FPS Training Arena

目录

1.基本采分点介绍

1.1 基本采分点简介

1. [10 分]基于 OpenGL，具有基本体素（立方体、球、圆柱、圆锥、多面棱柱、多面棱台）的建模表达能力；
2. [10 分]具有基本三维网格导入导出功能（建议 OBJ 格式）；
3. [15 分]具有基本材质、纹理的显示和编辑能力；
4. [10 分]具有基本几何变换功能（旋转、平移、缩放等）；
5. [15 分]基本光照明模型要求，并实现基本的光源编辑（如调整光源的位置，光强等参数）；
6. [15 分]能对建模后场景进行漫游如 Zoom In/Out， Pan, Orbit, Zoom To Fit 等观察功能。
7. [15 分]Awesomeness 指数：展示项目本身所独有的炫酷特点，包括但不限于有感染力的视觉特效。

第 1、2、3 点可以从以下截图看出，本程序能够导入.obj 文件，并且正确的绘制出体素和纹理来。在导入的过程中判断每一个 mesh 是否有贴图，包括漫反射贴图，法线贴图，镜面反射贴图，若没有则按照颜色绘制。

第 4 点在运行时可以用 wsad 对主角进行操作，能让主角模型进行旋转、平移等操作，敌人也会自动在场地中进行漫游，也可以有基本的几何变换。

第 5 点可以从以下截图看出，场景中有光照效果，具体是由多个点光源叠加计算得到的，模型为 Blinn-Phong。程序中也能够调整光源位置、光强等参数。

第 6 点可以在运行时按 c 键切换到自由视角，进行场景漫游。

第 7 点放到下一节一起介绍。

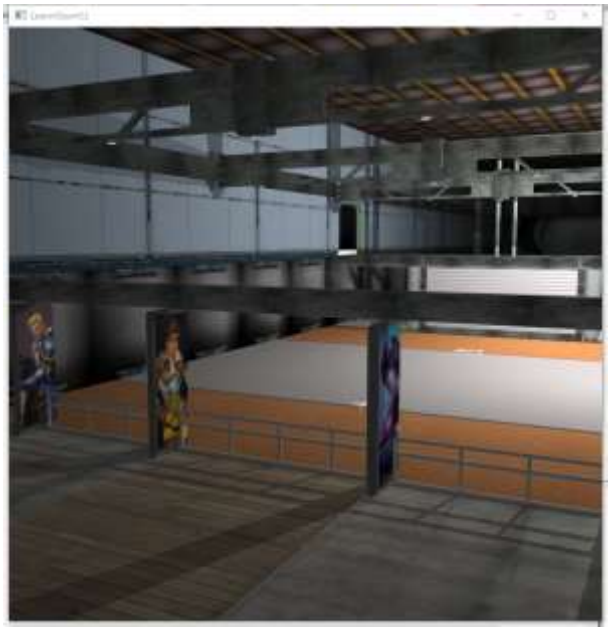

Figure 1 总体效果

1.2 加分项介绍

（1）实时的 OBB 碰撞检测。

整个程序分为两个部分，一个是图像渲染和显示的部分，另一个是游戏逻辑部分。游戏逻辑部分管理了一个物理世界，物理世界由一系列的 OBB 组成。下图即为 OBB 可视化(红色的框)后看到的效果。

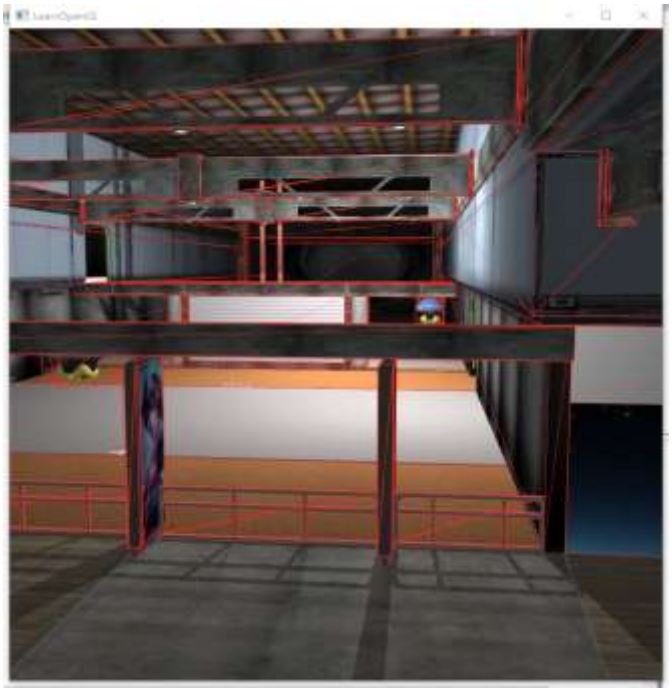

Figure 2 可视化物理世界

每个 OBB 包围了场景模型中个每一个 mesh 和敌人、主角的整个模型。在运动时进行 OBB-OBB 的碰撞检测。具体的算法实现为分离轴定理。

由 objects.cpp 中三个函数实现：

```
1.  bool detect_obb(Object* a, Object* b);
2.  void get_interval_obb(Object* a, glm::vec3 axis, float& min, float& max);
3.  bool axis_equal_obb(glm::vec3 a, glm::vec3 b);
```

detect_obb()为第一层调用，返回 a\b 两个物体是否相交。

get_interval_obb()为第二层调用，返回一个物体的 OBB 上八个顶点与一个轴相交的最小位置和最大位置。

axis_equal_obb()为第一层调用中计算两条轴是否平行。

（2）多个点光源叠加实现 Shadow Mapping

点光源从之前进行光照渲染的点光源选取。在点光源位置渲染场景得到深度贴图，传入绘制场景的 Shader 中，判断光源能否看到这个位置进行阴影的绘制。由于是点光源，绘制的贴图为 Cube_Map。具体实现如下。相对的，渲染时 shader 代码比较简单，只是判断距离大小，就不放出来了。最后对计算得到的阴影加权平均，就可以得到比较好的多个点光源的阴影。

```
1.      float far_plane = 200.0f;
2.      float near_plane = 1.0f;
3.      const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
4.      unsigned int depthMapFBO;
5.      glGenFramebuffers(1, &depthMapFBO);
6.
7.      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
8.      for (int i = 0; i < light.meshes.size(); i++){
9.          unsigned int &temp = light.meshes[i].depthCubeMap;
10.         glGenTextures(1, &temp);
11.         glBindTexture(GL_TEXTURE_CUBE_MAP, temp);
12.         for (int j = 0; j < 6; j++){
13.             glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + j, 0, GL_DEPTH_COMPONENT, SH
    ADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
14.         }
15.         glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
16.         glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
17.         glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
18.         glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
19.         glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
20.
21.         glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
22.         glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, temp, 0);
23.         glDrawBuffer(GL_NONE);
24.         glReadBuffer(GL_NONE);
25.         glBindFramebuffer(GL_FRAMEBUFFER, 0);
26.
27.         glm::mat4 shadowProj;
28.         std::vector<glm::mat4> shadowTransforms;
29.
30.         float aspect = (float)SHADOW_WIDTH / (float)SHADOW_HEIGHT;
31.         shadowProj = glm::perspective(glm::radians(90.0f), aspect, near_plane, far_pla
    ne);
32.
33.         glm::vec3 lightPos = light.meshes[i].vertices[0].Position;
```

```
34.         shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos + glm::
    vec3(1.0, 0.0, 0.0), glm::vec3(0.0, -1.0, 0.0)));
35.         shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos + glm::
    vec3(-1.0, 0.0, 0.0), glm::vec3(0.0, -1.0, 0.0)));
36.         shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos + glm::
    vec3(0.0, 1.0, 0.0), glm::vec3(0.0, 0.0, 1.0)));
37.         shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos + glm::
    vec3(0.0, -1.0, 0.0), glm::vec3(0.0, 0.0, -1.0)));
38.         shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos + glm::
    vec3(0.0, 0.0, 1.0), glm::vec3(0.0, -1.0, 0.0)));
39.         shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos + glm::
    vec3(0.0, 0.0, -1.0), glm::vec3(0.0, -1.0, 0.0)));
40.
41.         glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
42.         glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
43.         glClear(GL_DEPTH_BUFFER_BIT);
44.         simpleDepthShader.use();
45.         for (int i = 0; i < 6; i++){
46.             string num = std::to_string(i);
47.             string name = "shadowMatrices[" + num + "]";
48.             simpleDepthShader.setMat4(name.c_str(), shadowTransforms[i]);
49.         }
50.         simpleDepthShader.setFloat("far_plane", far_plane);
51.         simpleDepthShader.setVec3("lightPos", lightPos);
52.         glm::mat4 model = glm::mat4(1.0f);
53.         simpleDepthShader.setMat4("model", model);
54.         //render scene
55.         object.Draw(simpleDepthShader);
56.
57.         glBindFramebuffer(GL_FRAMEBUFFER, 0);
58.     }
```

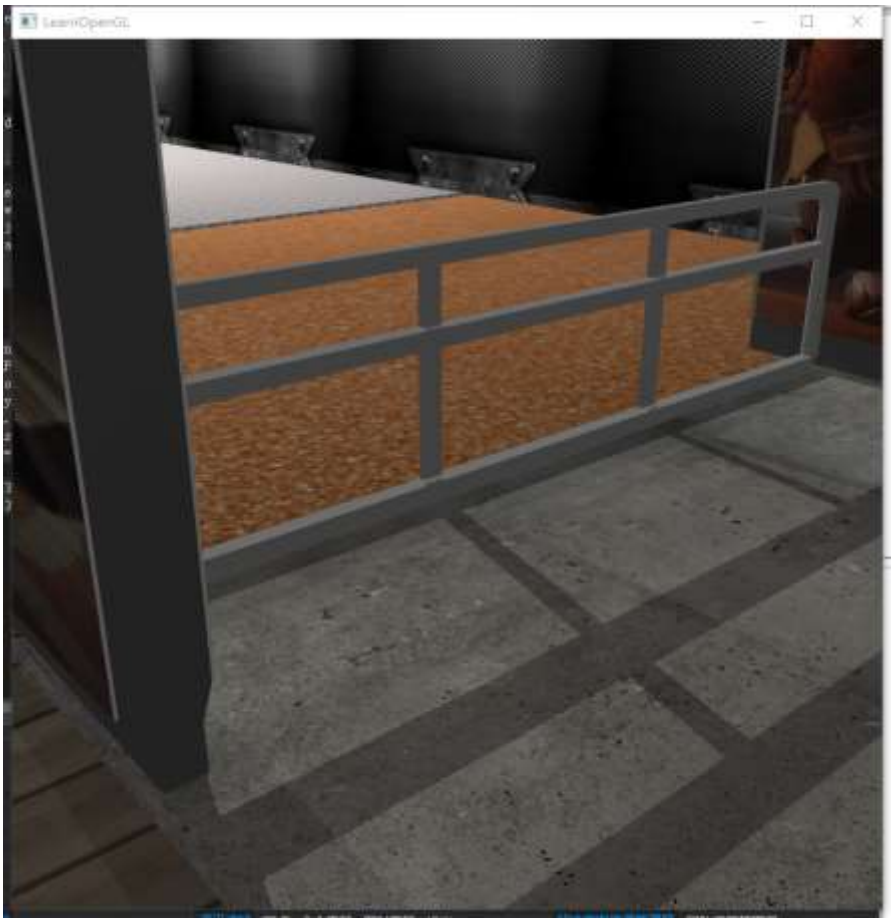

Figure 3 两个点光源阴影叠加

（3）主角射击时 picking 的实现与 picking 应用。

作为一个 FPS 游戏，射击时仅仅有 Ray-OBB 精度的判断是不够的，但全部遍历三角片元进行拾取又比较费。因此在第一层 Ray-OBB 相交检测之后，选出待定项，对待定项进行三角片元级别的检测。具体由以下函数实现。其中射线与三角片元的相交判断函数参考了 Fast, Minimum Storage Ray Triangle Intersection。

```
1.  ///////////
2.  //ray & obb
3.  ///////////
4.  bool detect_ray_obj_obb(glm::vec3 ray_p, glm::vec3 ray_d, Object* object);
5.  bool detect_ray_ene_obb(glm::vec3 ray_p, glm::vec3 ray_d, Object* object);
6.  //ray & plane
7.  //ray = ray_p + t * ray_d
8.  //plane : plane_d * x = plane_d * plane_p0
9.  glm::vec3 intersect_ray_obb(glm::vec3 ray_p, glm::vec3 ray_d, glm::vec3 plane_p, glm::
    vec3 plane_d);
10. //two line segments' interval
11. bool interval_v3_v3(glm::vec3 a, glm::vec3 b, glm::vec3 c, glm::vec3 d);
```

```
12. bool interval_f_f(float ta, float tb, float tc, float td);
13. float distance(glm::vec3 a, glm::vec3 b);
14.
15. /////////////////
16. //ray & triangles
17. /////////////////
18. bool detect_ray_enemy_tri(glm::vec3 ray_p, glm::vec3 ray_d, Object* object, float* min
    );
19. bool detect_ray_obj_tri(glm::vec3 ray_p, glm::vec3 ray_d, Object* object, float* min);

20. // Determine whether a ray intersect with a triangle
21. bool intersect_ray_tri(glm::vec3 orig, glm::vec3 dir, glm::vec3 v0, glm::vec3 v1, glm:
    :vec3 v2, float* t, float* u, float* v);
```

实现 Ray-OBB 之后可以用在主角移动的重力效果上。若从主角脚下发出多条射线，判断射线是否与地面相交，若相交，与地面距离多少，可以用来判断主角当前是下落还是站立状态。

（4）敌人自动漫游。

敌人自动漫游基于 OBB-OBB 的碰撞检测，漫游时检测与环境 OBB，非自身敌人 OBB，主角 OBB 进行碰撞检测。敌人要能自主寻找一个可以前进的方向，而不是自己去乱撞，发现碰撞了再弹回来。因此我实现了一个空的物体类，它仅在物理世界可见，是敌人进行下一步动作的“前驱”，只有这个空的“敌人”确认可以行动，敌人才会继续行动。

（5）敌人死亡的碎裂特效。

敌人处于死亡状态时，会有 100 帧的死亡动画。倒计时由程序进行，死亡动画由单独的 death_shader(vertex shader)计算，具体实现是让每一个三角片元向着三角形平均法向量的方向飞出，加一个重力加速度和地板的位置，就可以实现碎一地的效果，shader 代码如下:

```
1.   #version 330 core
2.  layout (location = 0) in vec3 aPos;
3.  layout (location = 1) in vec3 aNormal;
4.  layout (location = 2) in vec2 aTexCoords;
5.  layout (location = 3) in vec3 avgNormal;
6.
7.  out vec2 TexCoords;
8.  out vec3 Normal;
9.  out vec3 FragPos;
```

```
10.
11. uniform mat4 model;
12. uniform mat4 view;
13. uniform mat4 projection;
14. uniform float deathcount;
15. uniform float g;
16. uniform float p;
17.
18. void main()
19. {
20.     vec3 position = vec3(deathcount,deathcount,deathcount);
21.     position = aPos + position * avgNormal;
22.     position = position - vec3(0,0.5 * g * deathcount * deathcount,0);
23.     if(position.y < p)
24.         position.y = p;
25.     TexCoords = aTexCoords;
26.     gl_Position = projection * view * model * vec4(position, 1.0);
27.
28.     Normal = mat3(transpose(inverse(model)))*aNormal;
29.     FragPos =vec3(model* vec4(aPos,1.0f));
30. }
```

2. 系统设计说明

2.1 总体层次结构说明

没有被抽象成类的部分有：

天空盒的绘制，主场景的渲染绘制。

被抽象成类的部分共有以下几个部分的代码构成：

1、 摄像机类

抽象了摄像机的性质，管理了摄像机位置，方向，上方向量。实现了自由视角下的键盘、鼠标移动的相应几何变换。当然也可以绑定到一个物体上。创建摄像机只要创建一个类成员即可。

```
1.  class MyCamera{
2.  public:
```

```
    glm::vec3 cameraPos;
    glm::vec3 cameraFront;
    glm::vec3 cameraUp;
    glm::vec3 WorldUp;
    glm::vec3 cameraRight;
    float lastX, lastY;
    float yaw, pitch;
    float MouseSensitivity;
    float CameraSensitivity;
    float cameraSpeed; // adjust accordingly
    bool firstMouse = true;
    MyCamera();
    void KeyBoardMovement(int Direction);
    void MouseMovement(double xpos, double ypos);
};
```

2、Mesh 类

管理每一个 mesh，保存了 VAO,VBO,EBO,顶点信息，材质，贴图信息。在初始化的时候是在 Mesh 类中初始化 VAO 等并保存的，之后绘制(Draw()函数)直接激活相应的 VAO,VBP,EBO 即可。

```
struct Vertex{
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
    glm::vec3 avgNormal;
};

struct Texture{
    unsigned int id;
    string type;
    string path;
};

//普适性更强一点 可以适应没有贴图 只有颜色的模型
struct Material{
    //color of ambient
    glm::vec3 ambient;
    //color of diffuse;
    glm::vec3 diffuse;
    //color of specular;
    glm::vec3 specular;
    //color of emission
```

```
    glm::vec3 emissive;

    float shininess;
    vector<Texture> textures;
    bool text;
};

class MyMesh{
public:
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    Material material;
    unsigned int depthCubeMap = 0;

    MyMesh(vector<Vertex> vertices, vector<unsigned int> indices, Material material);

    void Draw(MyShader& shader);
    unsigned int VAO, VBO, EBO;
    void setupMesh();
};
```

3、Model 类

管理每一个模型，保存了该模型下的 mesh 数组，导入 obj 文件是从这一层开始的。绘制的时候调用 Draw()函数，会调用相应 mesh 的 Draw()函数绘制。

```
class MyModel{
public:
    int initialization;

    /*  函数   */
    MyModel();
    MyModel(const MyModel& a);
    MyModel(string path, bool gamma = false);
    void Draw(MyShader& shader);
    float minx, maxx, miny, maxy, minz, maxz;
    /*  模型数据  */
    vector<MyMesh> meshes;
    vector<Texture> textures_loaded;//用来统计这个重复载入的纹理
    string directory;
    bool gammaCorrection;

    /*  函数   */
```

```
    void loadModel(string path);
    void processNode(aiNode *node, const aiScene *scene);
    MyMesh processMesh(aiMesh *mesh, const aiScene *scene);
    vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type
, string typeName);
    int TextureFromFile(const char* path, const string& directory);

    //light
    void SetLight(MyShader shader);
};
```

4、Shader 类

读入几何着色器(点光源万向阴影贴图)、片段着色器、顶点着色器。

```
class MyShader{
public:
    GLint ID;

    // 构造器读取并构建着色器
    MyShader(const GLchar* VertexShaderPath, const GLchar* FragmentShaderPath);
    MyShader(const char* vertexPath, const char* fragmentPath, const char *geometryPat
h);

    // 使用/激活程序
    void use();
    // uniform 工具函数
    void setBool(const std::string name, bool value) const;
    void setInt(const std::string name, int value) const;
    void setFloat(const std::string name, float value) const;
    void setMat4(const std::string name, glm::mat4 value) const;
    void setVec3(const std::string name, glm::vec3 pos) const;
    void setVec4(const std::string name, glm::vec4 pos) const;
};
```

5、Objects 类

Objects 是物理世界的基础，管理了每一个物体的模型、位置、朝向、OBB 包围盒、状态等基本信息。模型采用指针保存，多个物体能够共享一个模型，即可以重复使用已经导入的 Model 类，绘制时只需要修改物体的 model 矩阵。并实现了基础的绘制函数。场景作为普通的静态 Objects 导入。

```
class Object{
```

```
public:
    int id;
    string name;
    MyModel* pModel;
    int state;
    bool test_mode;

    //model matrix
    glm::mat4 model_matrix;

    //model position
    glm::vec3 position;
    //model direction
    glm::vec3 front;
    glm::vec3 up;
    float lastX, lastY;
    float yaw, pitch;

    //obb detection
    glm::vec3 vertex[8];
    glm::vec3 vertex_ini[8];
    glm::vec3 surface[3];
    int obb_dirty;
    void obb_init();
    void obb_update();

    //obb draw test box
    unsigned int VAO, VBO;
    void draw_test_obb(glm::mat4& model_matrix, MyShader& tShader, MyCamera& camera);

    //normalized scale
    float scale;

    //default
    Object();
    //use the whole obj file to create a object
    Object(string path, int id);
    //use the MyModel* to create a object
    Object(MyModel* pModel, int id, bool test_m);
    ~Object();


    //test
```

```
    float t = 0;
    glm::vec3 cool;

    virtual void Draw(MyShader& omyshader, MyShader& tShader, MyCamera& camera);
};
```

以 Object 为基类，派生出两个特殊的物体类型，Hero 和 Enemy 类。Hero 类绑定了一个内置的摄像机，还有独立的移动函数；Enemy 有独立的构造函数，能够构造一个只有 OBB 信息的空对象，用来找到前进的方向(敌人自动漫游的实现)，并且有单独的死亡动画绘制和生命值、受击函数等。

```
class Hero :public Object{
public:
    MyCamera camera;
    float heroSpeed;

    Hero(string path);

    glm::vec3 cam_pos;
    glm::vec3 box_c;
    void update_cam_pos();
    /*process keyboard and mouse input
    update model_matrix
    update camera*/
    void move(GLFWwindow *window, double xpos, double ypos);
    void forward();
    void backward();
    void leftward();
    void rightward();
    void turn(double xpos, double ypos);

    bool firstMouse = true;
    bool stand = true;
    void Draw(MyShader& omyshader, MyShader& tShader, MyCamera& camera, MyMo
del* light);

};

class Enemy :public Object{
public:
    bool live, move;
    float dead_counter;
    int hp;
```

```
    float speed;
    int angle;
    Enemy(MyModel* model, int id);
    Enemy(const Enemy* f);
    void Draw(MyShader& omyshader, MyShader& deathshader, MyShader& tShader,
MyCamera& camera, MyModel* light);
    bool hit();
    bool dead_animation();
};
```

6、Scene 类

管理了整个物理世界，有含有所有 Object 的指针数组，管理灯光的灯光指针数组，英雄指针和敌人指针数组。并实现了场景中的碰撞管理、英雄和敌人的逐帧更新、射击事件的管理和实现，OBB 的生成、可视化绘制等功能。是一个最上层的管理中心。

关于 OBB 生成，由于场景是一整个独立的 OBJ 文件，Scene 采用了独立的一套系统将原 OBJ 的每一个 mesh 节点作为一个 object 管理。

```
class Scene{
public:
    Scene();
    //input:
    Scene(string file_name);
    ~Scene();

    //input: path to obj file
    vector<Object*> objects;
    void add_object(string file_name);

    //input: light model
    vector<MyModel*> lights;
    void add_light(string file_name);

    //input: enemy model
    MyModel* enemy_model;
    vector<Enemy*> enemys;
    void init_enemy_model(string file_name);
    void add_enemy();
    void enemy_update();

    //input: hero model
```

```
    Hero* hero;
    void add_hero(string file_name);

    //input: map model: the difference is that extract each node with mesh(es) as a model
    void add_map(string file_name, bool tm);
    void processNode(aiNode *node, const aiScene *scene, string directory, bool tm);

    //draw
    void Draw(MyShader& obj_shader, MyShader& deathshader, MyShader& test_Shader, MyShader& light_shader, MyCamera& camera);

    //hero controller
    glm::vec3 old_position;
    glm::vec3 old_front;
    glm::vec3 old_camera_front;
    glm::vec3 old_camera_position;
    glm::vec3 old_vertex[8];
    glm::vec3 old_surface[3];
    void hero_move(GLFWwindow *window, double xpos, double ypos);
    bool hero_move_is_collide(void);
    void hero_move_store(void);
    void hero_move_restore(void);

    void hero_shoot(void);

    void hero_update(void);
    float drop_t;


    //test
    int collision_obj;
    int hit_obj;
    vector<int> tobe;
};
```

7、总结：

整个系统采用层次设计，从底层的 mesh 抽象，model 管理 mesh, object 管理 model，scene 管理 object 和实现物理世界。最后的绘制考虑到效率，独立在这个系统之外，直接绘制一整个场景的 obj。