



GIS 图形算法基础课程大作业

《CSU 校徽收集者》3D 游戏开发技术报告

班级： 地信 1801

小组成员表

刘乐	8211180103
孙瑞阳	8211180123
杨杰	8211180107
李红	8211180125

指导： 邓 浩

2020 年 12 月 29 日

目录

1 概述.....	1
1.1 游戏简介.....	1
1.2 功能关键词.....	1
2 开发环境.....	1
2.1 开发环境.....	1
2.2 外部依赖库.....	1
3 玩法介绍.....	2
4 开发实现.....	3
4.1 类介绍.....	3
4.2 重要功能实现.....	3
4.2.1 碰撞检测.....	3
4.2.2 粒子系统.....	7
4.2.3 鹰眼地图.....	8
4.2.4 透明自转光照 logo	10
4.2.5 天空盒及立方体贴图.....	11
5 遇到的问题及解决.....	13
6 后续版本展望.....	14

1 概述

1.1 游戏简介

本游戏基于 OpenGL 实现。该游戏具有两个模式，分别为“收集”模式和“迷宫”模式。收集模式需控制角色前往地图中随机出现的 logo 进行收集，集齐 5 个即赢得游戏；迷宫模式需控制角色前往地图终点即可获胜。

该游戏地图可自定义，地图文件及规则存放于“resource/maps”文件夹中，玩家可按照提示通过记事本等文本编辑器对地图进行自定义。

1.2 功能关键词

摄像机、天空盒、自转透明光照 logo、碰撞检测、跳跃重力模拟、鹰眼地图、粒子特效。

2 开发环境

2.1 开发环境

Platform	Windows 10 x64
IDE	Visual Studio 2017
Graphics rendering	OpenGL
Language	C++

2.2 外部依赖库


当前项目主要使用的库如下：

名称	功能
glfw	创建 OpenGL 上下文，定义窗口参数以及处理用户输入
glad	管理 OpenGL 的函数指针
stb_image.h	加载纹理及现纹理映射等
glm	用于向量、矩阵运算

3 玩法介绍

本《CSU 校徽收集者》3D 游戏共分为两个游戏模式：“校徽收集”模式和“迷宫”模式。玩家进入运行主界面后可以自由通过键盘按键选择游戏模式。

“收集”模式中，玩家需通过键盘按键控制角色方向及跳跃等动作，使得玩家能够前往地图中随机出现的 CSU 校徽 logo 处，通过触碰 CSU 校徽 logo 进行收集，在此游戏模式下，若玩家集齐 5 个 CSU 校徽 logo 即赢得游戏，当前已收集校徽显示在左上角。具体操作如下表格所示：

	角色向前: W 键 角色向后: A 键 角色向左: S 键 角色向右: D 键 角色跳跃: 空格键 返回主界面: Esc 键 鼠标脱离\捕获: Tab
---	--

“迷宫”模式中，玩家需通过键盘按键控制角色前进方向，在右上角鹰眼地图中玩家可以查看整个游戏世界平面地图、当前角色所在位置以及终点位置，控制角色前往地图终点即可获胜，具体操作如下表格所示：

	角色向前: W 键 角色向后: A 键 角色向左: S 键 角色向右: D 键 角色跳跃: 空格键 返回主界面: Esc 键 鼠标脱离\捕获: Tab
--	--

4 开发实现

4.1 类介绍

类名称	主要功能
立方体类 Cube	该类主要用于绘制地图方块以及天空盒
校徽 logo 类 CSUlogo	该类用于绘制各类和 logo 相关的图案
方形类 Square	该类主要用于绘制鹰眼地图
资源管理类 resource_manager	该类主要对纹理、游戏地图、着色器进行统一管理
纹理工具类 Texture	该类用于读取 2D 纹理、立方体贴图纹理
着色器类 shader	该类用于读取构造着色器
摄像机类 camera	该类用于控制视点移动
物理引擎类 PhysicsEngine	该类实现碰撞检测以及模拟物理跳跃及重力
游戏地图类 GameMap	该类用于读取地图数组以及其他地图相关功能
游戏场景绘制类 GameSceneRender	该类对游戏场景绘制进行统一管理控制
粒子系统类 ParticleSystem	该类模拟了粒子系统并产生粒子数组

4.2 重要功能实现

4.2.1 碰撞检测

本游戏所实现碰撞检测及处理机制在 AABB 碰撞的基础上针对问题特点进行了特殊化。因为本游戏将世界抽象成离散立方体，立方体边长为 1，且中心都处于整数格点上。

(1) 粗略脚定位

首先获取角色的脚所在方格，可以简单地通过对坐标四舍五入取整得到。

```

1. glm::ivec3 PhysicsEngine::getLocation(glm::vec3 pos) {
2.     glm::ivec3 location;
3.     location.x = (int)(pos.x + 0.5f);
4.     location.y = (int)(pos.y - roleHeight + 0.6f); //稍高一些，防止跨方格时
    “抖动感”
5.     location.z = (int)(pos.z + 0.5f);
6.     return location;

```

```
7. }
```

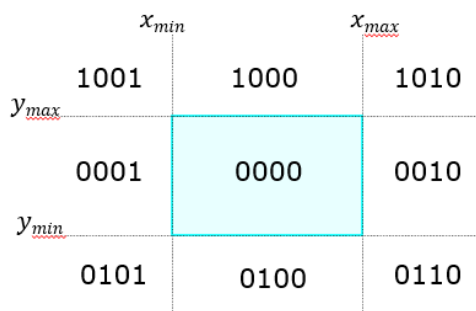
(2) 邻域可进入性判断

然后判断当前方格的八邻域内，哪些可进入，即八邻域的方块哪些比脚所在方块矮。将结果存储在 bool 数组里。

```
1. void PhysicsEngine::updateNeighbour(glm::ivec3 loc)
2. {
3.     int* currentP = &map[loc.z * mapx + loc.x];
4.     for (int i = 1; i < 10; i++) {
5.         bool result = true;
6.         switch (i) {
7.             case 1: //左上角(LT)
8.                 if (loc.x == 0 || loc.z == mapz - 1)
9.                     result = false;
10.                 if (*(currentP + mapx - 1) > loc.y)
11.                     result = false;
12.                 break;
13. ....
14.             case 9: //右下角(RB)
15.                 if (loc.x == mapx - 1 || loc.z == 0)
16.                     result = false;
17.                 if (*(currentP - mapx + 1) > loc.y)
18.                     result = false;
19.                 break;
20.             default:
21.                 result = true;
22.         }
23.         accessibleNei[i] = result;
24.     }
25. }
```

(3) 精细脚定位

将 XZ 平面上的每个方格都可再划分为 8 个区域，中间区域是 0.6*0.6 的正方形。该划分是为了解决视点太过贴近障碍物以及偶现的穿模问题。当前方不可进入时，则 1000，1001，1010 的区域均不可进入，以此类推。



```

1. // 八区域编码
2. int PhysicsEngine::encode(glm::vec3 pos)
3. {
4.     auto loc = getLocation(pos);
5.     int c = 0x0000;
6.     if (pos.x < loc.x - 0.3) c |= XL;
7.     if (pos.x > loc.x + 0.3) c |= XR;
8.     if (pos.z < loc.z - 0.3) c |= ZB;
9.     if (pos.z > loc.z + 0.3) c |= ZT;
10.    return c;
11. }

```

(4) XZ 平面碰撞判断及处理

依据之前的精定位以及对邻域的可进入性的判断,对 XZ 平面上的碰撞进行检测及处理。当边邻域不可进入时,对应的精定位的边同样不可进入,若进入,则将其移到 0x00 区域的边缘。对待角邻域的处理则还需判断当前 X、Z 哪一分量更大,然后对较小分量进行处理。

```

1. void PhysicsEngine::testXZCollision(glm::vec3 &pos, glm::ivec3 loc) {
2.     code = encode(pos);
3.     if (code == 0x0000) {
4.         return;
5.     }
6.     if (XL & code && !accessibleNei[4]) {
7.         pos.x = loc.x - 0.3;
8.     }
9.     if (XR & code && !accessibleNei[6]) {
10.        pos.x = loc.x + 0.3;
11.    }
12.    if (ZB & code && !accessibleNei[8]) {
13.        pos.z = loc.z - 0.3;
14.    }
15.    if (ZT & code && !accessibleNei[2]) {

```

```

16.     pos.z = loc.z + 0.3;
17. }
18. // 进入四个角
19. if (code == 0x1001 && !accessibleNei[1]) {
20.     if (accessibleNei[2] || accessibleNei[4])
21.         return;
22.     if (accessibleNei[2] && accessibleNei[4]) {
23.         if (abs(pos.z - loc.z) < abs(pos.x - loc.x))
24.             pos.z = loc.z + 0.3;
25.         else
26.             pos.x = loc.x - 0.3;
27.         return;
28.     }
29.     pos.x = loc.x - 0.3;
30.     pos.z = loc.z + 0.3;
31. }
32. ....
33. }

```

(5) Y 方向碰撞判断及处理

由于角色需要跳跃，并且受到重力作用，即考虑到角色的三维运动。因此还需对 Y 方向进行碰撞检测。该过程较简单，仅需要判断角色的脚的高度是否高于地面高度即可。若小于地面高度，则将其 Y 方向速度重置为 0，并取消跳跃状态。

```

1. if (newPos.y < y) {
2.     if (vy < 0.0f) {
3.         newPos.y = lastPos.y;
4.         newLoc.y = lastLoc.y;
5.         vy = 0.0f;
6.         isJumping = false;
7.     }
8. }

```

(6) 跳跃及重力模拟

模拟重力，只需要每一帧对角色的 Y 方向速度及 Y 位置进行更新即可。模拟跳跃，即在跳跃的瞬间给角色一向上的初速度即可实现。

```

1. //重力
2. newPos.y += (vy*dt + GravityAcceler * dt * dt / 2);

```



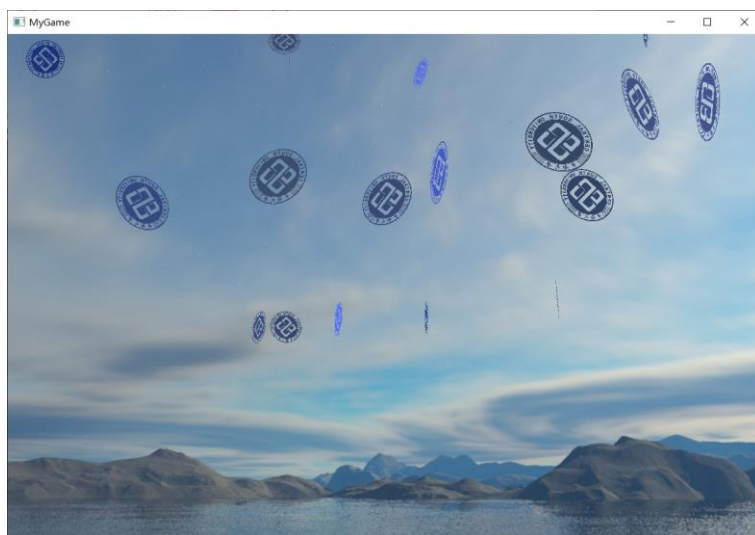
```

3. vy += GravityAcceler * dt;
4. //跳跃
5. if (!isJumping)
6.     vy += JumpInitialSpeed;
7. isJumping = true;

```

4.2.2 粒子系统

为了在收集到图标时，给玩家视觉反馈，我们在玩家收集到校徽后在校徽的位置生成若干随机方向的小校徽，随后，这些校徽做斜上抛运动四散开来，并随着时间流逝消亡。特效如图。



(1) 定义粒子属性

首先需要定义粒子的结构体。粒子有其位置、速度、大小、旋转角度、寿命、年龄等等，但其中每个粒子有差异的是位置、速度、旋转角度、寿命。因此只需要存储这些属性以节省空间。

```

1. struct Particle
2. {
3.     glm::vec3 position;
4.     glm::vec3 veclocosity;
5.     float life;
6.     int angle;
7. };

```

(2) 初始化参数并产生随机粒子

随机产生指定数量的随机方向、随机寿命以及随机角度的粒子，然后使用

动态数组进行保存。

```

1. void ParticleSystem::init(int count, glm::vec3 basePos)
2. {
3.     particles.clear();
4.     ptlCount = count;
5.     age = 0;
6.     srand(unsigned(time(0)));
7.     for (int i = 0; i < ptlCount; i++) {
8.         Particle tmp = {
9.             basePos,    //Position
10.            glm::vec3(((rand() % 20) - 10.0f) / 10,((rand() % 40)) / 10,((
            rand() % 20 - 10.0f)) / 10),//velocity
11.            1.0 + 0.1 * (rand() % 10), //Life
12.            rand() % 180    //angle
13.        };
14.        particles.push_back(tmp);
15.    }
16. }

```

(3) 每帧更新粒子状态

每一帧需要对粒子的位置、速度、旋转角度、年龄进行更新，并将消亡的粒子从粒子数组中移除。

```

1. void ParticleSystem::updateStatus(float dt) {
2.     age+=dt;
3.     for (int i = 0; i < particles.size(); i++) {
4.         //更新位置坐标
5.         particles[i].position += particles[i].veclocity * dt;
6.         particles[i].veclocity += gravity * dt;
7.         //更新旋转角度
8.         particles[i].angle = particles[i].angle++ % 360;
9.         //判断消亡
10.        if (age > particles[i].life) {
11.            particles.erase(particles.begin() + i);
12.        }
13.    }
14. }

```

4.2.3 鹰眼地图

由于迷宫具有一定难度，如若没有地图指示，很难走出迷宫。因此需要绘制

鹰眼地图，降低玩家的通关难度，提升游戏体验。



(1) 构建正方形类

首先需要构造一个能绘制正方形的类，该类存储正方形的顶点，并生成顶点缓存对象。该部分代码见项目 Square.h。

(2) 根据传入位置绘制小方块

先对其进行平移，确定根据地图的 XZ 坐标确定每个正方形的相对位置，再缩放使其能显示在窗口中，再进行整体平移，将小地图平移到左上角。由于旋转，平移，缩放等变换都为左乘，所以代码顺序应当为整体平移，再缩放，再相对平移。为了防止正方形边界连在一起，实际上在着色器里还有一个缩放变换。

```
1. void drawHawkeyeSquare(Shader shader,glm::vec3 pos,glm::vec4 color,float aspect)
2. {
3.     glm::mat4 model = glm::mat4(1.0f);
4.     model = glm::translate(model, glm::vec3(-0.95,0.95,0));
5.     model = glm::scale(model, glm::vec3(0.02,0.02*aspect,1));
6.     model = glm::translate(model, pos);
7.     shader.setMat4("model", model);
8.     shader.setVec4f("acolor", color);
9.     glDrawArrays(GL_TRIANGLES, 0, 6);
10. }
```

(3) 循环遍历绘制

首先循环遍历绘制地图方块，然后绘制目标和当前位置。在绘制时为了保证小地图不会被深度测试剔除，将深度测试方法设置为“GL_ALWAYS”。

```

1. void DrawHawkeye(float aspect) {
2.     glDepthFunc(GL_ALWAYS);
3.     if (currentScene != MAZE)
4.         return;
5.     hawkeyeShader.use();
6.     glBindVertexArray(square.VAO);
7.     glm::vec4 color;
8.     auto po = glm::ivec3(0,0,0);
9.     for (int z = 0; z < gmap.mapz; z++)
10.    {
11.        for (int x = 0; x < gmap.mapx; x++) {
12.            int y = gmap.getY(x, z);
13.            if (y == 1)
14.                color = glm::vec4(0.3, 0.8, 0.3, 0.8);
15.            if (y == 3)
16.                color = glm::vec4(0.1, 0.1, 0.1, 1);
17.            po.x = x;
18.            po.y = -z;
19.            square.drawHawkeyeSquare(hawkeyeShader,po,color ,aspect);
20.        }
21.    }
22.    color = glm::vec4(0, 0, 1, 1);
23.    po.x = pEngine.newLoc.x;
24.    po.y = -pEngine.newLoc.z;
25.    square.drawHawkeyeSquare(hawkeyeShader,po, color, aspect);
26.
27.    color = glm::vec4(1, 0, 1, 1);
28.    po.x = gmap.targetLoc.x;
29.    po.y = -gmap.targetLoc.z;
30.    square.drawHawkeyeSquare(hawkeyeShader, po, color, aspect);
31.    glDepthFunc(GL_LESS);
32. }

```

4.2.4 透明自转光照 logo

(1) 透明的实现

首先需要加载 RGBA 图片透明纹理，在加载纹理时需要将其设置为

“GL_RGBA”。然后在着色器中忽略透明像素的渲染即可。也可开启透明测试，但需要保证透明的物体最后绘制。

```
1. //shader 中代码
2. if(texColor.a < 0.1)
3.     discard;
```

(2) 自转的实现

首先应当设定一个随时间自增的旋转角度，然后对图形进行一个按 y 轴进行旋转的几何变换即可。

```
1. model = glm::rotate(model, glm::radians(float(angle)), glm::vec3(0, 1, 0));
```

(3) 光照的实现

由于在一个广阔环境中，为了增加真实感，本光照将实验中所用点光源改为平行光源。为了保证两面相同（不考虑背面不受光），可以对点乘结果取绝对值。

```
1. // diffuse
2. vec3 lightDir = normalize(vec3(0,-1,1));
3. vec3 normDir = normalize(normal);
4. float dotLN = abs(dot(lightDir, normDir)); //通过指定绝对值使得正反面相同
5. vec3 diffuse = lightDiffuse * matDiffuse * dotLN;
6.
7. // specular
8. vec3 reflectDir = normalize(reflect(-lightDir, normDir));
9. vec3 viewDir = normalize(viewPos - position);
10. float dotRV = abs(dot(reflectDir, viewDir));
11. vec3 specular = lightSpecular * matSpecular* pow(dotRV, matShininess);
```

4.2.5 天空盒及立方体贴图

为了方便地对立方体不同面进行纹理映射，OpenGL 提供了 CUBE_MAP 纹理对象。

(1) 读取 CUBE_MAP 纹理对象。

```
1. void GenerateTexCube(const char *filepath) {
2.     std::string fileName[6] = { "right", "left", "top", "bottom", "front", "back" };
3.     glBindTexture(GL_TEXTURE_CUBE_MAP, ID);
4.     for (int i = 0; i < 6; i++) {
5.         auto file = filepath + fileName[i] + ".jpg";
```

```

6.         unsigned char* data = stbi_load(file.data(), &Width, &Height, NULL
, 0);
7.         if (data)
8.             glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, Wi
dth, Height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
9.         else
10.            std::cout << "Cubemap texture failed to load at path: " << fil
e << std::endl;
11.            stbi_image_free(data);
12.    }
13.    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, Filter_Min
);
14.    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, Filter_Max
);
15.    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, Wrap_S);
16.    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, Wrap_T);
17.    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, Wrap_R);
18.    glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
19. }

```

(2) 绘制天空盒

天空盒实际就是将视点置身于一个带有纹理贴图的立方体之中。为了防止在深度测试时，该立方体遮挡其他图案，修改深度测试函数为“GL_LEQUAL”；为了让天空盒以玩家为中心的，使得不论玩家移动了多远，天空盒都不会变近，让玩家产生周围环境非常大的印象，将其观察矩阵中的位移部分去除。

```

1. void DrawSkybox() {
2.     glDepthFunc(GL_LEQUAL);
3.     skyboxShader.use();
4.     view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
5.     skyboxShader.setMat4("view", view);
6.     skyboxShader.setMat4("projection", projection);
7.     glBindVertexArray(cube.VAO);
8.     glActiveTexture(GL_TEXTURE0);
9.     skyboxTex.Bind();
10.    glDrawArrays(GL_TRIANGLES, 0, 36);
11.    glBindVertexArray(0);
12.    glDepthFunc(GL_LESS);
13. }

```

5 遇到的问题及解决

实际开发中遇到很多问题，无法一一列举，在此选取部分问题。

- 仅使用粗定位进行碰撞检测时，出现对角线穿模、“看穿”模型等 bug

解决方法：在粗定位的基础上，模仿图形算法所学 Cohen-Sutherland 的编码规则，将角色所在方格进行精定位，并根据邻域可进入性确定禁区，从而达到更好的碰撞检测效果。

- 加载四通道 RGBA 格式图片作为纹理时，会出现像素错乱。

解决方法：在调用 `glTexImage2D` 时，设置图片格式为 `GL_RGBA`。

```
1. glTexImage2D(GL_TEXTURE_2D, 0, Image_Format, Width, Height, 0, Image_Format, GL_UNSIGNED_BYTE, data);
```

- 绘制小地图时，无法保证小地图始终出现在最前方。

解决方法：绘制小地图前，将深度测试函数修改为“`GL_ALWAYS`”，即可使其在深度测试时始终绘制。

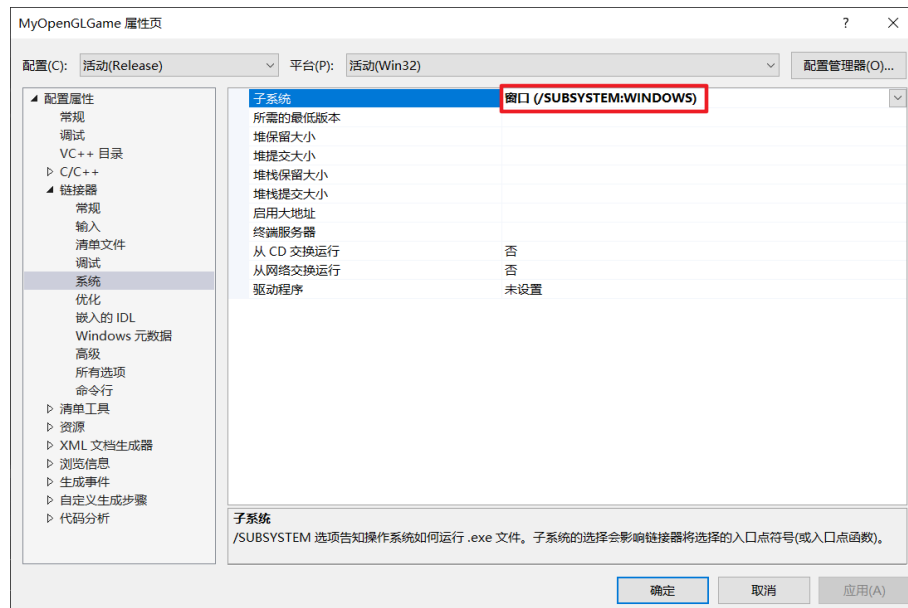
- 程序运行后窗口位置随机，有时不能居中；并且弹出控制台窗口，影响美观。

解决方法：

可以使用以下代码设置窗口居中

```
1. // 获取第一个显示器分辨率，从而设置窗口居中
2. int monitorCount;
3. GLFWmonitor** pMonitor = glfwGetMonitors(&monitorCount);
4. const GLFWvidmode* mode = glfwGetVideoMode(*pMonitor);
5. glfwSetWindowPos(window, (mode->width - SCR_WIDTH) / 2, (mode->height - SCR_HEIGHT) / 2);
```

弹出控制台窗口是因为建立的项目为 win32 控制台程序。可以将 release 版本的项目设置为窗口，这样相当于切换为 win32 程序，相应的，程序的入口也要进行修改。



```

1. // Release 版本改为 win32 程序后入口。
2. int WinMain(HINSTANCE hinstance, HINSTANCE hprevinstance, LPSTR lpcmdline,
   int ncmdshow) {
3.     main();
4. }

```

6 后续版本展望

本游戏实际上是实现了一个较为通用的框架，后续可以在这个框架中加入其它模式，或是借鉴本游戏中源码，开发其它游戏。

以下是一些未实现的想法：

加入射击模式：

首先设置枪械和手臂的贴图，使其深度测试函数设置为“GL_ALWAYS”；然后当鼠标点击时，通过摄像机方向和视点位置反推出鼠标点击处的目标，对目标进行相应处理。

地图编辑模式：

玩家在游戏中可类似于《我的世界》那样对地图方块进行放置和销毁，修改后的地图还可再进行保存到文件。

根据 DEM 栅格数据生成地图：

当前版本的地图通过读取 txt 文本文件生成，玩家可以按照格式进行地图的自定义。后续可以考虑读取 DEM 栅格数据来生成地图。但这样会造成地图过大、

高程太大影响性能。地图太大，可以在绘制时只绘制近处且在当前视角范围内的地图；高程太大可以只绘制某地块比地块邻域高的那一部分，从而减少绘制地块的数量。

迷宫模式改进：

当前的迷宫模式中，由于鹰眼地图的存在大大减小了迷宫的难度。后续可考虑增加鹰眼地图的迷雾显示，甚至不显示，只显示玩家的方向，并给玩家几个可以“贴”在地面上的 logo，玩家通过自己“贴”的 logo 来判断是否到过此处。

一三人称视角切换：

当前游戏为第一人称视角，视野受到较大的限制，后续可以模拟出角色实体，从而开启第三人称模式、上帝模式等等。