《程序设计综合实践》 工程实现报告

题目：迷宫小游戏

专业：软件工程

班级：24软件工程2班

姓名：余林隆

学号：2410321202

2025年 12月28日

目录

摘要

本次综合实验的任务是实现一个迷宫小游戏，能够完成基础的人物移动，游戏胜利，碰撞，以及进阶的特殊地块的功能，路径求解等等。目前全部完成，使用到了图，树，线性表，等等内容

一、系统总体设计

1.任务说明

本项目使用Raylib游戏库开发了一个图形化迷宫游戏，实现了迷宫可视化、路径搜索算法、角色控制、游戏流程管理及多种高级功能。以下是对所选题目的具体说明及已完成的功能和内容总结。

一、项目概述
本项目以迷宫探索为核心，通过Raylib库构建图形界面，实现了从文件加载迷宫、多种路径搜索算法、角色交互、地块效果、随机迷宫生成等完整功能。游戏支持键盘操作，具备完整的开始界面、游戏流程和胜负判定机制。

二、已完成功能及内容

1. 基础框架与可视化

   Raylib窗口与图像显示：成功创建游戏窗口，并加载图片资源（墙体、地板、草地、熔岩、起点、终点等），实现图形化迷宫的渲染。

   迷宫文件解析：支持从文本文件读取迷宫数据（包括普通地板、墙、草地、熔岩、起点和终点），并转换为可视化地图。

   自定义迷宫设计：已设计并测试了一个20×20的迷宫文件，确保算法在此迷宫上正常运行。

2. 路径搜索与显示

   深度优先遍历（DFS）与广度优先遍历（BFS）路径：实现了DFS和BFS算法，计算从起点到终点的路径，并在迷宫上以不同颜色标注显示。

   Dijkstra最短路径：实现了Dijkstra算法，计算不考虑特殊地块时的最短路径，并可视化展示。

   路径切换功能：通过键盘按键（如数字键1、2、3）可实时切换显示DFS、BFS和Dijkstra路径，方便对比。

3. 角色控制与游戏流程

   小人控制：添加了玩家角色，使用方向键（上、下、左、右）控制移动，并约束其仅能在非墙地块行走。

   完整游戏流程：

   a. 游戏开始界面：显示标题和操作说明，按空格键进入游戏。
   b. 起点与终点：小人初始位置位于起点地块，到达终点地块时弹出胜利提示。
   c. 胜负判定：到达终点即胜利；第二次踩到熔岩地板时游戏失败，并显示提示。

4. 高级地块功能

   草地减速：当小人进入草地地块时，移动速度降至正常速度的三分之一，增加游戏策略性。

   熔岩机制：第一次踩到熔岩地板无惩罚，第二次踩到则立即游戏失败，需谨慎规划路径。

5. 高级算法实现

   允许踩一次熔岩的最短路径算法：扩展Dijkstra算法，引入状态维度（是否已使用熔岩通行机会），计算可最多经过一次熔岩地块的最短路径，并在迷宫上显示该路径。

随机迷宫生成算法：实现了基于深度优先回溯的随机迷宫生成算法，可动态创建不同结构的迷宫，增加游戏可玩性。

6. 趣味性扩展功能

自主行走的敌人：在迷宫中加入了可自主移动的敌人，接触敌人会导致游戏失败，提升挑战性和趣味性。

三、总结

本项目全面完成了题目要求的各项基础与高级功能，实现了迷宫的可视化、多种路径搜索算法的集成、完整的角色控制与游戏流程，并通过随机迷宫生成、特殊地块机制、敌人AI等扩展功能增强了游戏的可玩性。所有功能均已通过测试，游戏运行稳定，交互流畅，展示了路径搜索算法在游戏中的实际应用，并提供了良好的用户体验。

2.系统总体设计思想

系统总体由三个模块构成，分别是地图模块，人物模块，算法模块。地图模块负责地图的渲染，显示出玩家游玩的地图，人物模块负责玩家和npc的移动以及相关趣味机制，算法模块负责找出在dfs和bfs等算法下对应的寻路功能和随机生成新地图的功能。其中，地图模块和人物模块相互配合，实现人物顺利移动以及碰撞相关检测和胜利相关检测，地图模块与算法模块配合，能够实现相应算法下求解的路径并且显示在地图上。地图模块需要用到线性表，图等内容，人物模块需要用到线性表，图，算法模块需要用到树，图，线性表等内容。

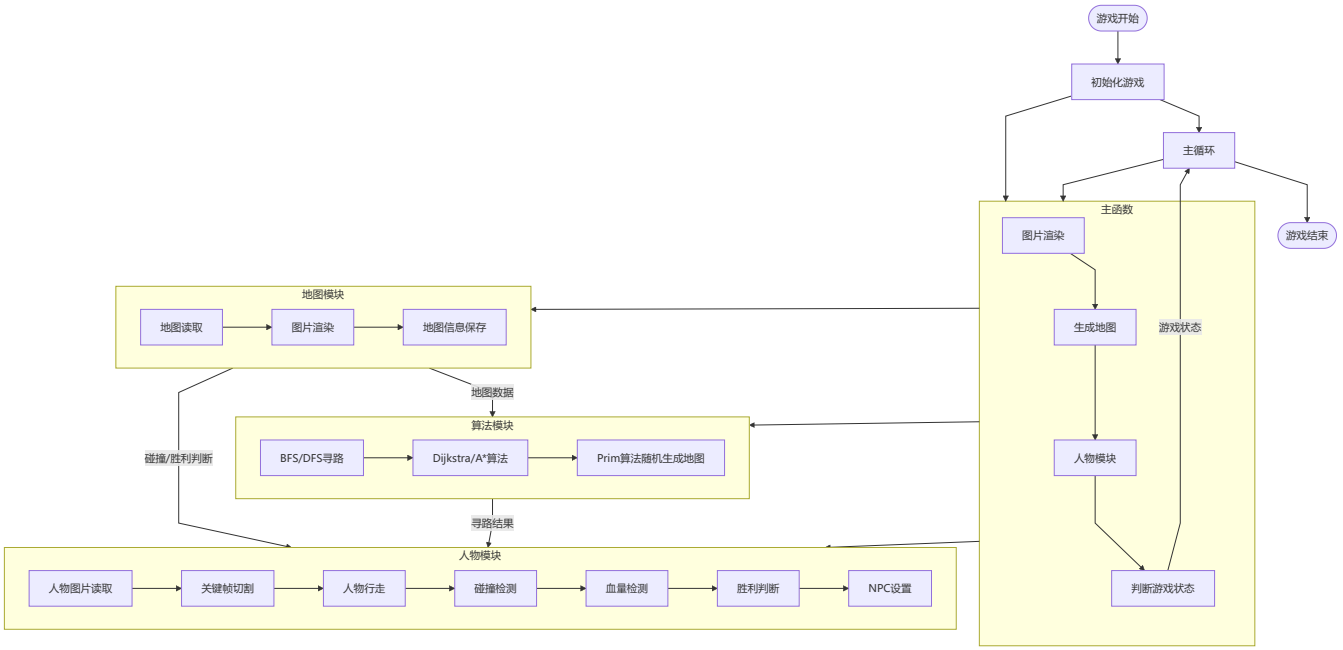
3.系统总体框架

（在本小节中，具体说明系统的总体框架实现，包括：系统的文件结构、每个源文件的功能、每个模块的文件存储位置、以及具体的系统总体框架实现。）

（使用文字描述系统的文件结构，使用markdown代码区显示相应的结构。代码区示例如下：）

```
工程源代码文件夹/
头文件代码文件夹/
    |- lujing.h # 路径显示头文件
    |- man.h# 人物模块头文件
    |- master.h # 怪物代码头文件
    |- mazegenerator.h # 随机生成地图头文件
    |- migong.h/ # 迷宫生成头文件
实现代码文件夹/
    |- lujing.cpp # 路径显示实现文件
    |- man.cpp# 人物模块实现文件
    |- master.cpp # 怪物代码实现文件
    |- mazegenerator.cpp # 随机生成地图实现
    |- migong.cpp/ # 迷宫生成实现
    |- main.cpp#迷宫游戏主函数
资源文件夹/
    |- texture
        |- character.png#主角图片
        |- end.png#终点图片
        |- floor.png#地板图片
        |- grass.png#草地图片
        |- lava.png#岩浆图片
        |- slime.png#怪物图片
        |- start.png#起点图片
        |- wall.png#墙图片
```

（使用流程图描述总体框架。mermaid流程图绘制示例如下：）



（给出总体框架代码。注意代码不要贴图，使用代码区进行展示。代码区展示示例如下：）

```
#include <raylib.h>
#include "migong.h"
#include "lujing.h"
#include "man.h"
#include "master.h"
#include <iostream>
#include <vector>
#include <string>
#include"mazegenerator.h"

int main(void) {
    // 添加迷宫生成选项
    bool useRandomMaze = false;  // 默认使用固定迷宫
    int currentAlgorithm = 0;     // 当前使用的算法
    // 设置窗口尺寸（使用你的窗口尺寸）
    int pingmuKuan = 1200;  // 你可以根据需要修改为 1000
    int pingmuGao = 1000;   // 你可以根据需要修改为 500
    bool x = true;
    InitWindow(pingmuKuan, pingmuGao, "Maze Game");
    SetTargetFPS(60);

    // 存储图片信息
    std::vector<std::pair<std::string, Texture2D>> loadedImages;

    // 检查并加载图片
    std::string imageFiles[] = { "wall.png", "grass.png", "lava.png",
                                 "floor.png", "start.png", "end.png"};

    for (const auto& filename : imageFiles) {
```

```
        if (FileExists(filename.c_str())) {
            Image img = LoadImage(filename.c_str());
            Texture2D texture = LoadTextureFromImage(img);
            loadedImages.push_back({ filename, texture });
            UnloadImage(img);
        }
    }

    // 初始化迷宫
    MiGong migong;

    if (!migong.JiaZaiDiTu("migong.txt")) {
        std::cout << "加载地图文件失败" << std::endl;
        CloseWindow();
        return -1;
    }
    if (!migong.JiaZaiTuPian()) {
        std::cout << "加载图片失败" << std::endl;
        CloseWindow();
        return -1;
    }
    migong.JisuanPianyi(pingmuKuan, pingmuGao);

    // 路径搜索
    LujingSousuo lujing(migong.GetDitu(), migong.GetQidian(), migong.GetZhongdian());
    Path dfsPath = lujing.dfs();
    Path bfsPath = lujing.bfs();
    // 新增：允许踩一次熔岩的最短路径
    int dijkstraCost = 0;
    Path dijkstraPath = lujing.dijkstraWithOneLava(dijkstraCost);

    int aStarCost = 0;
    Path aStarPath = lujing.aStarWithOneLava(aStarCost);
    bool showDFS = true;
    bool showBFS = true;
    bool showDijkstra = true;
    bool showAStar = true;
    // 创建人物与怪物
    Pos star = migong.GetQidian();
    int playerX = star.second * 48 + migong.GetPianyiX();
    int playerY = star.first * 48 + migong.GetPianyiY();
    Man player(playerX, playerY); // 人物起始位置
    MASTER master(500, 300);
    //传入地图指针
    player.setMigong(&migong);
    master.setMigong(&migong);
    bool gameStarted = false;
    // 加载开始界面的图片
    Image startScreen = LoadImage("C:/Users/余林隆/Desktop/壁纸/生成迷宫游戏开始界面.png");
    ImageResize(&startScreen, pingmuKuan, pingmuGao);  // 确保尺寸与窗口一致
    Texture2D start = LoadTextureFromImage(startScreen);
    UnloadImage(startScreen);
```

```
    // 主循环
    while (!WindowShouldClose()) {
        if (!gameStarted) {
            if (IsKeyPressed(KEY_SPACE)) {
                gameStarted = true;
                UnloadTexture(start);
            }
            BeginDrawing();
            ClearBackground(RAYWHITE);
            DrawTexture(start, 0, 0, WHITE);  // 绘制开始界面
            EndDrawing();
            continue;
        }
        // 在游戏主循环中添加按键检测
        if (IsKeyPressed(KEY_G)) {
            // 生成新的随机迷宫
            MazeGenerator generator(20, 20);
            generator.generate(currentAlgorithm, 15, 8); // 使用当前算法生成

            // 获取新迷宫
            auto newMaze = generator.getMaze();

            // 重新加载迷宫
            migong.LoadFromData(newMaze);

            // 重新计算路径
            LujingSousuo lujingNew(migong.GetDitu(), migong.GetQidian(),
migong.GetZhongdian());
            dfsPath = lujingNew.dfs();
            bfsPath = lujingNew.bfs();

            // 重新计算允许踩熔岩的路径
            dijkstraPath = lujingNew.dijkstraWithOneLava(dijkstraCost);
            aStarPath = lujingNew.aStarWithOneLava(aStarCost);

            // 重置玩家位置
            Pos newStart = migong.GetQidian();
            player.position.x = newStart.second * 48 + migong.GetPianyiX();
            player.position.y = newStart.first * 48 + migong.GetPianyiY();
            player.isFinished = false;
            player.isDead = false;
            player.blood = 3;
            player.lavaStepCount = 0;

            std::cout << "生成新迷宫完成! " << std::endl;
        }

        // 切换生成算法
        if (IsKeyPressed(KEY_F1)) {
            currentAlgorithm = 0;
            std::cout << "切换到递归回溯算法" << std::endl;
        }
        if (IsKeyPressed(KEY_F2)) {
```

```
            currentAlgorithm = 1;
            std::cout << "切换到Prim算法" << std::endl;
        }
        if (IsKeyPressed(KEY_F3)) {
            currentAlgorithm = 2;
            std::cout << "切换到Kruskal算法" << std::endl;
        }
        // 如果游戏结束（熔岩死亡），检查是否按R键重新开始
        if (player.isDead && IsKeyPressed(KEY_R)) {
            // 重置玩家状态
            player.blood = 3;
            player.isDead = false;
            player.lavaStepCount = 0;
            player.showLavaWarning = false;
            player.isFinished = false;
            player.isLavaInvincible = false;
            player.lavaInvincibleTime = 0.0f;
            // 重置玩家位置到起点
            Pos star = migong.GetQidian();
            player.position.x = star.second * 48 + migong.GetPianyiX();
            player.position.y = star.first * 48 + migong.GetPianyiY();
        }

        // 如果游戏胜利（到达终点）并且计时结束，可以退出或重置游戏
        if (player.isFinished && player.winTextTimer <= 0) {
            // 可以选择自动重新开始或等待按键
            // 例如：按空格键重新开始
            if (IsKeyPressed(KEY_SPACE)) {
                player.isFinished = false;
                player.winTextTimer = 3.0f;

                // 重置玩家位置到起点
                Pos star = migong.GetQidian();
                player.position.x = star.second * 48 + migong.GetPianyiX();
                player.position.y = star.first * 48 + migong.GetPianyiY();
            }
        }

        // 更新主角与怪物
        player.Update();
        master.Update();
        if (IsKeyPressed(KEY_ONE)) showDFS = !showDFS;
        if (IsKeyPressed(KEY_TWO)) showBFS = !showBFS;
        if (IsKeyPressed(KEY_THREE)) showDijkstra = !showDijkstra;
        if (IsKeyPressed(KEY_FOUR)) showAStar = !showAStar;
        // 检测主角与怪物的碰撞
        Rectangle playerRect = { player.position.x, player.position.y,
player.character_width, player.character_height };
        Rectangle masterRect = { master.position.x, master.position.y,
master.character_width, master.character_height };

        if (CheckCollisionRecs(playerRect, masterRect)) {
            if (!player.isInvincible) {
```

```
            player.blood -= 1;              // 扣血
            player.hurtTextTimer = 0.5f;    // 设置提示显示 0.5 秒
            player.isInvincible = true;    // 开启无敌
            player.invincibleTime = 2.0f; // 两秒无敌
        }
    }

    // 绘制迷宫和路径
    BeginDrawing();
    ClearBackground(Color{ 50, 50, 50, 255 });

    migong.HuiZhi();  // 绘制迷宫

    // 绘制路径
    float offsetX = migong.GetPianyiX();
    float offsetY = migong.GetPianyiY();

    if (showDFS) {
        for (const auto& pos : dfsPath) {
            int x = pos.second * 48 + offsetX;
            int y = pos.first * 48 + offsetY;
            DrawRectangle(x + 12, y + 12, 24, 24, ColorAlpha(BLUE, 0.6f));
        }
    }

    if (showBFS) {
        for (const auto& pos : bfsPath) {
            int x = pos.second * 48 + offsetX;
            int y = pos.first * 48 + offsetY;
            DrawRectangle(x + 12, y + 12, 24, 24, ColorAlpha(GREEN, 0.6f));
        }
    }
    if (showDijkstra && !dijkstraPath.empty()) {
        for (const auto& pos : dijkstraPath) {
            int x = pos.second * 48 + offsetX;
            int y = pos.first * 48 + offsetY;
            DrawRectangle(x + 12, y + 12, 24, 24, ColorAlpha(RED, 0.6f));
        }
    }

    if (showAStar && !aStarPath.empty()) {
        for (const auto& pos : aStarPath) {
            int x = pos.second * 48 + offsetX;
            int y = pos.first * 48 + offsetY;
            DrawRectangle(x + 12, y + 12, 24, 24, ColorAlpha(YELLOW, 0.6f));
        }
    }
    // 绘制UI

    if (IsKeyDown(KEY_SIX)) x = !x;
    if (x==true) {
        DrawRectangle(10, 95, 400, 200, ColorAlpha(BLACK, 0.7f));
        DrawText("Maze Path Finder", 20, 100, 22, YELLOW);
```

```
            DrawText(showDFS ? "1: DFS ON" : "1: DFS OFF", 20, 135, 16, BLUE);
            DrawText(showBFS ? "2: BFS ON" : "2: BFS OFF", 20, 160, 16, GREEN);
            DrawText(TextFormat("DFS: %d steps", dfsPath.size()), 20, 185, 16, BLUE);
            DrawText(TextFormat("BFS: %d steps", bfsPath.size()), 150, 185, 16, GREEN);
            DrawText(showDijkstra ? "3: Dijkstra ON" : "3: Dijkstra OFF", 20, 220, 16, RED);
            DrawText(showAStar ? "4: A* ON" : "4: A* OFF", 20, 245, 16, YELLOW);
            if (!dijkstraPath.empty()) {
                DrawText(TextFormat("Dijkstra: %d steps (cost: %d)", dijkstraPath.size(),
dijkstraCost),
                    150, 220, 16, RED);
            }
            if (!aStarPath.empty()) {
                DrawText(TextFormat("A*: %d steps (cost: %d)", aStarPath.size(), aStarCost),
                    150, 245, 16, YELLOW);
            }
        }
        // 在底部显示已加载的图片
        int bottomAreaY = pingmuGao - 140; // 图片区域起始Y坐标

        DrawRectangle(0, bottomAreaY, pingmuKuan, 140, ColorAlpha(BLACK, 0.7f));


        // 标题
        DrawText("Loaded Images:", 20, bottomAreaY + 10, 20, WHITE);

        // 显示图片缩略图
        int startX = 20;
        int imageY = bottomAreaY + 40;

        for (size_t i = 0; i < loadedImages.size(); i++) {
            int x = startX + i * 110; // 每个图片间隔110像素

            // 绘制图片（48x48）
            DrawTexture(loadedImages[i].second, x, imageY, WHITE);

            // 绘制文件名
            DrawText(loadedImages[i].first.c_str(), x, imageY + 55, 10, LIGHTGRAY);

            // 简单边框
            DrawRectangleLines(x, imageY, 48, 48, GRAY);
        }
        // 绘制控制说明
        DrawRectangle(pingmuKuan - 210, 10, 200, 70, ColorAlpha(BLACK, 0.7f));
        DrawText("Control instruction:", pingmuKuan - 200, 15, 20, YELLOW);
        DrawText("ESC - Exit the game", pingmuKuan - 200, 45, 16, WHITE);

        // 绘制主角与怪物
        player.Draw();
        master.Draw();

        // 游戏结束显示
        if (player.blood <= 0) {
            DrawText("YOU DIED", 400, 230, 40, RED);
```

```
        }

        // 绘制帧率
        DrawFPS(pingmuKuan - 90, pingmuGao - 30);

        EndDrawing();
    }

    // 清理资源
    CloseWindow();
    std::cout << "Game over" << std::endl;
    return 0;
}
```

二、模块详细设计

1.地图模块

负责读取，存贮地图信息，以及相关渲染，同时和人物模块一起实现人物碰撞，胜利等。

实现方法：迷宫模块主要负责游戏中地图数据的管理、地图资源的加载与释放，以及地图的绘制显示。该模块以 MiGong 类的形式实现，将地图的逻辑处理与显示过程集中管理，为主程序和人物模块提供统一的地图接口。

在地图数据加载方面，迷宫模块支持从文件和从算法数据两种方式加载地图。通过 JiaZaiDiTu 函数从外部地图文件中读取迷宫的行数和列数，并根据读取到的尺寸初始化二维地图数组。在逐个读取地图格子数据的过程中，程序会自动识别并记录起点和终点的位置，从而保证后续人物移动和胜利判断可以直接使用这些坐标信息。对于由算法模块生成的迷宫数据，模块通过 LoadFromData 函数直接接收二维数组形式的地图，并扫描整个地图查找起点和终点。如果未检测到起点或终点，则会设置默认位置，保证地图的完整性和可用性。

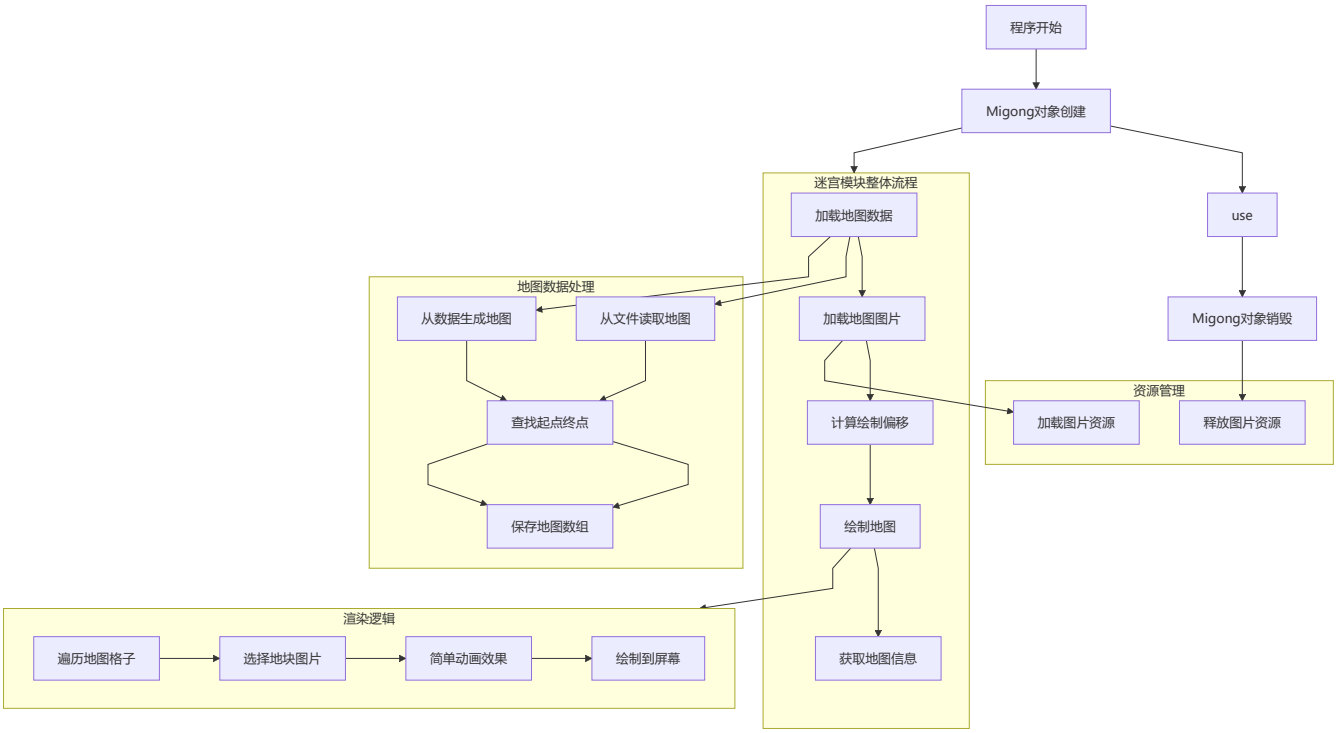
在地图资源管理方面，迷宫模块负责加载和释放地图所需的图片资源。JiaZaiTuPian 函数分别加载墙体、普通地面、草地、熔岩、起点和终点等贴图资源。加载过程中会先判断文件是否存在，再将 Image 转换为 Texture2D，并设置纹理过滤方式以保证显示效果。在迷宫对象销毁时，通过析构函数统一调用 ShiFangTuPian 释放所有纹理资源，避免内存泄漏。

在地图绘制方面，迷宫模块通过 HuiZhi 函数将地图显示到屏幕上。绘制时使用双重循环遍历整个地图二维数组，根据当前格子的位置计算其在屏幕中的实际像素坐标，并判断该格子是否位于屏幕可视范围内，从而减少不必要的绘制操作。程序会根据格子类型选择对应的贴图进行绘制，并对草地、熔岩、起点和终点等特殊地块加入简单的正弦函数动画效果，使画面更加生动。绘制过程中使用 DrawTexturePro 函数实现对纹理大小和位置的精确控制。

为了保证地图在不同分辨率下显示合理，迷宫模块提供了 JisuanPianyi 函数用于计算地图的绘制偏移量。该函数根据屏幕尺寸和地图实际尺寸计算居中显示所需的偏移值，并对偏移量进行限制，防止出现负坐标导致绘制异常。

此外，迷宫模块还提供了地图信息获取功能。通过 HuoQuDiTuXinxi 函数可以获得当前地图的行列数以及起点和终点的位置，该功能主要用于调试输出或界面信息显示，不参与游戏核心逻辑处理。

流程图：



模块实现代码：

```
#include "migong.h"
#include <cmath>
#include <sstream>


// 修改构造函数，显式初始化所有 Texture2D 成员变量
MiGong::MiGong() :
    hangshu(0),
    lieshu(0),
    qidianWeizhi({ 0, 0 }),
    zhongdianWeizhi({ 0, 0 }),
    huizhiPianyi({ 0, 0 }),
    qiangTupian({}),
    caodiTupian({}),
    rongyanTupian({}),
    putongTupian({}),
    qidianTupian({}),
    zhongdianTupian({})
{
    // 构造函数体为空
}


MiGong::~MiGong() {
    ShiFangTuPian();
}

bool MiGong::JiaZaiDiTu(const std::string& wenjianming) {
    std::ifstream wenjian(wenjianming);
    if (!wenjian.is_open()) {
```

```
        return false;
    }

    // 读取地图尺寸
    wenjian >> hangshu >> lieshu;

    if (hangshu <= 0 || lieshu <= 0) {
        return false;
    }

    // 初始化地图数组
    ditu.resize(hangshu, std::vector<int>(lieshu));

    // 读取地图数据
    for (int i = 0; i < hangshu; ++i) {
        for (int j = 0; j < lieshu; ++j) {
            wenjian >> ditu[i][j];

            // 记录起点和终点位置
            if (ditu[i][j] == QIDIAN) {
                qidianWeizhi.x = j;
                qidianWeizhi.y = i;
            }
            else if (ditu[i][j] == ZHONGDIAN) {
                zhongdianWeizhi.x = j;
                zhongdianWeizhi.y = i;
            }
        }
    }

    wenjian.close();
    return true;
}

bool MiGong::JiaZaiTuPian() {
    // 1. 加载墙图片
    if (FileExists("wall.png")) {
        Image qiangTuxiang = LoadImage("wall.png");
        qiangTupian = LoadTextureFromImage(qiangTuxiang);
        SetTextureFilter(qiangTupian, TEXTURE_FILTER_POINT);
        UnloadImage(qiangTuxiang);
        }

        // 2. 加载草地图片
        if (FileExists("grass.png")) {
            Image caodiTuxiang = LoadImage("grass.png");
            caodiTupian = LoadTextureFromImage(caodiTuxiang);
            SetTextureFilter(caodiTupian, TEXTURE_FILTER_POINT);
            UnloadImage(caodiTuxiang);
        }

        // 3. 加载熔岩图片
        if (FileExists("lava.png")) {
```

```
            Image rongyanTuxiang = LoadImage("lava.png");
            rongyanTupian = LoadTextureFromImage(rongyanTuxiang);
            SetTextureFilter(rongyanTupian, TEXTURE_FILTER_POINT);
            UnloadImage(rongyanTuxiang);
        }

        // 4. 加载普通地面图片
        if (FileExists("floor.png")) {
            Image putongTuxiang = LoadImage("floor.png");
            putongTupian = LoadTextureFromImage(putongTuxiang);
            SetTextureFilter(putongTupian, TEXTURE_FILTER_POINT);
            UnloadImage(putongTuxiang);
        }

        // 5. 加载起点图片
        if (FileExists("start.png")) {
            Image qidianTuxiang = LoadImage("start.png");
            qidianTupian = LoadTextureFromImage(qidianTuxiang);
            SetTextureFilter(qidianTupian, TEXTURE_FILTER_POINT);
            UnloadImage(qidianTuxiang);
        }

        // 6. 加载终点图片
        if (FileExists("end.png")) {
            Image zhongdianTuxiang = LoadImage("end.png");
            zhongdianTupian = LoadTextureFromImage(zhongdianTuxiang);
            SetTextureFilter(zhongdianTupian, TEXTURE_FILTER_POINT);
            UnloadImage(zhongdianTuxiang);
        }

        return true;
    };

void MiGong::ShiFangTuPian() {
    UnloadTexture(qiangTupian);
    UnloadTexture(caodiTupian);
    UnloadTexture(rongyanTupian);
    UnloadTexture(putongTupian);
    UnloadTexture(qidianTupian);
    UnloadTexture(zhongdianTupian);
}

void MiGong::HuiZhi() {


    // 绘制所有地块
    for (int i = 0; i < hangshu; ++i) {
        for (int j = 0; j < lieshu; ++j) {
            int x = huizhiPianyi.x + j * 48;
            int y = huizhiPianyi.y + i * 48;

            // 检查地块是否在屏幕外（解决下半部分显示不全问题）
            if (y + 48 < 0 || y > GetScreenHeight() ||
```

```
            x + 48 < 0 || x > GetScreenWidth()) {
            continue; // 跳过完全在屏幕外的地块
        }

        // 选择图片
        Texture2D* dangqianTupian = &putongTupian;
        Color yanse = WHITE;

        // 根据地块类型选择图片
        switch (ditu[i][j]) {
        case QIANG:
            dangqianTupian = &qiangTupian;
            break;

        case CAODI:
            dangqianTupian = &caodiTupian;
            // 草地轻微动画效果
            yanse = ColorAlpha(WHITE, 0.9f + 0.1f * sinf(GetTime() * 2));
            break;

        case RONGYAN:
            dangqianTupian = &rongyanTupian;
            // 熔岩动画效果
            yanse = ColorAlpha(WHITE, 0.8f + 0.2f * sinf(GetTime() * 3));
            break;

        case QIDIAN:
            dangqianTupian = &qidianTupian;
            // 起点动画效果
            yanse = ColorAlpha(WHITE, 0.7f + 0.3f * sinf(GetTime() * 4));
            break;

        case ZHONGDIAN:
            dangqianTupian = &zhongdianTupian;
            // 终点动画效果
            yanse = ColorAlpha(WHITE, 0.6f + 0.4f * sinf(GetTime() * 2));
            break;

        default: // PUTONG
            dangqianTupian = &putongTupian;
            break;
        }

        Rectangle destRect = {
            static_cast<float>(x),
            static_cast<float>(y),
            48.0f,
            48.0f
        };

        // 使用 DrawTexturePro 进行精确控制绘制
        DrawTexturePro(
            *dangqianTupian,
```

```
                Rectangle{ 0, 0, static_cast<float>(dangqianTupian->width),
static_cast<float>(dangqianTupian->height) },
                destRect,
                Vector2{ 0, 0 },
                0.0f,
                yanse
            );

        }
    }
}

void MiGong::JisuanPianyi(int pingmuKuan, int pingmuGao) {
    // 计算地图总尺寸
    int dituKuan = lieshu * 48;
    int dituGao = hangshu * 48;

    // 计算居中偏移量
    huizhiPianyi.x = (pingmuKuan - dituKuan) / 2.0f;
    huizhiPianyi.y = (pingmuGao - dituGao) / 2.0f;

    // 确保偏移量不小于0
    if (huizhiPianyi.x < 0) huizhiPianyi.x = 0;
    if (huizhiPianyi.y < 0) huizhiPianyi.y = 0;
}

std::string MiGong::HuoQuDiTuXinxi() const {
    std::stringstream ss;
    ss << "Map: " << hangshu << "x" << lieshu ;
    ss << " | Start: (" << (int)qidianWeizhi.y << "," << (int)qidianWeizhi.x << ")";
    ss << " | End: (" << (int)zhongdianWeizhi.y << "," << (int)zhongdianWeizhi.x << ")";
    return ss.str();
}
bool MiGong::LoadFromData(const std::vector<std::vector<int>>& mazeData) {
    if (mazeData.empty() || mazeData[0].empty()) {
        return false;
    }

    ditu = mazeData;
    hangshu = ditu.size();        // 替换 rows
    lieshu = ditu[0].size();      // 替换 cols

    // 查找起点和终点
    bool foundStart = false, foundEnd = false;
    for (int i = 0; i < hangshu; i++) {        // 替换 rows  hangshu
        for (int j = 0; j < lieshu; j++) {     // 替换 cols  lieshu
            if (ditu[i][j] == -1) {
                qidianWeizhi = { (float)j, (float)i };  // 替换 qidian  qidianWeizhi
                foundStart = true;
            }
            else if (ditu[i][j] == -2) {
                zhongdianWeizhi = { (float)j, (float)i }; // 替换 zhongdian  zhongdianWeizhi
                foundEnd = true;
```

```
            }
        }
    }

    // 如果没有找到起点或终点，设置默认值
    if (!foundStart) {
        qidianWeizhi = { 1, 1 };            // 替换 qidian qidianWeizhi
        if (1 < hangshu && 1 < lieshu) {    // 确保坐标有效
            ditu[1][1] = -1;
        }
    }
    if (!foundEnd) {
        zhongdianWeizhi = { (float)(lieshu - 2), (float)(hangshu - 2) }; // 替换 zhongdian
zhongdianWeizhi
        if ((hangshu - 2) >= 0 && (hangshu - 2) < hangshu &&
            (lieshu - 2) >= 0 && (lieshu - 2) < lieshu) {  // 确保坐标有效
            ditu[hangshu - 2][lieshu - 2] = -2;
        }
    }

    return true;
}
```

2.人物模块

负责人物的绘制，人物的移动，人物的状态，人物碰撞，等内容。

实现方法：人物模块主要负责主角的显示、移动控制以及人物状态的更新。首先在构造函数中接收人物的初始坐标和移动速度，并加载人物图片资源。通过将整张图片按行和列切分为多个关键帧，并使用 frameRect 记录当前需要显示的关键帧区域。不同方向的行走动画通过 numberx 和 numbery 进行编号，在人物移动时根据方向切换对应的关键帧，从而实现人物上下左右行走的效果。

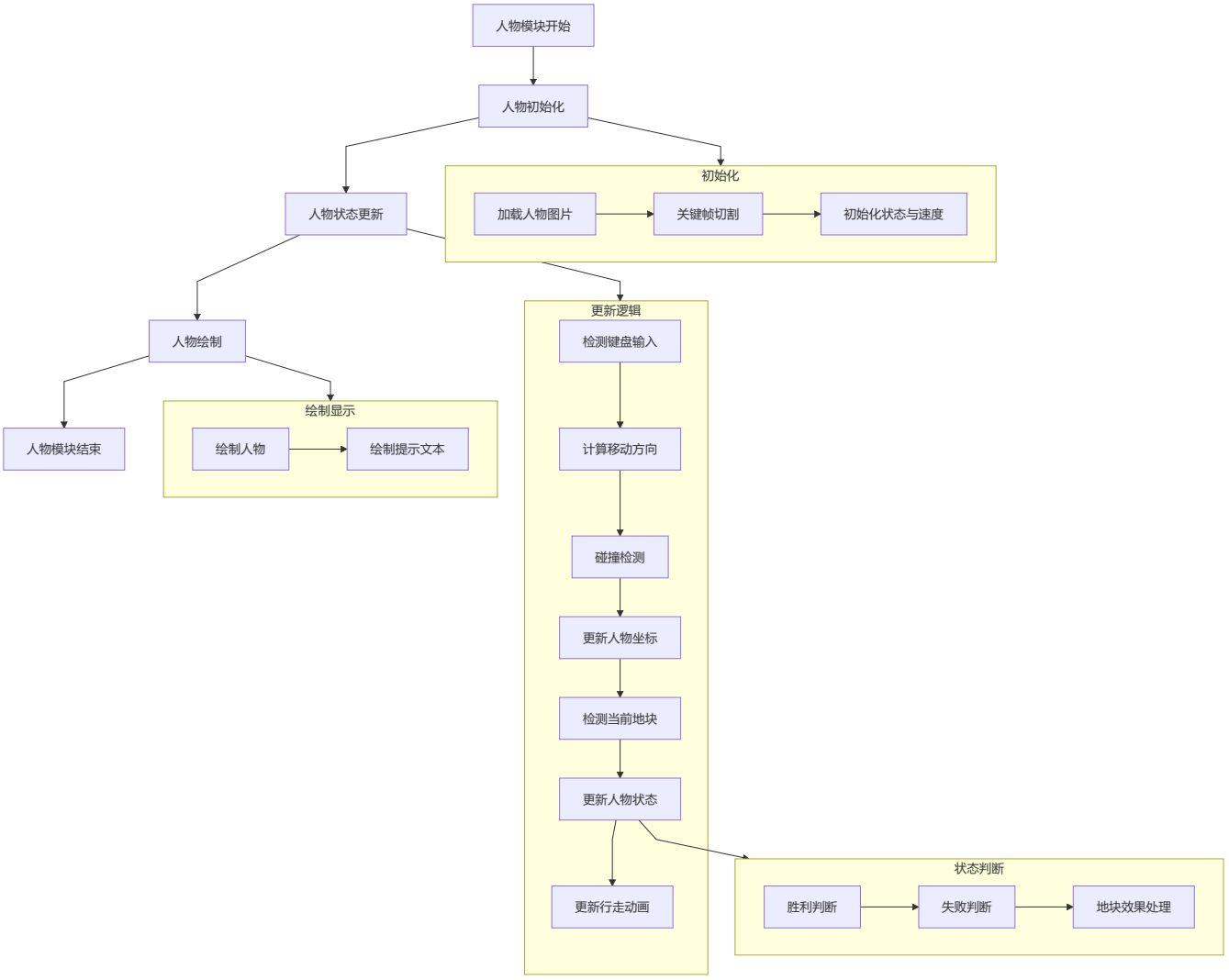
在人物移动实现方面，首先检测键盘输入，根据按下的方向键计算人物在该方向上的计划移动距离。人物的实际移动过程分为 X 轴和 Y 轴两个方向分别处理，先在某一个方向上进行碰撞检测，确认不会撞到墙体后再更新人物坐标。碰撞检测通过将人物的碰撞箱转换为地图格子坐标，并判断当前覆盖的格子是否为墙体来实现，从而保证人物无法穿墙移动。在没有绑定迷宫指针的情况下，人物可以直接按照输入方向移动，方便单独测试人物模块。

在人物与地图交互方面，人物会实时获取当前所处地块的类型。程序通过计算人物中心点所在的地图格子，判断该位置对应的地块编号，并根据不同地块类型触发相应效果。当人物进入终点地块时，将胜利状态设置为真，并显示对应的胜利提示文本；当人物行走在草地上时，会降低人物的移动速度；当人物踩到熔岩地块时，第一次会给出警告并进入短暂的无敌状态，如果再次踩到熔岩，则判定人物失败并结束游戏。

在人物状态管理方面，人物状态主要包括行走状态、胜利状态和失败状态。行走状态用于控制人物动画的播放，当人物处于移动状态时按照固定时间间隔切换关键帧，未移动时则显示站立帧。胜利状态用于控制胜利提示文字的显示时间，在倒计时结束后结束游戏。失败状态主要用于熔岩相关的死亡判断，当人物被熔岩击败后，停止人物移动，并在屏幕中央显示游戏失败提示和重新开始的说明。

在人物绘制方面，人物模块通过 Draw 函数完成角色和相关提示信息的显示。人物本身使用 DrawTextureRec 函数绘制当前关键帧，同时根据人物当前状态显示受伤提示、熔岩警告文字以及胜利或失败的提示界面，使游戏反馈更加直观。

流程图：



实现代码：

```
#include "man.h"

Man::Man(float x, float y, float s) {
    position = { x, y };
    speed = s;
    basespeed = s;
    // 加载图片（翻转）
    Image img = LoadImage("character.png");
    ImageResize(&img, 140,180 );
    texture = LoadTextureFromImage(img);
    UnloadImage(img);
    if (texture.id == 0) {
        TraceLog(LOG_ERROR, "Texture load failed!");
    }
    // 初始化行走帧尺寸
```

```
    character_width = texture.width / 3;
    character_height = texture.height / 4;
    //初始化熔岩相关
    isLavaInvincible = false;
    lavaInvincibleTime = 0.0f;
    lavaInvincibleDuration = 1.0f;
    // 默认人物状态（站立，向下）
    numberx = 1;
    numbery = 0;

    // 初始化帧矩形，否则第一帧会不显示
    frameRect = {
        numberx * character_width,
        numbery * character_height,
        character_width,
        character_height
    };
}


void Man::Update() {
    // 如果已经到达终点或死亡，不处理移动
    if (isFinished || isDead) {
        // 只是更新计时器
        if (isFinished && winTextTimer > 0) {
            winTextTimer -= GetFrameTime();
        }
        if (showLavaWarning && lavaWarningTimer > 0) {
            lavaWarningTimer -= GetFrameTime();
            if (lavaWarningTimer <= 0) {
                showLavaWarning = false;
            }
        }
        return;
    }
    float dt = GetFrameTime();
    bool moving = false;

    // 记录计划移动的距离
    float plannedMoveX = 0, plannedMoveY = 0;

    // 1. 先计算玩家想要移动的方向和距离
    if (IsKeyDown(KEY_UP)) {
        plannedMoveY = -speed * dt;
        numbery = 3;
        moving = true;
    }
    if (IsKeyDown(KEY_DOWN)) {
        plannedMoveY = speed * dt;
        numbery = 0;
        moving = true;
    }
    if (IsKeyDown(KEY_LEFT)) {
```

```
        plannedMoveX = -speed * dt;
        numbery = 1;
        moving = true;
    }
    if (IsKeyDown(KEY_RIGHT)) {
        plannedMoveX = speed * dt;
        numbery = 2;
        moving = true;
    }

    // 2. 先尝试X轴移动（如果计划有X轴移动）
    if (plannedMoveX != 0 && migongptr != nullptr) {
        float testX = position.x + plannedMoveX; // 测试移动后的X位置

        // 计算测试位置的碰撞箱
        float left = testX + collisionOffset;
        float right = testX + character_width - collisionOffset;
        float top = position.y + collisionOffset;
        float bottom = position.y + character_height - collisionOffset;

        const auto& map = migongptr->GetDitu();
        float offsetX = migongptr->GetPianyiX();
        float offsetY = migongptr->GetPianyiY();

        // 转换为格子坐标
        int gridLeft = static_cast<int>((left - offsetX) / 48);
        int gridRight = static_cast<int>((right - offsetX) / 48);
        int gridTop = static_cast<int>((top - offsetY) / 48);
        int gridBottom = static_cast<int>((bottom - offsetY) / 48);

        // 确保坐标在有效范围内
        if (gridTop >= 0 && gridBottom < map.size() &&
            gridLeft >= 0 && gridRight < map[0].size()) {

            bool canMove = true;

            // 检查碰撞箱覆盖的所有格子
            for (int y = gridTop; y <= gridBottom; ++y) {
                for (int x = gridLeft; x <= gridRight; ++x) {
                    if (map[y][x] == 1) { // 碰到墙
                        canMove = false;
                        break;
                    }
                }
                if (!canMove) break;
            }

            // 如果可以移动，则更新X位置
            if (canMove) {
                position.x = testX;
            }
            // 如果不能移动，就保持原位置（不做任何事）
        }
```

```
    }

    // 3. 再尝试Y轴移动（如果计划有Y轴移动）
    if (plannedMoveY != 0 && migongptr != nullptr) {
        float testY = position.y + plannedMoveY; // 测试移动后的Y位置

        // 计算测试位置的碰撞箱（使用可能已更新的X位置）
        float left = position.x + collisionOffset;
        float right = position.x + character_width - collisionOffset;
        float top = testY + collisionOffset;
        float bottom = testY + character_height - collisionOffset;

        const auto& map = migongptr->GetDitu();
        float offsetX = migongptr->GetPianyiX();
        float offsetY = migongptr->GetPianyiY();

        // 转换为格子坐标
        int gridLeft = static_cast<int>((left - offsetX) / 48);
        int gridRight = static_cast<int>((right - offsetX) / 48);
        int gridTop = static_cast<int>((top - offsetY) / 48);
        int gridBottom = static_cast<int>((bottom - offsetY) / 48);

        // 确保坐标在有效范围内
        if (gridTop >= 0 && gridBottom < map.size() &&
            gridLeft >= 0 && gridRight < map[0].size()) {

            bool canMove = true;

            // 检查碰撞箱覆盖的所有格子
            for (int y = gridTop; y <= gridBottom; ++y) {
                for (int x = gridLeft; x <= gridRight; ++x) {
                    if (map[y][x] == 1) { // 碰到墙
                        canMove = false;
                        break;
                    }
                }
                if (!canMove) break;
            }

            // 如果可以移动，则更新Y位置
            if (canMove) {
                position.y = testY;
            }
            // 如果不能移动，就保持原位置
        }
    }

    // 4. 如果迷宫指针为空，则直接移动（用于测试）
    if (migongptr == nullptr) {
        position.x += plannedMoveX;
        position.y += plannedMoveY;
    }
    // 4. 【新增】检测当前位置的地块类型并处理效果
```

```
    int tileType = GetCurrentTileType();

    // 重置速度为基本速度
    speed = basespeed;

    // 根据地块类型处理
    switch (tileType) {
    case -2: // 终点
        isFinished = true;
        winTextTimer = 3.0f; // 显示3秒胜利信息
        break;

    case 2: // 草地，减速三分之一
        speed = basespeed * 0.33f; // 减速到三分之一
        break;

    case 3: // 熔岩
        // 更新熔岩无敌计时器
        if (isLavaInvincible) {
            lavaInvincibleTime -= dt;
            if (lavaInvincibleTime <= 0) {
                isLavaInvincible = false;
            }
        }

        // 如果不在无敌状态，处理熔岩效果
        if (!isLavaInvincible) {
            lavaStepCount++;

            if (lavaStepCount == 1) {
                // 第一次踩到熔岩，显示警告并进入无敌状态
                showLavaWarning = true;
                lavaWarningTimer = 2.0f;
                isLavaInvincible = true;
                lavaInvincibleTime = lavaInvincibleDuration;
            }
            else if (lavaStepCount >= 2) {
                // 第二次踩到熔岩，游戏失败
                isDead = true;
            }
        }
        break;
    default:
        // 其他地块不做特殊处理
        break;
    }


    // 5. 窗口边界限制
    float w = (float)GetScreenWidth();
    float h = (float)GetScreenHeight();
    if (position.x < 0) position.x = 0;
    if (position.y < 0) position.y = 0;
```

```
    if (position.x > w - character_width) position.x = w - character_width;
    if (position.y > h - character_height) position.y = h - character_height;

    // 6. 动画更新
    if (moving) {
        timer += dt;
        if (timer >= frametime) {
            timer = 0;
            numberx = (numberx + 1) % 3;
        }
    }
    else {
        numberx = 1;
    }

    // 7. 受伤逻辑
    if (isInvincible) {
        invincibleTime -= dt;
        if (invincibleTime <= 0) {
            isInvincible = false;
        }
    }

    if (hurtTextTimer > 0) {
        hurtTextTimer -= dt;
    }// 8. 熔岩警告计时器递减
    if (showLavaWarning && lavaWarningTimer > 0) {
        lavaWarningTimer -= dt;
        if (lavaWarningTimer <= 0) {
            showLavaWarning = false;
        }
    }

    // 8. 更新帧矩形
    frameRect = {
        numberx * character_width,
        numbery * character_height,
        character_width,
        character_height
    };
}
int Man::GetCurrentTileType() {
    if (migongptr == nullptr) return 0; // 默认普通地面

    const auto& map = migongptr->GetDitu();
    float offsetX = migongptr->GetPianyiX();
    float offsetY = migongptr->GetPianyiY();

    // 计算人物中心点
    float centerX = position.x + character_width / 2.0f;
    float centerY = position.y + character_height / 2.0f;

    // 将屏幕坐标转换为地图格子坐标
```

```
    int gridX = static_cast<int>((centerX - offsetX) / 48);
    int gridY = static_cast<int>((centerY - offsetY) / 48);

    // 检查格子坐标是否有效
    if (gridY >= 0 && gridY < map.size() && gridX >= 0 && gridX < map[0].size()) {
        return map[gridY][gridX];
    }

    return 0; // 默认普通地面
}
void Man::Draw() {
    DrawTextureRec(texture, frameRect, position, WHITE);
    // 受伤提示文字
    if (hurtTextTimer > 0) {
        DrawText("-1 HP", position.x, position.y - 20, 20, RED);
    } // 熔岩警告
    if (showLavaWarning && lavaWarningTimer > 0) {
        const char* warningText = "WARNING: You stepped on lava! Step again to die!";
        int textWidth = MeasureText(warningText, 30);
        int screenWidth = GetScreenWidth();
        DrawText(warningText, (screenWidth - textWidth) / 2, 50, 30, ORANGE);
    }// 胜利显示
    if (isFinished && winTextTimer > 0) {
        const char* winText = "VICTORY! You reached the destination!";
        int textWidth = MeasureText(winText, 50);
        int screenWidth = GetScreenWidth();
        int screenHeight = GetScreenHeight();

        // 半透明背景
        DrawRectangle(0, screenHeight / 2 - 60, screenWidth, 120, Fade(BLACK, 0.7f));

        // 胜利文字
        DrawText(winText, (screenWidth - textWidth) / 2, screenHeight / 2 - 25, 50, GREEN);

        // 倒计时提示
        std::string countdownText = "Game ends in " +
std::to_string((int)ceil(winTextTimer)) + " seconds";
        int countdownWidth = MeasureText(countdownText.c_str(), 20);
        DrawText(countdownText.c_str(), (screenWidth - countdownWidth) / 2, screenHeight / 2
+ 35, 20, YELLOW);
    }

    // 游戏失败显示（熔岩）
    if (isDead) {
        const char* loseText = "GAME OVER! You were burned by lava!";
        int textWidth = MeasureText(loseText, 50);
        int screenWidth = GetScreenWidth();
        int screenHeight = GetScreenHeight();

        // 半透明背景
        DrawRectangle(0, screenHeight / 2 - 60, screenWidth, 120, Fade(BLACK, 0.7f));

        // 失败文字
```

```
        DrawText(loseText, (screenWidth - textWidth) / 2, screenHeight / 2 - 25, 50, RED);

        // 提示重新开始
        const char* restartText = "Press R to restart the game";
        int restartWidth = MeasureText(restartText, 30);
        DrawText(restartText, (screenWidth - restartWidth) / 2, screenHeight / 2 + 35, 30,
YELLOW);
    }
}

void Man::setMigong(MiGong* m) {
    migongptr = m;
}
Man::~Man() {
    UnloadTexture(texture);
}
```

3.算法部分

负责整个bfs和dfs寻路，包括路径显示，随机生成地图，寻找成本最低的路径等内容

实现方法：首先是bfs和dfs。

先接收地图二维数组的信息以及起点，终点的位置，初始化相应信息，然后使用一个函数isvalid当前坐标是否合法，防止出现越界行为。再使用getneighbors来获取当前位置相邻的节点，为后续两个提供方法。

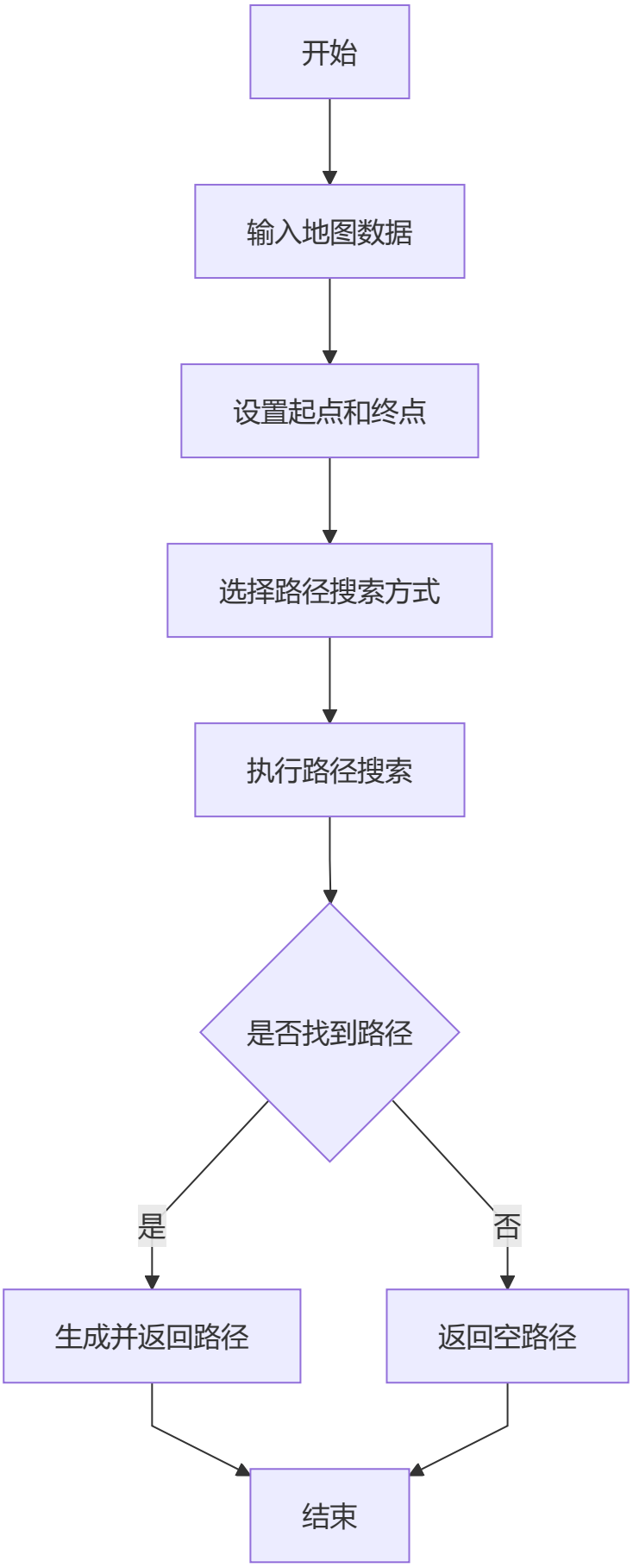
在dfs上，首先使用一个栈来模拟递归调用，并且记录每一个节点从起点到当前位置的路径，使用数组来标记所有已经访问的格子，防止重复搜索。搜索到终点时再返回。

在bfs上，使用队列来逐渐处理节点，用一个数组来记录下每个探索过的节点，bfs能够找到的路径就回事最短的。

在考虑不同地块移动代价的情况下，路径搜索模块引入了“状态”这一概念。通过 函数为不同类型地块设置不同的移动代价：普通地面、起点和终点代价较低，草地代价较高，而熔岩地块则限制只能低代价通过一次。`getNeighborsWithState` 在获取相邻节点时，将是否已经经过熔岩作为状态的一部分，从而保证搜索过程中能够正确处理熔岩的使用规则。

在 Dijkstra 算法中，通过 `dijkstraWithOneLava` 函数实现带状态的最短路径搜索。算法使用三维数组记录在不同位置和熔岩使用状态下的最小代价，并借助优先队列每次选择当前代价最小的节点进行扩展。当到达终点时，根据记录的前驱节点反向重建路径并返回结果；若搜索失败，则返回空路径。

在 A* 算法中，通过 `aStarWithOneLava` 函数在 Dijkstra 的基础上加入启发式策略。启发函数采用当前位置到终点的曼哈顿距离，用于引导搜索方向，从而减少不必要的节点扩展，提高搜索效率。路径找到后，同样根据前驱信息重建并输出完整路径。


开始
输入地图数据
设置起点和终点
选择路径搜索方式
执行路径搜索
是否找到路径
是
否
生成并返回路径
返回空路径
结束

实现代码：

```cpp
#include "lujing.h"
#include <vector>
#include <utility>
#include <queue>
#include <stack>
#include <limits>
#include <cmath>
#include <unordered_set>
#include <tuple>
LujingSousuo::LujingSousuo(const std::vector<std::vector<int>>& m, Pos s, Pos e)//构造函数，对节点等初始化
    : maze(m), rows(m.size()), cols(m[0].size()), start(s), end(e) {
}

bool LujingSousuo::isValid(int x, int y) {//判断是否在地图内且该地方是否为墙
    return x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] != 1;
}

std::vector<Pos> LujingSousuo::getNeighbors(int x, int y) {
    std::vector<Pos> neighbors;
    int dirs[4][2] = { {-1,0}, {1,0}, {0,-1}, {0,1} };//定义邻居的位置

    for (auto& d : dirs) {
        int nx = x + d[0];
        int ny = y + d[1];
        if (isValid(nx, ny)) {
            neighbors.push_back({ nx, ny });
        }
    }
    return neighbors;
}

// 深度优先搜索（DFS）函数，返回一条从起点到终点的路径（如果存在）
Path LujingSousuo::dfs() {
    // 使用栈来模拟递归调用，栈中存储（当前位置，从起点到当前位置的路径）对
    std::stack<std::pair<Pos, Path>> stk;

    // 创建访问标记数组，用于记录每个位置是否已被访问过
    std::vector<std::vector<bool>> visited(rows, std::vector<bool>(cols, false));

    // 初始化：将起点及其路径（只包含起点）压入栈中，并标记起点为已访问
    stk.push({ start, {start} });
    visited[start.first][start.second] = true;

    // 主循环：当栈不为空时继续搜索
    while (!stk.empty()) {
        // 获取栈顶元素（最近添加的位置）
        auto [pos, path] = stk.top();
        stk.pop();  // 弹出已处理的节点

        // 提取当前位置坐标
        int x = pos.first, y = pos.second;
```

```
        // 终止条件：如果当前位置就是终点，返回当前路径
        if (pos == end) {
            return path;  // 找到目标路径，立即返回
        }

        // 遍历当前节点的所有合法邻居节点
        for (auto& nb : getNeighbors(x, y)) {
            int nx = nb.first, ny = nb.second;

            // 如果邻居节点未被访问过
            if (!visited[nx][ny]) {
                visited[nx][ny] = true;  // 标记为已访问，避免重复探索

                // 创建新路径：将当前路径复制一份，然后添加新节点
                Path newPath = path;
                newPath.push_back(nb);

                // 将新节点及其路径压入栈中，以便后续继续探索
                stk.push({ nb, newPath });
            }
        }
    }

    // 如果栈已空但未找到终点，返回空路径表示搜索失败
    return {};
}

Path LujingSousuo::bfs() {
    std::queue<std::pair<Pos, Path>> q;
    std::vector<std::vector<bool>> visited(rows, std::vector<bool>(cols, false));

    q.push({ start, {start} });
    visited[start.first][start.second] = true;

    while (!q.empty()) {
        auto [pos, path] = q.front();
        q.pop();

        int x = pos.first, y = pos.second;

        if (pos == end) {
            return path;
        }

        for (auto& nb : getNeighbors(x, y)) {
            int nx = nb.first, ny = nb.second;
            if (!visited[nx][ny]) {
                visited[nx][ny] = true;
                Path newPath = path;
                newPath.push_back(nb);
                q.push({ nb, newPath });
            }
        }
```

```
    }
    return {};
}
int LujingSousuo::getMoveCost(int tileType, int usedLava) {
    switch (tileType) {
    case 0:  // 普通地块
    case -1: // 起点
    case -2: // 终点
        return 1;
    case 2:  // 草地
        return 3;
    case 3:  // 熔岩
        if (usedLava == 0) {
            return 0;  // 第一次踩熔岩免费
        }
        else {
            return 1000;  // 第二次踩熔岩高成本
        }
    default:
        return 1;
    }
}
std::vector<NodeState> LujingSousuo::getNeighborsWithState(const NodeState& node) {
    std::vector<NodeState> neighbors;
    int dirs[4][2] = { {-1,0}, {1,0}, {0,-1}, {0,1} };

    for (auto& d : dirs) {
        int nx = node.x + d[0];
        int ny = node.y + d[1];

        if (nx >= 0 && nx < rows && ny >= 0 && ny < cols && maze[nx][ny] != 1) {
            int tileType = maze[nx][ny];

            // 如果是熔岩且已经使用过熔岩机会，跳过
            if (tileType == 3 && node.usedLava == 1) {
                continue;
            }

            int newUsedLava = node.usedLava;
            if (tileType == 3 && node.usedLava == 0) {
                newUsedLava = 1;  // 标记已使用熔岩机会
            }

            int moveCost = getMoveCost(tileType, node.usedLava);
            neighbors.push_back(NodeState(nx, ny, newUsedLava, node.cost + moveCost));
        }
    }

    return neighbors;
}Path LujingSousuo::dijkstraWithOneLava(int& totalCost) {
    // 三维距离数组：dist[x][y][usedLava]
    std::vector<std::vector<std::vector<int>>> dist(
```

```
        rows, std::vector<std::vector<int>>(cols, std::vector<int>(2,
std::numeric_limits<int>::max())));

    // 三维前驱数组
    std::vector<std::vector<std::vector<NodeState>>> prev(
        rows, std::vector<std::vector<NodeState>>(cols, std::vector<NodeState>(2,
NodeState(-1, -1, -1))));

    // 优先队列（最小堆）
    std::priority_queue<NodeState, std::vector<NodeState>, std::greater<NodeState>> pq;

    // 初始化起点
    dist[start.first][start.second][0] = 0;
    pq.push(NodeState(start.first, start.second, 0, 0));

    while (!pq.empty()) {
        NodeState current = pq.top();
        pq.pop();

        int x = current.x;
        int y = current.y;
        int usedLava = current.usedLava;
        int currentCost = current.cost;

        // 如果当前距离大于记录的距离，跳过
        if (currentCost > dist[x][y][usedLava]) {
            continue;
        }

        // 到达终点
        if (x == end.first && y == end.second) {
            totalCost = currentCost;

            // 重建路径
            Path path;
            NodeState curr = current;

            while (curr.x != -1 && curr.y != -1) {
                path.push_back({ curr.x, curr.y });
                curr = prev[curr.x][curr.y][curr.usedLava];
            }

            std::reverse(path.begin(), path.end());
            return path;
        }

        // 探索邻居
        for (auto& neighbor : getNeighborsWithState(current)) {
            int nx = neighbor.x;
            int ny = neighbor.y;
            int nUsedLava = neighbor.usedLava;
            int newCost = neighbor.cost;
```

```
            if (newCost < dist[nx][ny][nUsedLava]) {
                dist[nx][ny][nUsedLava] = newCost;
                prev[nx][ny][nUsedLava] = NodeState(x, y, usedLava, currentCost);
                pq.push(neighbor);
            }
        }
    }

    totalCost = -1;
    return {};
}

Path LujingSousuo::aStarWithOneLava(int& totalCost) {
    // 启发函数：曼哈顿距离
    auto heuristic = [&](int x, int y) {
        return abs(x - end.first) + abs(y - end.second);
        };

    // 三维距离数组
    std::vector<std::vector<std::vector<int>>> gScore(
        rows, std::vector<std::vector<int>>(cols, std::vector<int>(2,
std::numeric_limits<int>::max())));

    std::vector<std::vector<std::vector<NodeState>>> cameFrom(
        rows, std::vector<std::vector<NodeState>>(cols, std::vector<NodeState>(2,
NodeState(-1, -1, -1))));

    // 优先队列：存储(fScore, state)
    using PQElement = std::tuple<int, int, int, int, int>; // fScore, x, y, usedLava, gScore
    std::priority_queue<PQElement, std::vector<PQElement>, std::greater<PQElement>> openSet;

    // 初始化起点
    gScore[start.first][start.second][0] = 0;
    int fScoreStart = heuristic(start.first, start.second);
    openSet.push({ fScoreStart, start.first, start.second, 0, 0 });

    while (!openSet.empty()) {
        auto [fScore, x, y, usedLava, currentGScore] = openSet.top();
        openSet.pop();

        // 到达终点
        if (x == end.first && y == end.second) {
            totalCost = currentGScore;

            // 重建路径
            Path path;
            NodeState curr(x, y, usedLava);

            while (curr.x != -1 && curr.y != -1) {
                path.push_back({ curr.x, curr.y });
                curr = cameFrom[curr.x][curr.y][curr.usedLava];
            }
```

```
            std::reverse(path.begin(), path.end());
            return path;
        }

        // 如果当前gScore不是最新的，跳过
        if (currentGScore > gScore[x][y][usedLava]) {
            continue;
        }

        // 探索邻居
        NodeState currentState(x, y, usedLava, currentGScore);
        for (auto& neighbor : getNeighborsWithState(currentState)) {
            int nx = neighbor.x;
            int ny = neighbor.y;
            int nUsedLava = neighbor.usedLava;
            int tentativeGScore = neighbor.cost;

            if (tentativeGScore < gScore[nx][ny][nUsedLava]) {
                // 找到更好的路径
                cameFrom[nx][ny][nUsedLava] = currentState;
                gScore[nx][ny][nUsedLava] = tentativeGScore;
                int fScoreNew = tentativeGScore + heuristic(nx, ny);
                openSet.push({ fScoreNew, nx, ny, nUsedLava, tentativeGScore });
            }
        }
    }

    totalCost = -1;
    return {};
}
```

然后是随机生成地图的相关方法

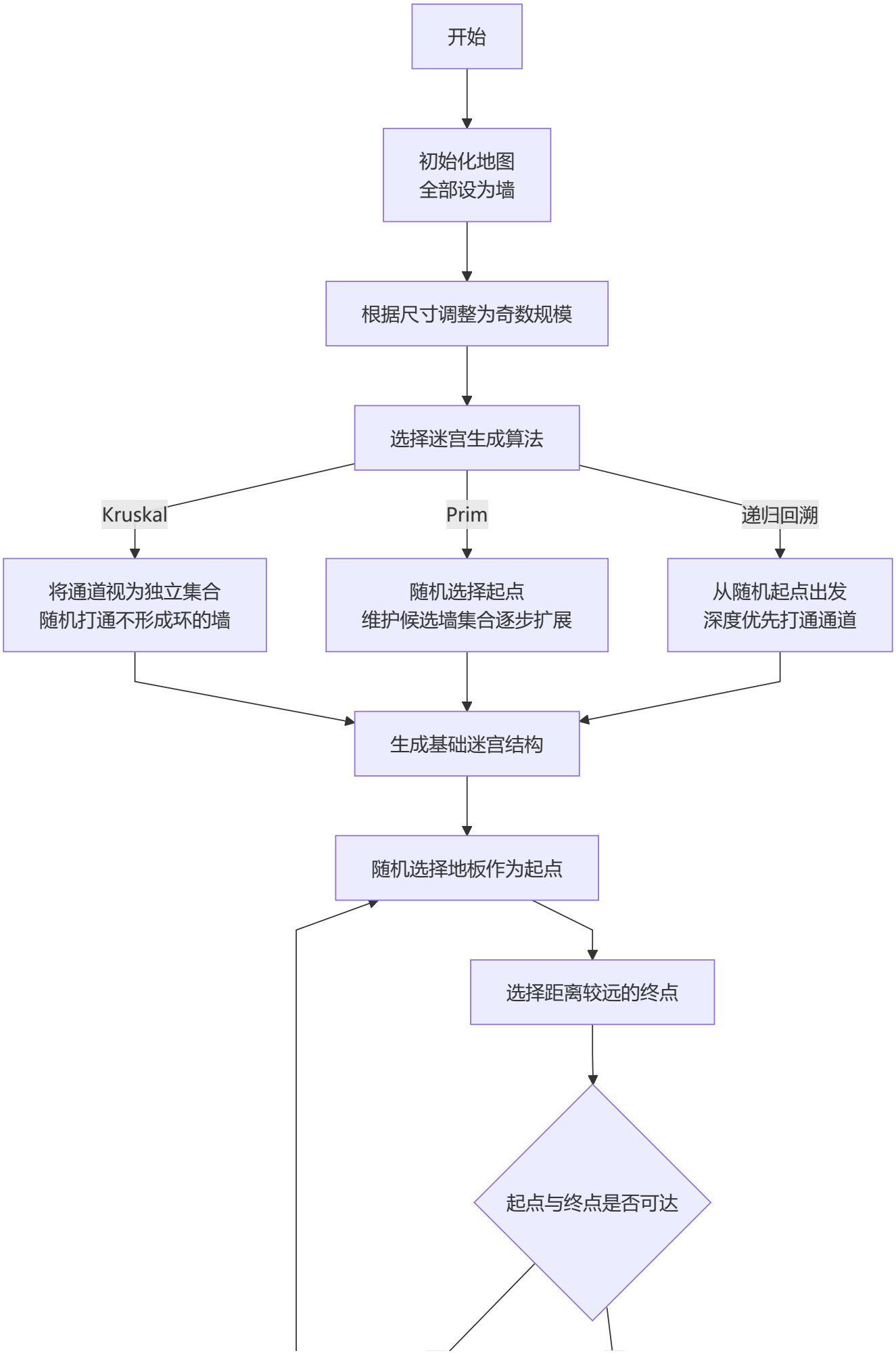
在随机地图生成方面，迷宫生成模块主要负责自动构建结构合理、可通行且具有一定随机性的迷宫地图。整体流程上，首先设置所有方块为墙体，然后使用不同的算法打通，形成一个迷宫，主要有prim和kruskal两种算法

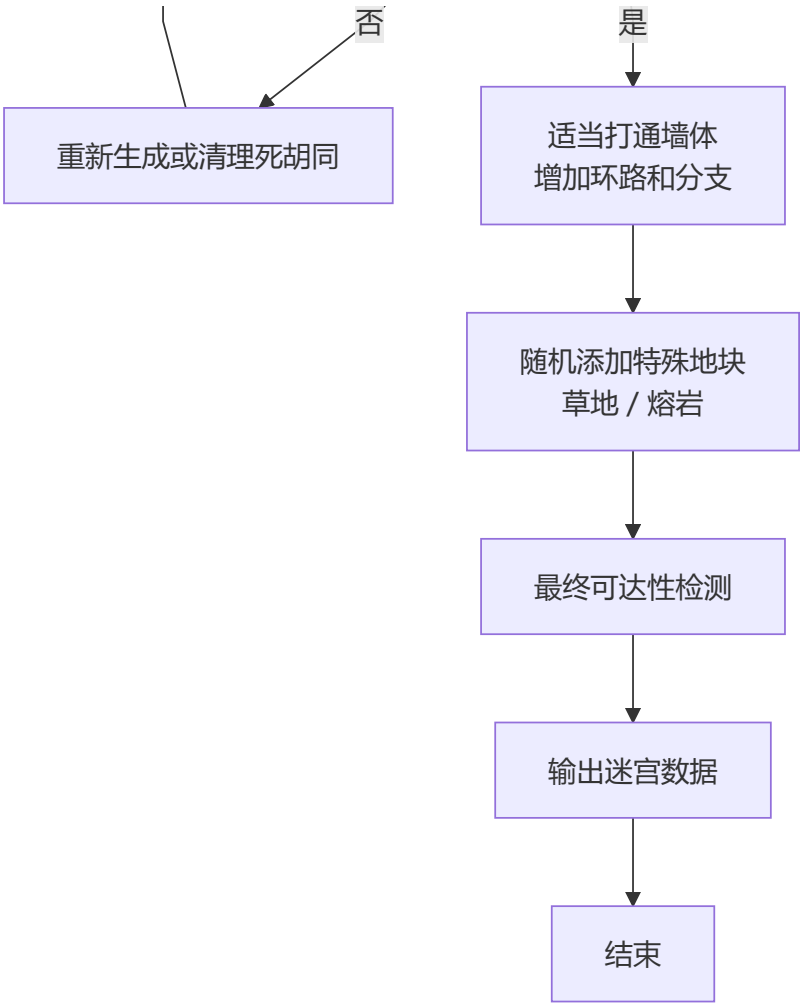
在 Prim 算法生成迷宫 中，从地图中的一个随机通道位置作为起点开始，将该位置加入已生成区域，并把其周围的墙体作为候选边界。随后通过不断随机选择一堵边界墙进行处理，如果该墙连接的另一侧仍是未打通的区域，则打通该墙并将新的通道加入迷宫结构，同时更新其周围的边界墙列表。通过这种方式，迷宫会逐步向外扩展，最终形成整体连通但分支较多的结构。

在 Kruskal 算法生成迷宫中，模块将迷宫中的可通行单元视为独立的集合，并随机打乱所有可能被打通的墙体。生成过程中依次处理这些墙体，如果某堵墙两侧的通道属于不同的集合，则打通该墙并合并对应的集合，从而保证迷宫在逐步连通的同时不会产生冗余连接。该方法生成的迷宫整体结构较为规整，路径分布均匀。

在基础迷宫生成完成后，模块会随机在通道中设置起点和终点位置，随之检测是否有至少一条的可行路径，如果随机生成的起点和终点不可达，模块会多次重新生成，以保证最终地图一定可以正常游玩。

在地块类型扩展方面，在普通通道的基础上随机加入草地和熔岩等特殊地块。这些特殊地块只会生成在原有可通行区域中，并尽量避开起点和终点附近，从而避免影响游戏的基本流程。


开始
初始化地图
全部设为墙
根据尺寸调整为奇数规模
选择迷宫生成算法
Kruskal
Prim
递归回溯
将通道视为独立集合
随机打通不形成环的墙
随机选择起点
维护候选墙集合逐步扩展
从随机起点出发
深度优先打通通道
生成基础迷宫结构
随机选择地板作为起点
选择距离较远的终点
起点与终点是否可达



最后就是寻找一条成本最短的路径了

主要用A*算法来查找这样一条路径：

将“是否踩过熔岩”作为一个变量，整个坐标作为二维变量，合成一个三维的状态。这样在搜索过程中，同一个位置在不同状态下会被当成不同的节点处理，从而避免多次低成本通过熔岩地块的问题。

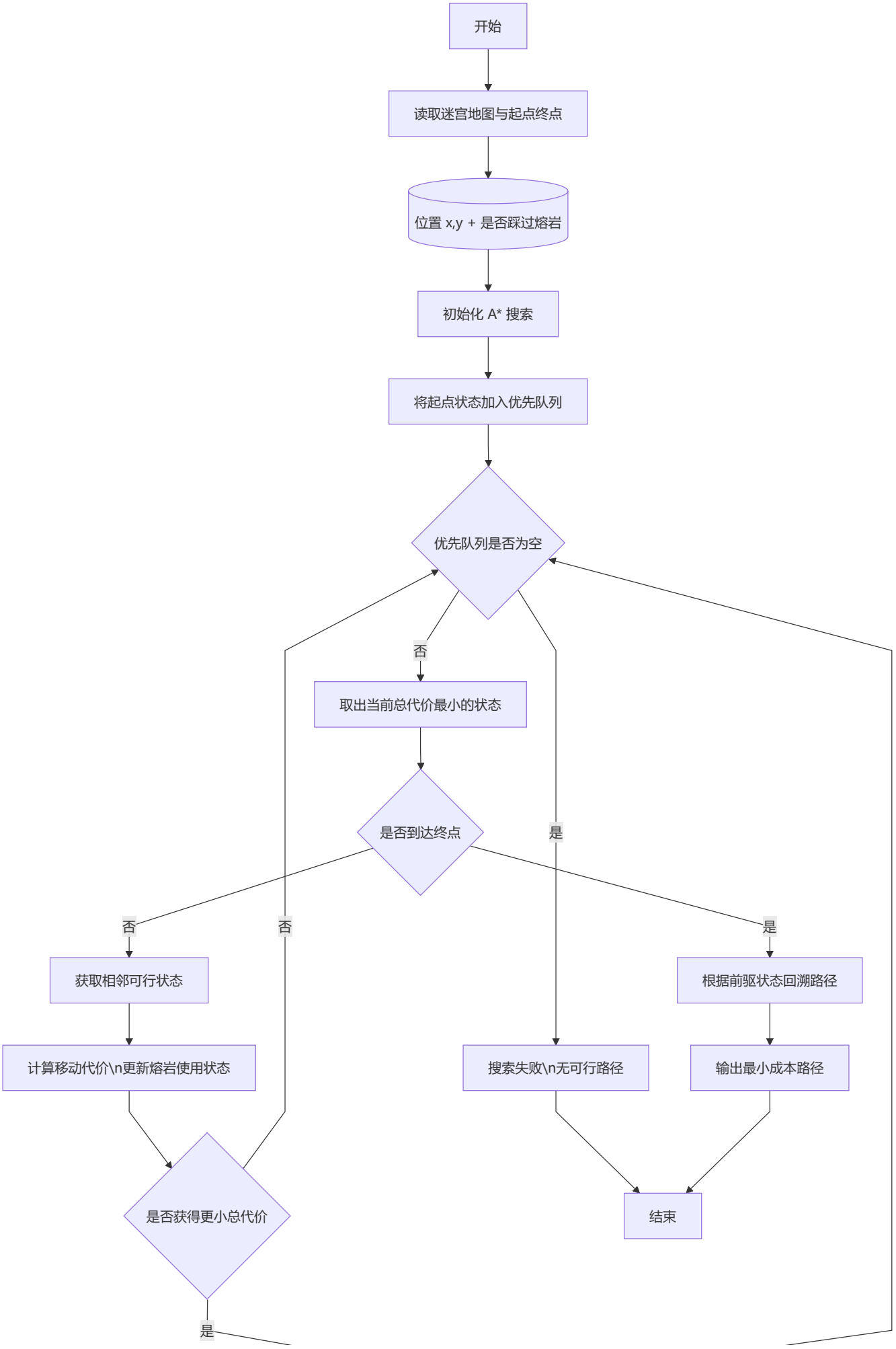
在路径搜索开始前，会根据地图中不同地块的类型设置对应的移动代价。普通地面以及起点、终点的代价较低，草地的代价较高，而熔岩地块在第一次经过时允许进入但代价较大，如果已经踩过一次熔岩，则再次尝试通过时直接视为不可行。

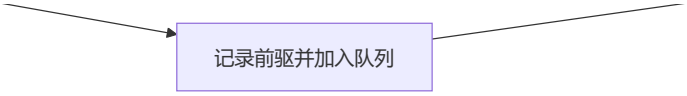
搜索过程中，以起点状态作为初始节点加入优先队列，每次从队列中取出当前总代价最小的状态节点进行扩展。扩展时向上下左右四个方向寻找相邻格子，并根据目标格子的类型计算新的移动代价，同时更新“是否已踩过熔岩”的状态信息。若新的状态在当前条件下获得了更小的总代价，则记录其前驱状态并加入队列继续搜索。

在 A* 算法中引入了启发函数，用当前位置到终点的曼哈顿距离作为估计代价，使搜索方向更偏向终点，从而减少不必要的遍历，提高整体搜索效率。当搜索首次到达终点位置时，即认为找到了在当前规则下成本最小的一条路径，随后通过前驱信息反向回溯，得到完整的行走路径。

如果在搜索结束后仍未到达终点，则说明在当前地图和规则限制下不存在合法路径，此时返回空结果，由上层逻辑进行相应处理。

流程图：


开始
读取迷宫地图与起点终点
位置 x,y + 是否踩过熔岩
初始化 A* 搜索
将起点状态加入优先队列
优先队列是否为空
否
取出当前总代价最小的状态
是否到达终点
是
否
否
是
获取相邻可行状态
计算移动代价\n更新熔岩使用状态
是否获得更小总代价
是
搜索失败\n无可行路径
根据前驱状态回溯路径
输出最小成本路径
结束



相应代码：

```
#include "mazegenerator.h"
#include <fstream>
#include <iostream>
#include <unordered_set>
#include <set>

MazeGenerator::MazeGenerator(int r, int c) : rows(r), cols(c), gen(rd()) {
    maze.resize(rows, std::vector<int>(cols, 1)); // 初始化为全墙
}

bool MazeGenerator::isValid(int x, int y) {
    return x >= 0 && x < rows && y >= 0 && y < cols;
}

void MazeGenerator::recursiveBacktracking(int x, int y, std::vector<std::vector<bool>>&
visited) {
    visited[x][y] = true;
    maze[x][y] = 0; // 设置为通道

    // 四个方向
    int dirs[4][2] = { {-2, 0}, {2, 0}, {0, -2}, {0, 2} };

    // 随机打乱方向
    std::vector<int> indices = { 0, 1, 2, 3 };
    std::shuffle(indices.begin(), indices.end(), gen);

    for (int idx : indices) {
        int nx = x + dirs[idx][0];
        int ny = y + dirs[idx][1];

        if (isValid(nx, ny) && !visited[nx][ny]) {
            // 打通中间的墙
            maze[x + dirs[idx][0] / 2][y + dirs[idx][1] / 2] = 0;
            recursiveBacktracking(nx, ny, visited);
        }
    }
}

void MazeGenerator::primAlgorithm() {
    // 初始化全墙
    maze = std::vector<std::vector<int>>(rows, std::vector<int>(cols, 1));

    // 从随机点开始
    std::uniform_int_distribution<> disX(1, rows - 2);
    std::uniform_int_distribution<> disY(1, cols - 2);

    int startX = disX(gen);
```

```
    int startY = disY(gen);

    // 确保是奇数坐标（为了更好的迷宫结构）
    if (startX % 2 == 0) startX--;
    if (startY % 2 == 0) startY--;

    maze[startX][startY] = 0;

    // 边界墙列表
    std::vector<std::tuple<int, int, int, int>> walls; // (墙x, 墙y, 邻接单元格1x, 邻接单元格1y)

    // 添加初始墙
    if (startX > 1) walls.push_back({ startX - 1, startY, startX - 2, startY });
    if (startX < rows - 2) walls.push_back({ startX + 1, startY, startX + 2, startY });
    if (startY > 1) walls.push_back({ startX, startY - 1, startX, startY - 2 });
    if (startY < cols - 2) walls.push_back({ startX, startY + 1, startX, startY + 2 });

    while (!walls.empty()) {
        // 随机选择一堵墙
        std::uniform_int_distribution<> dis(0, walls.size() - 1);
        int idx = dis(gen);

        auto [wallX, wallY, cellX, cellY] = walls[idx];
        walls.erase(walls.begin() + idx);

        // 如果邻接单元格是墙，则打通
        if (maze[cellX][cellY] == 1) {
            maze[wallX][wallY] = 0;
            maze[cellX][cellY] = 0;

            // 添加新的墙
            int dirs[4][2] = { {-2, 0}, {2, 0}, {0, -2}, {0, 2} };
            for (auto& dir : dirs) {
                int nx = cellX + dir[0];
                int ny = cellY + dir[1];
                int wx = cellX + dir[0] / 2;
                int wy = cellY + dir[1] / 2;

                if (isValid(nx, ny) && maze[nx][ny] == 1) {
                    walls.push_back({ wx, wy, nx, ny });
                }
            }
        }
    }
}

void MazeGenerator::kruskalAlgorithm() {
    // 初始化全墙，每个单元格都是独立的集合
    maze = std::vector<std::vector<int>>(rows, std::vector<int>(cols, 1));

    // 为每个单元格创建并查集
    std::vector<int> parent(rows * cols);
    for (int i = 0; i < rows * cols; i++) {
```

```
        parent[i] = i;
    }

    // 查找函数
    auto find = [&](int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
        };

    // 合并函数
    auto unite = [&](int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            parent[rootY] = rootX;
            return true;
        }
        return false;
        };

    // 收集所有可能的墙（只考虑奇数坐标的单元格）
    std::vector<std::tuple<int, int, int, int>> walls;

    for (int x = 1; x < rows - 1; x += 2) {
        for (int y = 1; y < cols - 1; y += 2) {
            maze[x][y] = 0; // 先设置为通道

            if (x > 1) walls.push_back({ x - 1, y, x, y });
            if (y > 1) walls.push_back({ x, y - 1, x, y });
        }
    }

    // 随机打乱墙
    std::shuffle(walls.begin(), walls.end(), gen);

    // 处理每堵墙
    for (auto [wallX, wallY, cellX, cellY] : walls) {
        int cell1 = (wallX)*cols + wallY;
        int cell2 = cellX * cols + cellY;

        // 如果两个单元格属于不同的集合，打通墙
        if (unite(cell1, cell2)) {
            maze[wallX][wallY] = 0;
        }
    }
}

bool MazeGenerator::isReachable(int startX, int startY, int endX, int endY) {
    if (maze[startX][startY] == 1 || maze[endX][endY] == 1) {
        return false;
```

```
    }

    std::vector<std::vector<bool>> visited(rows, std::vector<bool>(cols, false));
    std::queue<std::pair<int, int>> q;

    q.push({ startX, startY });
    visited[startX][startY] = true;

    int dirs[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        if (x == endX && y == endY) {
            return true;
        }

        for (auto& dir : dirs) {
            int nx = x + dir[0];
            int ny = y + dir[1];

            if (isValid(nx, ny) && !visited[nx][ny] && maze[nx][ny] != 1) {
                visited[nx][ny] = true;
                q.push({ nx, ny });
            }
        }
    }

    return false;
}

std::vector<std::pair<int, int>> MazeGenerator::getAvailableFloorTiles() {
    std::vector<std::pair<int, int>> tiles;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (maze[i][j] == 0) { // 普通地板
                tiles.push_back({ i, j });
            }
        }
    }

    return tiles;
}

std::vector<std::pair<int, int>> MazeGenerator::getAllWallPositions() {
    std::vector<std::pair<int, int>> walls;

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            if (maze[i][j] == 1) {
                walls.push_back({ i, j });
```

```
            }
        }
    }

    return walls;
}

void MazeGenerator::addSpecialTiles(int grassCount, int lavaCount) {
    auto floorTiles = getAvailableFloorTiles();

    if (floorTiles.empty()) return;

    // 随机打乱地板位置
    std::shuffle(floorTiles.begin(), floorTiles.end(), gen);

    // 添加草地
    int grassAdded = 0;
    for (const auto& tile : floorTiles) {
        if (grassAdded >= grassCount) break;

        int x = tile.first;
        int y = tile.second;

        // 确保不是起点或终点附近
        if (abs(maze[x][y]) != 1 && abs(maze[x][y]) != 2) {
            maze[x][y] = 2;
            grassAdded++;
        }
    }

    // 重新获取可用的地板（排除已添加的草地）
    floorTiles = getAvailableFloorTiles();
    std::shuffle(floorTiles.begin(), floorTiles.end(), gen);

    // 添加熔岩
    int lavaAdded = 0;
    for (const auto& tile : floorTiles) {
        if (lavaAdded >= lavaCount) break;

        int x = tile.first;
        int y = tile.second;

        // 确保不是起点或终点附近
        if (abs(maze[x][y]) != 1 && abs(maze[x][y]) != 2) {
            maze[x][y] = 3;
            lavaAdded++;
        }
    }
}

void MazeGenerator::generate(int algorithm, int grassCount, int lavaCount) {
    // 确保尺寸为奇数以便生成更好的迷宫
    if (rows % 2 == 0) rows--;
```

```
    if (cols % 2 == 0) cols--;

    // 重置迷宫
    maze = std::vector<std::vector<int>>(rows, std::vector<int>(cols, 1));

    // 选择算法生成迷宫
    switch (algorithm) {
    case 0: // 递归回溯法
    {
        std::vector<std::vector<bool>> visited(rows, std::vector<bool>(cols, false));
        // 从随机起点开始
        std::uniform_int_distribution<> disX(1, rows - 2);
        std::uniform_int_distribution<> disY(1, cols - 2);
        int startX = disX(gen);
        int startY = disY(gen);
        // 确保是奇数
        if (startX % 2 == 0) startX--;
        if (startY % 2 == 0) startY--;
        recursiveBacktracking(startX, startY, visited);
        break;
    }
    case 1: // Prim算法
        primAlgorithm();
        break;
    case 2: // Kruskal算法
        kruskalAlgorithm();
        break;
    default:
        primAlgorithm();
        break;
    }

    // 生成起点和终点
    generateStartAndEnd();

    // 确保起点和终点可达
    int attempts = 0;
    std::pair<int, int> start = getStart();
    std::pair<int, int> end = getEnd();

    while (!isReachable(start.first, start.second, end.first, end.second) && attempts < 10)
{
        generateStartAndEnd();
        start = getStart();
        end = getEnd();
        attempts++;
    }

    // 如果仍然不可达，进行清理
    if (!isReachable(start.first, start.second, end.first, end.second)) {
        cleanupMaze();
    }
```

```
    // 添加环路和分支增加趣味性
    addLoopsAndBranches(8);

    // 添加特殊地块
    addSpecialTiles(grassCount, lavaCount);

    // 最终确保可达性
    if (!isReachable(start.first, start.second, end.first, end.second)) {
        // 如果还是不可达，重新打通一条路径
        std::vector<std::pair<int, int>> path;
        std::vector<std::vector<bool>> visited(rows, std::vector<bool>(cols, false));
        std::queue<std::pair<std::pair<int, int>, std::vector<std::pair<int, int>>>>  q;

        q.push({ start, {start} });
        visited[start.first][start.second] = true;

        int dirs[4][2] = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

        while (!q.empty()) {
            auto [pos, currentPath] = q.front();
            q.pop();

            if (pos == end) {
                path = currentPath;
                break;
            }

            for (auto& dir : dirs) {
                int nx = pos.first + dir[0];
                int ny = pos.second + dir[1];

                if (isValid(nx, ny) && !visited[nx][ny]) {
                    visited[nx][ny] = true;
                    std::vector<std::pair<int, int>> newPath = currentPath;
                    newPath.push_back({ nx, ny });
                    q.push({ {nx, ny}, newPath });
                }
            }
        }

        // 打通路径
        for (const auto& pos : path) {
            maze[pos.first][pos.second] = 0;
        }
    }
}

std::vector<std::vector<int>> MazeGenerator::getMaze() const {
    return maze;
}

std::pair<int, int> MazeGenerator::getStart() const {
    for (int i = 0; i < rows; i++) {
```

```
        for (int j = 0; j < cols; j++) {
            if (maze[i][j] == -1) {
                return { i, j };
            }
        }
    }
    return { 1, 1 }; // 默认起点
}

std::pair<int, int> MazeGenerator::getEnd() const {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (maze[i][j] == -2) {
                return { i, j };
            }
        }
    }
    return { rows - 2, cols - 2 }; // 默认终点
}

void MazeGenerator::generateStartAndEnd() {
    // 清除可能的旧起点和终点
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (maze[i][j] == -1 || maze[i][j] == -2) {
                maze[i][j] = 0;
            }
        }
    }

    auto floorTiles = getAvailableFloorTiles();
    if (floorTiles.size() < 2) return;

    std::shuffle(floorTiles.begin(), floorTiles.end(), gen);

    // 设置起点
    maze[floorTiles[0].first][floorTiles[0].second] = -1;

    // 设置终点（尽量远离起点）
    int maxDistance = 0;
    std::pair<int, int> bestEnd = floorTiles[1];

    for (size_t i = 1; i < std::min(floorTiles.size(), (size_t)20); i++) {
        int dist = abs(floorTiles[i].first - floorTiles[0].first) +
            abs(floorTiles[i].second - floorTiles[0].second);
        if (dist > maxDistance) {
            maxDistance = dist;
            bestEnd = floorTiles[i];
        }
    }

    maze[bestEnd.first][bestEnd.second] = -2;
}
```

```
void MazeGenerator::cleanupMaze() {
    // 移除死胡同
    bool changed = true;

    while (changed) {
        changed = false;

        for (int i = 1; i < rows - 1; i++) {
            for (int j = 1; j < cols - 1; j++) {
                if (maze[i][j] == 0) { // 只处理普通通道
                    int wallCount = 0;

                    if (maze[i - 1][j] == 1) wallCount++;
                    if (maze[i + 1][j] == 1) wallCount++;
                    if (maze[i][j - 1] == 1) wallCount++;
                    if (maze[i][j + 1] == 1) wallCount++;

                    // 如果三面是墙，这是一个死胡同，可以移除
                    if (wallCount >= 3 && maze[i][j] != -1 && maze[i][j] != -2) {
                        maze[i][j] = 1;
                        changed = true;
                    }
                }
            }
        }
    }
}

void MazeGenerator::addLoopsAndBranches(int extraPaths) {
    auto walls = getAllWallPositions();
    std::shuffle(walls.begin(), walls.end(), gen);

    int pathsAdded = 0;

    for (const auto& wall : walls) {
        if (pathsAdded >= extraPaths) break;

        int x = wall.first;
        int y = wall.second;

        // 检查这堵墙是否可以打通（不会创建太大的开放空间）
        int passageCount = 0;

        if (x > 0 && maze[x - 1][y] != 1) passageCount++;
        if (x < rows - 1 && maze[x + 1][y] != 1) passageCount++;
        if (y > 0 && maze[x][y - 1] != 1) passageCount++;
        if (y < cols - 1 && maze[x][y + 1] != 1) passageCount++;

        // 如果墙的两边都有通道，打通它创建环路
        if (passageCount >= 2) {
            maze[x][y] = 0;
            pathsAdded++;
```

```
        }
    }
}

void MazeGenerator::printMaze() const {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            switch (maze[i][j]) {
            case 1: std::cout << "██"; break;  // 墙
            case 0: std::cout << "  "; break;  // 通道
            case 2: std::cout << "▓▓"; break;  // 草地
            case 3: std::cout << "??"; break;  // 熔岩
            case -1: std::cout << "S "; break; // 起点
            case -2: std::cout << "E "; break; // 终点
            default: std::cout << "??"; break;
            }
        }
        std::cout << std::endl;
    }
}

bool MazeGenerator::saveToFile(const std::string& filename) const {
    std::ofstream file(filename);

    if (!file.is_open()) {
        std::cerr << "无法打开文件: " << filename << std::endl;
        return false;
    }

    // 写入尺寸
    file << rows << " " << cols << std::endl;

    // 写入迷宫数据
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            file << maze[i][j] << " ";
        }
        file << std::endl;
    }

    file.close();
    return true;
}
```

三、测试

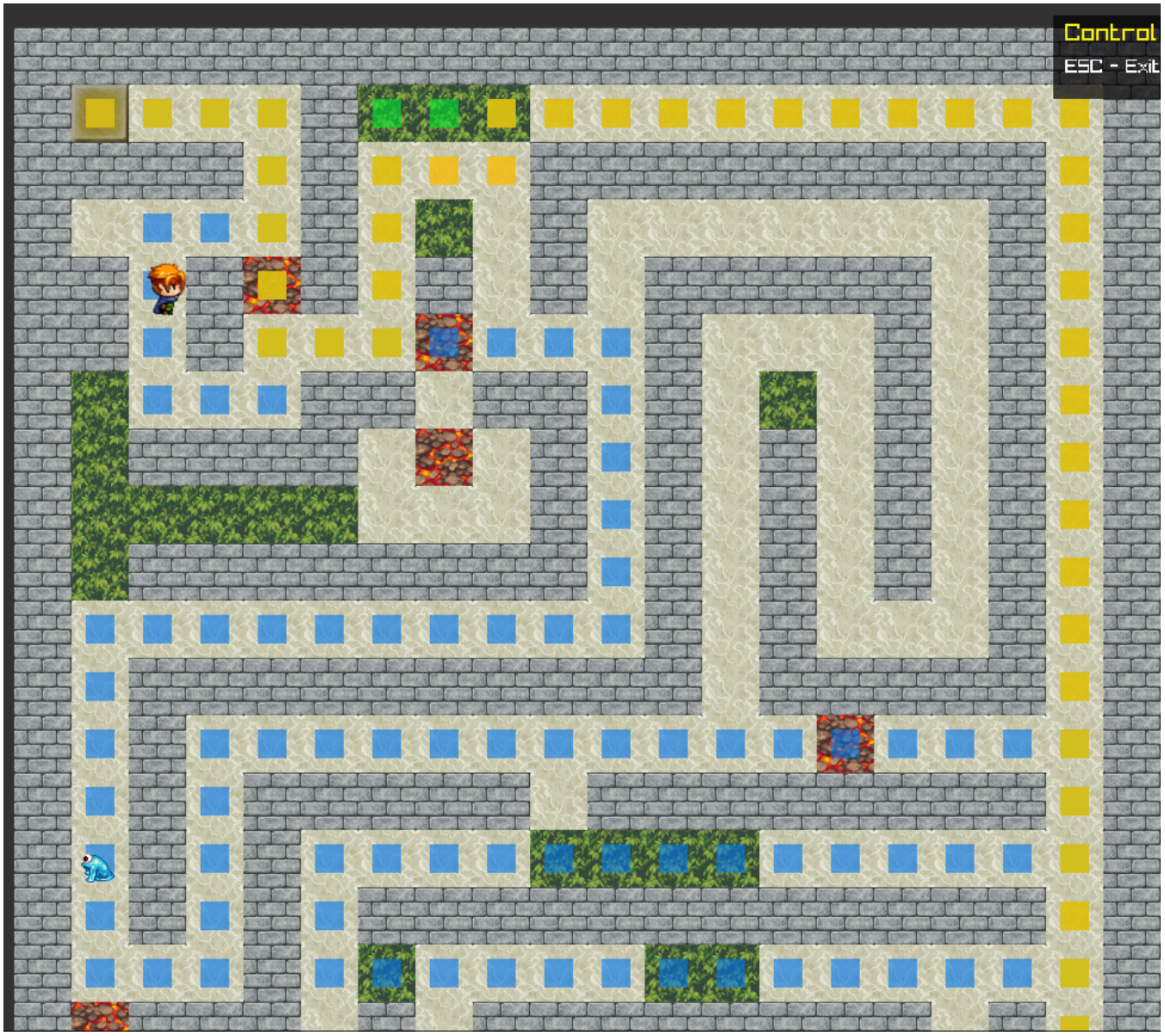
地图模块测试：

主要看能不能正确绘制出相应地图即可

人物部分：

测试碰撞，移动，胜利和失败结算即可，移动到相应位置即可测试功能

碰撞：



胜利结算：


Control instructio
ESC - Exit the game
VICTORY! You reached the destination!
Game ends in 1 seconds
Loaded Images:
wall.png
grass.png
lava.png
floor.png
start.png
end.png
60 FPS

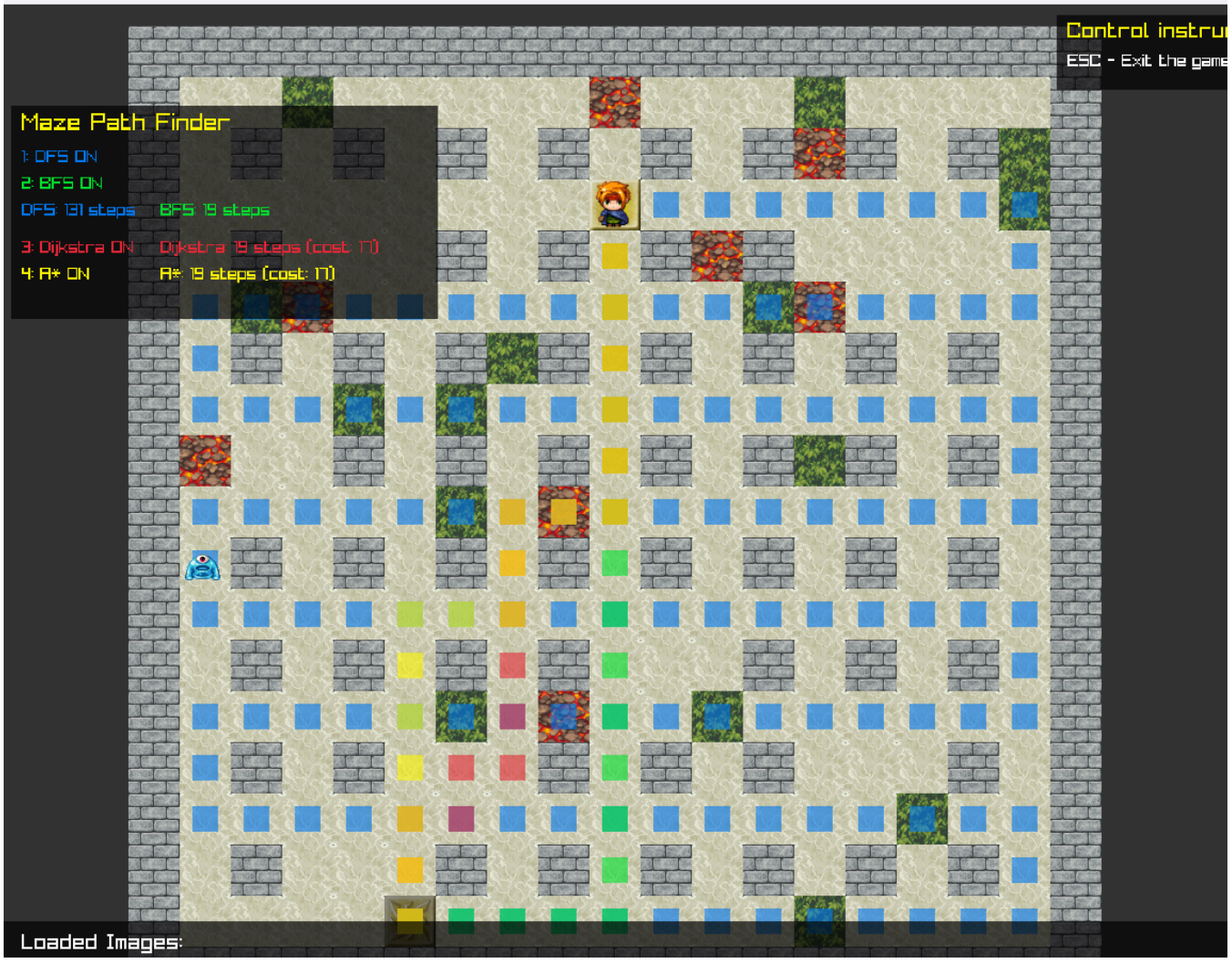
失败结算：


Control instructior
ESC - Exit the game
GAME OVER! You were burned by lava!
Press R to restart the game
Loaded Images:
wall.png
grass.png
lava.png
floor.png
start.png
end.png
60 FPS

算法模块：

测试相关路径的显示，以及随机生成地图的算法即可


Maze Path Finder
1: DFS ON
2: BFS ON
DFS: 37 steps
BFS: 37 steps
3: Dijkstra ON
Dijkstra: 41 steps (cost: 43)
4: A* ON
A*: 41 steps (cost: 43)
Control instructions:
ESC - Exit the game
Loaded Images:
wall.png
grass.png
lava.png
floor.png
start.png
end.png
60 FPS



四、总结

在系统的开发过程中，我学习到了很多有关项目开发的知识。也发现了很多问题，解决这些问题正是我成长的关键。总的来说这是一次受益匪浅的综合项目实践。

第一次着手一个项目，问题当然是有很多的，从一开始拿到项目的不止所措，再到冷静分析，逐步摸清项目需求，再到将项目分为一个个可以执行的代码单元，这一个个问题是我开始学习如何完成这个迷宫项目的关键。如何分类，如何明确每一个类别的目标，再到如何实现，这一步步走来令我印象深刻，同时也让我从中成长了很多。说完了大体的，再来说细分的问题，首先便是地图信息到渲染的转化，人物的渲染以及原理，还有相关内容，很多都对我来说是陌生的，同时，其中对数据结构的熟练度要求很高，在写代码的时候，往往稍有不慎就容易出错，而且修改起来越到后面越是繁琐，一段代码牵连的内容越多，修改起来也就越困难。所以注释，变量名都很值的考究，如果能够见名知意，那样也可以节约很多的时间。同时在编写bfs，dfs，等等算法的的时候，合理使用各种数据结构合理释放资源，合理调动各个函数之间的关系都非常重要，这关乎到你能不能让项目正常运作。

还有就是要做一些适合调试的事情，往往代码不能够一下子就解决，需要很多调试，来看当前代码板块的内容有没有实现正常的功能，所以写一些适合调试的代码也很关键。

在这次的项目实践中，令人欣慰的是最终还是按时完成了项目相关的所有要求，包括地图的绘制，人物的行走，特异地块的功能，npc的设置，游戏的胜利和失败，bfs，dfs算法寻路，以及随机生成地图和寻找最小成本路径，最终都得到了较好的完成。

在项目完成中也是有一些问题，特别是当涉及到一些比较难以完成的算法时，出现了举手无措的情况，最终还是依靠大数据才得以完成这一功能，这说明在算法的学习上，我仍然存在很大不足，需要继续精进才能满足以后日常代码学习过程中的要求。

总的来说，这是一次收获颇丰的一次项目实践，整个迷宫小游戏虽然在内容上比较多，但是确实是实打实的锻炼了我的代码能力，也让我学会了更多的代码知识与细节，为我往后的学习做好了铺垫。

参考资料

无。