接绘制的类，在后续会详细介绍具体实现）。

```
void processNode(aiNode *node, const Scene *scene){
        for(unsigned int i = 0; i < node->mNumMeshes; i++)
        {
            aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
            meshes.push_back(processMesh(mesh, scene));
        }
        for(unsigned int i = 0; i < node->mNumChildren; i++)
        {
            processNode(node->mChildren[i], scene);
        }
    }
```

其中转换为Mesh实例的函数为processMesh(),这里就用到了刚才loadModel中读入的数据，因为Mesh类的构造如下,所以我们需要得到这些数值。这时我们刚刚保存的Vertex的列表就可以作为第一个参数直接传入。

```
class Mesh {
public:
    // mesh Data
    vector<Vertex>       vertices;
    vector<unsigned int> indices;
    vector<Texture>      textures;
    unsigned int VAO;

    // constructor
    Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture>
textures)
    {
        this->vertices = vertices;
        this->indices = indices;
        this->textures = textures;

        // now that we have all the required data, set the vertex buffers and its
attribute pointers.
        setupMesh();
    }
    ......
}
```

对于indices可以遍历face中的坐标传入。而纹理texture还需要进一步加工。

```
    Mesh processMesh(aiMesh *mesh, const Scene *scene){
        vector<unsigned int> indices;
        vector<Texture> textures;

        for(unsigned int i = 0; i < mesh->mNumFaces; i++)
        {
            aiFace face = mesh->mFaces[i];
            for(unsigned int j = 0; j < face.mNumIndices; j++){
                indices.push_back(face.mIndices[j]);
            }
        }
        std::map <std::string, aiMaterial*>temp = scene->mMaterial;
        aiMaterial* material = temp[(mesh->mMaterialIndex)];
        // 1. diffuse maps
        vector<Texture> diffuseMaps = loadMaterialTextures(material,
"texture_diffuse");
        textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
        // 2. specular maps
        vector<Texture> specularMaps = loadMaterialTextures(material,
"texture_specular");
        textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
        // 3. normal maps
        std::vector<Texture> normalMaps = loadMaterialTextures(material,
"texture_normal");
        textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());
        // 4. height maps
        std::vector<Texture> heightMaps = loadMaterialTextures(material,
"texture_height");
        textures.insert(textures.end(), heightMaps.begin(), heightMaps.end());
        int cnt=0;
        int cnt_i=0;
        // return a mesh object created from the extracted mesh data
        return Mesh(Vertices, indices, textures);
    }
```

加载纹理的函数为loadMaterialTextures(),其中有一个小技巧是每次都查看一下纹理是否已经被加载过，如果已经被加载过，那么直接使用即可，节约时间。

```
vector<Texture> loadMaterialTextures(aiMaterial *mat, string typeName)
    {
        vector<Texture> textures;
        if(mat->GetTextureCount.count(typeName)==0) return textures;
        for(unsigned int i = 0; i < mat->GetTextureCount[typeName]; i++)
        {
            string str = mat->GetTexture[typeName][i];
            // cout<<str<<endl;
            // check if texture was loaded before and if so, continue to next
iteration: skip loading a new texture
            bool skip = false;
            for(unsigned int j = 0; j < textures_loaded.size(); j++)
            {
                if(strcmp(textures_loaded[j].path.data(), str.c_str()) == 0)
                {
                    textures.push_back(textures_loaded[j]);
                    skip = true;
                    break;
                }
            }
            if(!skip)
            {   // if texture hasn't been loaded already, load it
                Texture texture;
                texture.id = TextureFromFile(str, this->directory);
                texture.type = typeName;
                texture.path = str;
                textures.push_back(texture);
                textures_loaded.push_back(texture);
            }
        }
        return textures;
    }
unsigned int TextureFromFile(const string path, const string &directory)
{
    string filename = directory + '/' + path;
    unsigned int textureID;
    glGenTextures(1, &textureID);

    int width, height, nrComponents;
    unsigned char *data = stbi_load(filename.c_str(), &width, &height,
&nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format,
GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);

        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_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        stbi_image_free(data);
    }
    else
    {
        std::cout << "Texture failed to load at path: " << path << std::endl;
        stbi_image_free(data);
    }

    return textureID;
}
```

至此，Model.h 已经实现完成了。下面给出Mesh类的具体实现。调用Draw()函数可以直接绘制。

```
class Mesh {
public:
    // mesh Data
    vector<Vertex>       vertices;
    vector<unsigned int> indices;
    vector<Texture>      textures;
    unsigned int VAO;

    // constructor
    Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture>
textures)
```

```
    {
        this->vertices = vertices;
        this->indices = indices;
        this->textures = textures;

        // now that we have all the required data, set the vertex buffers and its
attribute pointers.
        setupMesh();
    }

    // render the mesh
    void Draw(Shader &shader)
    {
        // bind appropriate textures
        unsigned int diffuseNr  = 1;
        unsigned int specularNr = 1;
        unsigned int normalNr   = 1;
        unsigned int heightNr   = 1;
        for(unsigned int i = 0; i < textures.size(); i++)
        {
            glActiveTexture(GL_TEXTURE0 + i); // active proper texture unit before
binding
            // retrieve texture number (the N in diffuse_textureN)
            string number;
            string name = textures[i].type;
            if(name == "texture_diffuse")
                number = std::to_string(diffuseNr++);
            else if(name == "texture_specular")
                number = std::to_string(specularNr++); // transfer unsigned int to
string
            else if(name == "texture_normal")
                number = std::to_string(normalNr++); // transfer unsigned int to
string
             else if(name == "texture_height")
                number = std::to_string(heightNr++); // transfer unsigned int to
string

            // now set the sampler to the correct texture unit
            glUniform1i(glGetUniformLocation(shader.ID, (name + number).c_str()),
i);
            // and finally bind the texture
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
        }

        // draw mesh
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, static_cast<unsigned int>(indices.size()),
GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);

        // always good practice to set everything back to defaults once
configured.
        glActiveTexture(GL_TEXTURE0);
    }
```

```
private:
    // render data
    unsigned int VBO, EBO;

    // initializes all the buffer objects/arrays
    void setupMesh()
    {
        // create buffers/arrays
        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);
        // load data into vertex buffers
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        // A great thing about structs is that their memory layout is sequential
for all its items.
        // The effect is that we can simply pass a pointer to the struct and it
translates perfectly to a glm::vec3/2 array which
        // again translates to 3/2 floats which translates to a byte array.
        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex),
&vertices[0], GL_STATIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned
int), &indices[0], GL_STATIC_DRAW);

        // set the vertex attribute pointers
        // vertex Positions
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
        // vertex normals
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Normal));
        // vertex texture coords
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, TexCoords));
        // vertex tangent
        glEnableVertexAttribArray(3);
        glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Tangent));
        // vertex bitangent
        glEnableVertexAttribArray(4);
        glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Bitangent));
        // ids
        glEnableVertexAttribArray(5);
        glVertexAttribIPointer(5, 4, GL_INT, sizeof(Vertex),
(void*)offsetof(Vertex, m_BoneIDs));

        // weights
        glEnableVertexAttribArray(6);
        glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, m_Weights));
        glBindVertexArray(0);
    }
};
```

摄像头系统

我们实现了两个摄像头系统，分别用于锁定车身和场景漫游，下面分别介绍

锁定车身摄像头类

摄像头设置相关

构造函数如下,初始化时给定一个位置，上向量和指向的目标点，然后计算摄像头的偏航角和俯仰角。

```
Camera(glm::vec3 position = glm::vec3(0.0f, 50.0f, 50.0f),
           glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f),
           glm::vec3 target = glm::vec3(0.0f, 0.0f, 0.0f)) : Zoom(ZOOM)
    {
        Position = position;
        WorldUp = up;
        glm::vec3 front = target - position;
        Front = glm::normalize(front);
        Right = glm::normalize(glm::cross(Front, WorldUp));
        Up = glm::normalize(glm::cross(Right, Front));
        if (front.x != 0)
        {
            Yaw = glm::degrees(glm::atan(front.z / front.x));
            if (front.x < 0)
            {
                Yaw += 180.0f;
            }
            Pitch = glm::degrees(glm::atan(front.y / (glm::sqrt(front.x * front.x
+ front.z * front.z))));
        }
        else
        {
            if (front.z > 0)
            {
                Yaw = 90.0f;
                Pitch = glm::degrees(glm::atan(front.y / (glm::sqrt(front.x *
front.x + front.z * front.z))));
            }
            else if (front.z < 0)
            {
                Yaw = 270.0f;
                Pitch = glm::degrees(glm::atan(front.y / (glm::sqrt(front.x *
front.x + front.z * front.z))));
            }
            else
            {
                Yaw = 0.0f;
                if (front.y > 0)
                {
                    Pitch = 90.0f;
                }
                else if (front.y < 0)
                {
                    Pitch = -90.0f;
                }
                else
                {
                    Pitch = 0.0f;
                }
            }
        }
        R = glm::sqrt((position.x - target.x) * (position.x - target.x) +
(position.z - target.z) * (position.z - target.z));
    }
```

此后所有的操作都在updateCameraVectors()函数的基础上实现，此函数的作用是根据偏航角和俯仰角计算前向量，上向量，右向量，所以我们只需要改变Yaw和Pitch就可以改变摄像头的整体位置和方向信息。

```
void updateCameraVectors()
    {
        // calculate the new Front vector
        glm::vec3 front;
        front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        front.y = sin(glm::radians(Pitch));
        front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        // std::cout<<front.x<<","<<front.y<<","<<front.z<<std::endl;
        Front = glm::normalize(front);
        // also re-calculate the Right and Up vector
        Right = glm::normalize(glm::cross(Front, WorldUp)); // normalize the
vectors, because their length gets closer to 0 the more you look up or down which
results in slower movement.
        Up = glm::normalize(glm::cross(Right, Front));
    }
```

实现速度变化时的视角变化

为了让加速和减速更真实，我们想实现一种加速时摄像头和车的距离的拉大的感觉，减速同理。按下 W 键时视角变大，S 键视角变小，什么都不按时恢复。

```
    void ProcessKeyboard(Camera_Movement direction, float deltaTime)
    {
        if (direction == FORWARD)
        {
            ZoomOut();
        }
        if (direction == BACKWARD)
        {
            ZoomIn();
        }
        if (direction == NO)
        {
            ZoomRecover();
        }
    }
    void ZoomIn()
    {
        if (Zoom >= 40.0f)
            Zoom -= 0.01f;
    }

    void ZoomOut()
    {
        if (Zoom <= 60.0f)
            Zoom += 0.02f;
    }

    void ZoomRecover()
    {
        if (Zoom < ZOOM)
            Zoom += ZOOM_SPEED / 2;
        if (Zoom > ZOOM)
            Zoom -= ZOOM_SPEED / 2;
    }
```

实现车辆转弯时摄像头缓动，造成漂移感

这主要就是生成lookat矩阵时的问题，只需要让车的前方与摄像头前向量水平面投影有角度差别时摄像头缓慢回正即可。

```

        glm::mat4 GetViewMatrix(Car car)
        {
            glm::vec3 Target = car.Position;
            float car_yaw = car.getYaw();
            glm::vec3 new_Position;
            if (car_yaw < Yaw)
            {
                if (Yaw - car_yaw <= 0.02f)
                {
                    Yaw = car_yaw;
                }
                else
                {
                    Yaw -= 0.02f;
                }
                updateCameraVectors();
            }
            else if (car_yaw > Yaw)
            {
                if (car_yaw - Yaw <= 0.02f)
                {
                    Yaw = car_yaw;
                }
                else
                {
                    Yaw += 0.02f;
                }
                updateCameraVectors();
            }
            glm::vec3 front_temp;
            front_temp.x = cos(glm::radians(Yaw));
            front_temp.y = 0.0f;
            front_temp.z = sin(glm::radians(Yaw));
            glm::vec3 Front_temp = glm::normalize(front_temp);
            new_Position = Target - R * Front_temp;
            new_Position.y = Position.y;
            Position = new_Position;

            return glm::lookAt(Position, Target, WorldUp);
        }
```

场景漫游摄像头

基本实现与上述摄像头相同，只是需要加入以下的交互设置，实现漫游场景。

```
    void ProcessKeyboard(TCamera_Movement direction, float deltaTime)
    {
        float velocity = MovementSpeed * deltaTime;
        if (direction == TFORWARD)
            Position += Front * velocity;
        if (direction == TBACKWARD)
            Position -= Front * velocity;
        if (direction == TLEFT)
            Position -= Right * velocity;
        if (direction == TRIGHT)
            Position += Right * velocity;
    }

    // processes input received from a mouse input system. Expects the offset
value in both the x and y direction.
    void ProcessMouseMovement(float xoffset, float yoffset, GLboolean
constrainPitch = true)
    {
        xoffset *= MouseSensitivity;
        yoffset *= MouseSensitivity;
        Yaw   += xoffset;
        Pitch += yoffset;

        // make sure that when pitch is out of bounds, screen doesn't get flipped
        if (constrainPitch)
        {
            if (Pitch > 89.0f)
                Pitch = 89.0f;
            if (Pitch < -89.0f)
                Pitch = -89.0f;
        }

        // update Front, Right and Up Vectors using the updated Euler angles
        updateCameraVectors();
    }

    // processes input received from a mouse scroll-wheel event. Only requires
input on the vertical wheel-axis
    void ProcessMouseScroll(float yoffset)
    {
        Zoom -= (float)yoffset;
        if (Zoom < 1.0f)
            Zoom = 1.0f;
        if (Zoom > 45.0f)
            Zoom = 45.0f;
    }
```

车辆运动系统

车辆基础设置

构造函数如下，同样是计算Yaw。

```
    Car(glm::vec3 position, glm::vec3 front) : MovementSpeed(0.0f),
RoundSpeed(90.0f)
    {
        Position = position;
        Front = front;
        if (front.x != 0)
        {
            Yaw = glm::degrees(glm::atan(front.z / front.x));
            if (front.x < 0)
            {
                Yaw += 180.0f;
            }
        }
        else
        {
            if (front.z > 0)
            {
                Yaw = 90.0f;
            }
            else if (front.z < 0)
            {
                Yaw = 270.0f;
            }
            else
            {
                Yaw = 0.0f;
            }
        }
        // updateFront();
    }
```

同样实现只需要改变Yaw即可改变车辆的所以位置信息

```
    void updateFront()
    {
        glm::vec3 front;
        front.x = cos(glm::radians(Yaw));
        front.y = 0.0f;
        front.z = sin(glm::radians(Yaw));
        Front = glm::normalize(front);
    }
```

车辆运动系统

为了模拟真实世界，设计了符合物理规律的运动系统，加速时，加速度与速度成负相关。无动力状态，车辆以很大的减速度减速。

```
    void ProcessKeyboard(D direction, float deltaTime, glm::vec3 Camera_Front)
    {
        if (direction == CAR_FORWARD)
        {
            if (MovementSpeed == 0.0f)
            {
                a = 500.0f;
            }
            else
            {
                a = 500.0f / (MovementSpeed);
                if (a < 0)
                    a = -a;
            }
            glm::vec3 temp = glm::vec3(Camera_Front.x, 0.0f, Camera_Front.z);
            Position += glm::normalize(temp) * (MovementSpeed * deltaTime + 0.5f *
a * deltaTime * deltaTime);
            MovementSpeed = MovementSpeed + a * deltaTime;
        }
        if (direction == CAR_BACKWARD)
        {
            if (MovementSpeed == 0.0f)
            {
                a = -30.0f;
            }
            else
            {
                a = 300.0f / (MovementSpeed);
                if (a > 0)
                    a = -a;
            }
            glm::vec3 temp = glm::vec3(Camera_Front.x, 0.0f, Camera_Front.z);
            Position += glm::normalize(temp) * (MovementSpeed * deltaTime + 0.5f *
a * deltaTime * deltaTime);
            MovementSpeed = MovementSpeed + a * deltaTime;
        }
        if (direction == CAR_NO)
        {
            if (MovementSpeed > 0.0f)
            {
                a = -20.0f;
                glm::vec3 temp = glm::vec3(Camera_Front.x, 0.0f, Camera_Front.z);
                Position += glm::normalize(temp) * (MovementSpeed * deltaTime +
0.5f * a * deltaTime * deltaTime);
                MovementSpeed = MovementSpeed + a * deltaTime;
                if (MovementSpeed < 0.0f)
                    MovementSpeed = 0.0f;
            }
            else if (MovementSpeed < 0.0f)
            {
                a = 20.0f;
                glm::vec3 temp = glm::vec3(Camera_Front.x, 0.0f, Camera_Front.z);
                Position += glm::normalize(temp) * (MovementSpeed * deltaTime +
0.5f * a * deltaTime * deltaTime);
                MovementSpeed = MovementSpeed + a * deltaTime;
                if (MovementSpeed > 0)
                    MovementSpeed = 0.0f;
            }
        }
        if (direction == CAR_LEFT)
        {
            if (MovementSpeed != 0.0f)
            {
                Yaw -= RoundSpeed * deltaTime;
                if (Yaw > 360.0f)
                {
                    Yaw -= 360.0f;
                }
            }
        }
        if (direction == CAR_RIGHT)
        {
            if (MovementSpeed != 0.0f)
            {
                Yaw += RoundSpeed * deltaTime;
                if (Yaw > 360.0f)
                    Yaw -= 360.0f;
            }
        }
        updateFront();
    }
```

场景搭建

天空盒绘制

绘制一个深度无限大的立方体作为天空盒，以下是天空盒的六个面坐标点

```
float skyboxVertices[] = {
    // positions
    -1.0f, 1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f, 1.0f, -1.0f,
    -1.0f, 1.0f, -1.0f,

    -1.0f, -1.0f, 1.0f,
    -1.0f, -1.0f, -1.0f,
    -1.0f, 1.0f, -1.0f,
    -1.0f, 1.0f, -1.0f,
    -1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f,

    1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,

    -1.0f, -1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, -1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f,

    -1.0f, 1.0f, -1.0f,
    1.0f, 1.0f, -1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
    -1.0f, -1.0f, 1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    -1.0f, -1.0f, 1.0f,
    1.0f, -1.0f, 1.0f};
```

通过loadCubemap函数绑定纹理到天空盒

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height,
&nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                         0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
data);
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " << faces[i] <<
std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    return textureID;
}
```

在绘制天空盒的时候，设置其深度为无限大，绘制完成后复原深度。

```
        glDepthFunc(GL_LEQUAL); // change depth function so depth test passes when
values are equal to depth buffer's content
        skyboxShader.use();
        view = glm::mat4(glm::mat3(camera.GetViewMatrix(car))); // remove
translation from the view matrix
```

```
        if(trave_camera){
            projection = glm::perspective(glm::radians(tcamera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
            view = tcamera.GetViewMatrix();
        }
        skyboxShader.setMat4("view", view);
        skyboxShader.setMat4("projection", projection);
        glBindVertexArray(skyboxVAO);
        glActiveTexture(GL_TEXTURE30);
        glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
        glDrawArrays(GL_TRIANGLES, 0, 36);
        glBindVertexArray(0);
        glDepthFunc(GL_LESS); // set depth function back to default
```

阴影渲染

```
        // 从光源角度渲染整个场景
        depthShader.use();
        depthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

        // 改变视口大小以便于进行深度的渲染
        glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);

        glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO); // 绑定 FrameBuffer

        // 使用深度shader渲染生成场景

        glClear(GL_DEPTH_BUFFER_BIT);

        // glCullFace(GL_FRONT);
        renderRaceTrack(Model_track, depthShader);

        rendercar(Model_car, depthShader);
        // glCullFace(GL_BACK);

        // glActiveTexture(GL_TEXTURE29);
        // glBindTexture(GL_TEXTURE_2D,cubemapTexture);
        // renderScene(depthShader);

        glBindFramebuffer(GL_FRAMEBUFFER, 0); // 绑定回默认

        // 复原视口
        glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

在绘制好阴影贴图后，正常渲染到屏幕。 使用的着色器如下：(需要能计算阴影)

```
#version 330 core
out vec4 FragColor;
```

```
in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;

uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;

uniform vec3 lightPos;
uniform vec3 viewPos;

uniform bool shadows;

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 as coords)
    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)
    {
        for(int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) *
texelSize).r;
            shadow += currentDepth - bias > pcfDepth  ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;

    // Keep the shadow at 0.0 when outside the far_plane region of the light's
frustum.
    if(projCoords.z > 1.0)
        shadow = 0.0;

    return shadow;
}
```

```
void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(0.8);
    // Ambient
    vec3 ambient = 0.2 * color;
    // Diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // Specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // Calculate shadow
    float shadow = shadows ? ShadowCalculation(fs_in.FragPosLightSpace) : 0.0;
    shadow = min(shadow, 0.75); // reduce shadow strength a little: allow some
diffuse/specular light in shadowed regions
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

    FragColor = vec4(lighting, 1.0f);
}
```