数据结构实践

课程报告

**题 目： 基于unity的算法与数据结构可视游戏化项目**

**院 （系）： 计算机科学与技术学院**

**专业班级： 2023级计算机类4班**

**学号： 2022463010911**

**姓名： 匡剑衡**

**指导老师： 卢安**

**起止时间： 2024年9月-2024年12月**

目录

[1项目需求分析 1](#_Toc12605)

[1.1 算法与数据结构可视游戏化项目介绍 1](#_Toc30671)

[1.2 算法与数据结构可视游戏化项目用户分析 1](#_Toc23479)

[1.3 算法与数据结构可视游戏化项目功能介绍 1](#_Toc6295)

[1.4 算法与数据结构可视游戏化项目具体功能需求分析 4](#_Toc14812)

[1.4.1 迷宫系统-迷宫生成 4](#_Toc22860)

[1.4.2 迷宫系统-迷宫更新 4](#_Toc4210)

[1.4.3 迷宫系统-迷宫重置 5](#_Toc3341)

[1.4.4 迷宫系统-迷宫交互 5](#_Toc18881)

[1.4.5 玩家系统-视角切换 5](#_Toc29192)

[1.4.6 玩家系统-玩家控制 6](#_Toc15422)

[1.4.7 玩家系统-随身光源 6](#_Toc14827)

[1.4.8 玩家系统-相机跟随 6](#_Toc31653)

[1.4.9 UI系统-信息面板 6](#_Toc22311)

[1.4.10 UI系统-演示面板 7](#_Toc12221)

[1.4.11 UI系统-结束面板 7](#_Toc15642)

[1.4.12 UI系统-退出面板 8](#_Toc1624)

[1.4.13 算法演示-深度优先搜索(DFS) 8](#_Toc29324)

[1.4.14 算法演示-广度优先搜索(BFS) 8](#_Toc16983)

[1.4.15 算法演示-可视化控制 9](#_Toc4398)

[2项目总体设计 10](#_Toc14258)

[2.1项目结构设计 10](#_Toc25082)

[2.2数据结构设计 10](#_Toc17620)

[2.2.1主要类设计 10](#_Toc1444)

[2.2.2数据结构设计 15](#_Toc7117)

[3项目详细设计 17](#_Toc20857)

[3.1迷宫系统详细设计 17](#_Toc554)

[3.1.1迷宫生成功能详细设计 17](#_Toc3175)

[3.1.1.1 DFS生成 17](#_Toc9329)

[3.1.1.2 Kruskal生成 18](#_Toc21756)

[3.1.1.3 Prim生成 18](#_Toc153)

[3.1.1.4 递归切割生成 18](#_Toc25808)

[3.1.2迷宫管理功能详细设计 19](#_Toc1344)

[3.1.2.1控制迷宫的生成和重置 19](#_Toc1251)

[3.1.2.2管理迷宫的可视化显示 20](#_Toc13684)

[3.1.2.3处理迷宫与玩家的交互 20](#_Toc25591)

[3.2 玩家系统详细设计 21](#_Toc25113)

[3.2.1 摄像头跟随 21](#_Toc1312)

[3.2.1.1 第一人称视角 21](#_Toc21621)

[3.2.1.2 俯视角 21](#_Toc19238)

[3.2.1.3 视角更新 22](#_Toc20276)

[3.2.2 玩家控制器 22](#_Toc2073)

[3.2.2.1 玩家控制 22](#_Toc21999)

[3.2.2.2 状态管理 23](#_Toc27596)

[3.2.2.3 随身光源 23](#_Toc12616)

[3.2.3 玩家管理器 23](#_Toc31760)

[3.2.3.1 玩家管理 23](#_Toc30899)

[3.2.3.2 光照控制 24](#_Toc8132)

[3.2.3.3 其他控制 24](#_Toc30672)

[3.3 UI系统详细设计 25](#_Toc23393)

[3.3.1 游戏UI管理器 25](#_Toc13117)

[3.3.1.1 UI面板管理 25](#_Toc4844)

[3.3.1.2 事件处理 26](#_Toc12614)

[3.3.1.3 可视化控制 26](#_Toc1141)

[3.3.2 可视化信息面板 26](#_Toc3303)

[3.3.2.1 状态显示 26](#_Toc1927)

[3.3.2.2 面板控制 27](#_Toc19953)

[3.3.3 游戏控制面板 27](#_Toc16704)

[3.3.3.1 UI组件初始化 27](#_Toc5980)

[3.3.3.2 事件管理 28](#_Toc14816)

[3.3.3.3 按钮交互 28](#_Toc1279)

[3.4 算法演示详细设计 29](#_Toc26638)

[3.4.1 深度优先搜索（DFS） 30](#_Toc7246)

[3.4.2 广度优先搜索（BFS） 30](#_Toc10033)

[3.5主要类图关系 30](#_Toc4051)

[3.6 游戏流程详细设计 31](#_Toc31104)

[3.7界面层设计 32](#_Toc21356)

[4系统实现与测试 35](#_Toc24529)

[4.1迷宫生成算法测试 35](#_Toc13454)

[4.2可视化算法演示测试 37](#_Toc11349)

[4.3 按键监听测试 39](#_Toc12599)

[5总结 41](#_Toc13150)

[参考文献 43](#_Toc6237)

[附录：核心代码 45](#_Toc31215)

# 1项目需求分析

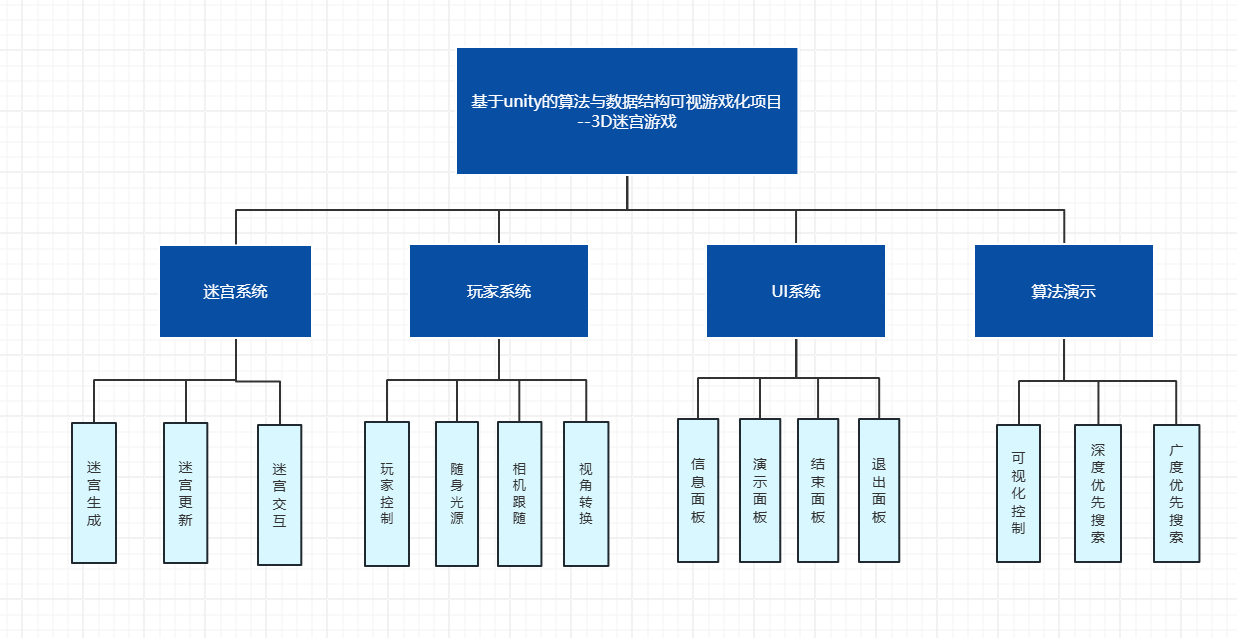
## 算法与数据结构可视游戏化项目介绍

随着计算机技术的持续进步和游戏化学习理念的日益普及，为了促进学生和编程爱好者更高效地学习和掌握算法与数据结构的精髓，基于Unity平台的算法与数据结构可视化游戏化项目应运而生。该项目旨在打造一个界面直观、操作简便的系统，通过互动式可视化技术展示常见的数据结构和算法，例如迷宫生成算法和路径查找算法等。系统集成了数据结构和算法的动画演示和玩家游玩等多种功能，使用户能够通过亲自操作和观察算法的执行过程，深入洞察每个算法背后的原理和应用场景。通过游戏化的任务和挑战，用户不仅能够提升算法思维能力，还能在完成任务的过程中获得成就感，从而提高学习兴趣和效率。这是一个基于Unity的3D迷宫游戏项目，通过游戏化的方式展示和学习数据结构与算法。

## 算法与数据结构可视游戏化项目用户分析

本项目是一个学习型游戏。目标用户群体：编程初学者、学生、教师、对算法和数据结构感兴趣的技术爱好者、迷宫游戏爱好者等。

## 算法与数据结构可视游戏化项目功能介绍



**图1-1 项目功能结构图**

项目主要分为迷宫系统、玩家系统、UI系统、算法演示模块，具体说明如下：

1. 迷宫系统

迷宫生成：

使用四种算法生成随机迷宫

- DFS递归生成

- Prim最小生成树

- Kruskal并查集生成

- 递归分割法

迷宫更新：

地板等的可视化更新

迷宫的重置

迷宫交互：

迷宫与玩家的交互

1. 玩家系统

视角切换：

第一人称/俯视角自由切换

玩家控制：

- WASD移动控制

- 鼠标视角控制

- 物理碰撞检测

- 玩家重置动画

随身光源：

随身光源效果

L键开启/关闭随时光源

相机跟随：

平滑相机跟随

第一人称与俯视角各自的相机跟随

1. UI系统

渐变动画效果

信息面板：

三栏式信息显示

- 执行状态

- 操作指南

- 算法说明

Tab键显示/隐藏界面

演示面板：

常驻面板，可以点击对应的按钮进行对应的功能

结束面板：

当玩家到达终点的时候弹出相对应的提示

退出面板：

玩家ESC键弹出，确实是否退出游戏

1. 算法演示

深度优先搜索(DFS)：

- 实时路径可视化

- 回溯路径显示

- 搜索效率优化

广度优先搜索(BFS)：

- 实时路径可视化

- 最短路径查找

- 平滑移动效果

- 优化路径显示

可视化控制：

路径颜色标记

- 黄色：当前访问

- 灰色：已访问

- 绿色：最终路径

- 红色：玩家路径

## 算法与数据结构可视游戏化项目具体功能需求分析

### 迷宫系统-迷宫生成

**功能介绍：**系统支持四种不同的迷宫生成算法，每种算法能够生成不同类型的迷宫，以供用户在游戏中进行路径查找和挑战。由系统随机数随机选择不同的算法生成迷宫。

**（1）输入**

选择生成算法：系统可以选择不同的算法随机生成迷宫。

DFS递归生成

Prim最小生成树

Kruskal并查集生成

递归分割法

**（2）输出**

生成的随机迷宫：根据系统随机选择的算法，系统生成并展示相应的迷宫。

### 迷宫系统-迷宫更新

**功能介绍：**在迷宫生成后，系统支持迷宫的动态更新，包括地板、墙体等可视化更新，确保游戏体验流畅。

**（1）输入**

玩家操作：玩家与迷宫的互动（例如移动、重置迷宫等）会触发迷宫的更新。

**（2）输出**

更新后的迷宫界面：迷宫的墙体、路径、地面等元素会根据玩家的互动进行更新，确保游戏体验真实。

### 迷宫系统-迷宫重置

**功能介绍：**用户可以随时重置迷宫，生成一个新的迷宫供玩家挑战。

**（1）输入**

重置命令：玩家按下“1键”触发迷宫重置。

**（2）输出**

新的迷宫界面：重新生成的迷宫，会根据随机选择的算法重新布局。

### 迷宫系统-迷宫交互

**功能介绍：**玩家可以与迷宫进行互动，探索路径，执行各种操作，如移动、使用算法演示等。

**（1）输入**

玩家输入：玩家通过键盘和鼠标进行角色控制、视角切换等操作。

**（2）输出**

迷宫与玩家交互界面：玩家的每个操作都会实时影响迷宫的显示，例如路径的更新、玩家位置的移动等。

### 玩家系统-视角切换

**功能介绍：**系统提供第一人称和俯视角两种视角，玩家可以随时切换，选择适合自己的游戏体验方式，进行多方面的观察。

**（1）输入**

按键输入：“V键”切换第一人称与俯视角。

**（2）输出**

切换后的视角界面：根据玩家选择的视角，系统调整相机位置和视角显示。

### 玩家系统-玩家控制

**功能介绍：**玩家通过键盘和鼠标控制角色的移动和视角，游戏支持WASD键盘控制和鼠标视角控制。

**（1）输入**

WASD键：控制角色移动。

鼠标：控制视角方向。

R：启动重置动画，重置玩家位置。

**（2）输出**

玩家实时控制：玩家通过操作键盘和鼠标，角色和视角随之变化。

### 玩家系统-随身光源

**功能介绍：**玩家在黑暗的迷宫中有一个随身光源，帮助玩家更好地进行迷宫探索，默认开启光源。

**（1）输入**

L键：开启或关闭随身光源。

**（2）输出**

光源效果：开启后，玩家周围区域会被照亮，提升可视范围。

### 玩家系统-相机跟随

**功能介绍：**系统提供平滑的相机跟随效果，使得玩家在移动时，相机平滑跟随玩家。

**（1）输入**

玩家移动：玩家控制角色移动时，相机会自动平滑跟随。

**（2）输出**

相机跟随效果：相机平滑地跟随玩家，保持合适的视角和位置。

### UI系统-信息面板

**功能介绍：**系统提供一个信息面板，实时显示迷宫执行状态、操作指南以及算法说明。

**（1）输入**

按键输入：玩家按下Tab键，控制信息面板的显示或隐藏。

**（2）输出**

信息面板界面：显示三栏式的面板，包含执行状态、操作指南和算法说明。

操作说明

- `WASD` - 移动角色

- `鼠标移动` - 第一人称视角控制

- `鼠标右键` - 俯视角旋转

- `V键` - 切换第一人称/俯视角

- `1键` - 重新生成迷宫

- `2键` - 深度优先搜索演示

- `3键` - 广度优先搜索演示

- `R键` - 重置位置和路径

- `Tab键` - 显示/隐藏信息面板

- `L键` - 开关聚光灯

- `Alt键` - 显示/隐藏鼠标

- `ESC键` - 退出游戏

### UI系统-演示面板

**功能介绍：**系统会提供常驻演示面板，玩家可以点击相应按钮启动不同的功能，包括迷宫重新生成、深度优先搜索演示、广度优先搜索演示等。

**（1）输入**

点击按钮：玩家通过点击面板上的按钮，触发不同的功能。

**（2）输出**

功能启动：点击后相应功能会被启动。

### UI系统-结束面板

**功能介绍：**当玩家成功到达迷宫的终点时，弹出提示面板，告知玩家挑战成功。弹出窗口后，点击任意位置可关闭该面板。

**（1）输入**

玩家到达终点：玩家成功完成迷宫挑战。

**（2）输出**

结束面板：展示玩家的挑战结果，提示玩家成功通关。

### UI系统-退出面板

**功能介绍：**当玩家按下ESC键时，弹出退出确认面板，玩家可以选择是否退出游戏。点击确认退出游戏，取消则回到游戏状态。

**（1）输入**

ESC键：玩家按下退出键。

**（2）输出**

退出确认面板：确认是否退出游戏的提示面板。

### 算法演示-深度优先搜索(DFS)

**功能介绍：**系统支持深度优先搜索算法的演示，实时显示路径的探索过程，展示回溯的路径。演示时，迷宫地板同时更新。演示完毕后，玩家根据路径进行移动。

**（1）输入**

启动DFS演示：玩家点击相应按钮或按键2，启动DFS算法演示。

**（2）输出**

DFS路径可视化：实时展示DFS算法的路径生成过程和回溯的路径。

### 算法演示-广度优先搜索(BFS)

**功能介绍：**系统支持广度优先搜索算法的演示，实时展示最短路径查找，并优化路径的显示效果。演示时，迷宫地板同时更新。演示完毕后，玩家根据路径进行移动。

**（1）输入**

启动BFS演示：玩家点击相应按钮或按键3，启动BFS算法演示。

**（2）输出**

BFS路径可视化：展示BFS算法从起点到终点的路径搜索过程，标记路径的最短路线。

### 算法演示-可视化控制

**功能介绍：**系统通过路径颜色标记来展示算法的执行状态，帮助玩家清晰地理解算法的进程。

**（1）输入**

算法执行：根据算法的执行，路径节点会被标记不同的颜色。

**（2）输出**

路径颜色标记：路径的节点根据状态显示不同颜色，帮助玩家区分已访问、当前访问和最终路径。

路径颜色标记：

黄色：当前访问的节点

灰色：已访问的节点

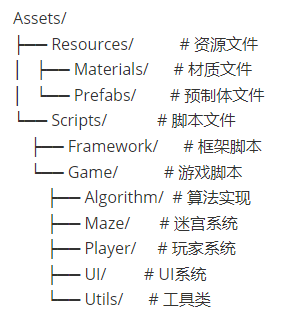
绿色：最终路径

红色：玩家路径

# 2项目总体设计

## 2.1项目结构设计

本系统是桌面应用程序，项目结构如下图所示。



**图2-1 项目结构**

## 2.2数据结构设计

### 2.2.1主要类设计

1. 迷宫单元格类：表示迷宫中的一个基本单元格，包含位置和状态信息。

/// <summary>

/// 迷宫单元格类

/// 表示迷宫中的一个基本单元格，包含位置和状态信息

/// </summary>

/// <remarks>

/// 主要功能：

/// 1. 状态管理：

/// - IsWall：是否为墙壁

/// - IsVisited：是否已访问

/// - IsPath：是否为路径

/// - IsEnd：是否为终点

/// 2. 视觉效果：

/// - IsLit：是否被照亮

/// - HighlightColor：高亮颜色

/// - FloorMaterialType：地板材质类型

///

/// 使用方式：

/// - 通过构造函数创建新单元格

/// - 使用属性访问和修改状态

/// - 调用Reset()重置单元格

/// </remarks>

public class MazeCell

{

public int X { get; private set; }

public int Y { get; private set; }

public bool IsWall { get; set; }

public bool IsVisited { get; set; }

public bool IsLit { get; set; }

public bool IsEnd { get; set; }

public GameObject CellObject { get; set; }

public bool IsPath { get; set; }

public Color HighlightColor { get; set; }

public MazeCell(int x, int y)

{

X = x;

Y = y;

IsWall = true;

IsVisited = false;

IsLit = false;

IsEnd = false;

CellObject = null;

IsPath = false;

HighlightColor = Color.white;

}

public void Reset()

{

IsWall = true;

IsVisited = false;

IsPath = false;

HighlightColor = Color.white;

}

}

public enum FloorMaterialType

{

Default,

PathFloor,

HighlightFloor,

PlayerMaterial

}

1. 迷宫寻路算法基类：定义寻路算法的通用接口和基本功能。

/// <summary>

/// 迷宫寻路算法基类

/// 定义寻路算法的通用接口和基本功能

/// </summary>

/// <remarks>

/// 主要功能：

/// 1. 寻路控制：

/// - StartPathFinding()：开始寻路

/// - StopPathFinding()：停止寻路

/// - ResetPath()：重置路径

/// 2. 路径管理：

/// - HighlightPath()：高亮显示路径

/// - MarkVisited()：标记已访问单元格

/// 3. 状态通知：

/// - OnPathFound：找到路径时的回调

/// - OnVisitCell：访问单元格时的回调

///

/// 使用方式：

/// - 继承此类实现具体的寻路算法

/// - 重写FindPath()方法实现算法逻辑

/// - 通过事件系统通知UI更新

/// </remarks>

public abstract class PathFinder

{

protected MazeCell[,] maze;

protected List<Vector2Int> path = new List<Vector2Int>();

protected Vector2Int startPos;

protected Vector2Int endPos;

protected bool[,] visited;

/// <summary>可移动的四个方向：上、右、下、左</summary>

protected static readonly Vector2Int[] directions = new Vector2Int[]

{

new Vector2Int(-1, 0),

new Vector2Int(0, 1),

new Vector2Int(1, 0),

new Vector2Int(0, -1)

};

protected bool pathFound = false;

protected List<Vector2Int> finalPath = new List<Vector2Int>();

public PathFinder(MazeCell[,] maze)

{

this.maze = maze;

this.startPos = new Vector2Int(1, 1);

this.endPos = new Vector2Int(maze.GetLength(0) - 2, maze.GetLength(1) - 2);

this.visited = new bool[maze.GetLength(0), maze.GetLength(1)];

}

public abstract IEnumerator FindPath();

public List<Vector2Int> GetPath()

{

return path;

}

protected bool IsValid(Vector2Int pos)

{

return pos.x >= 0 && pos.x < maze.GetLength(0) &&

pos.y >= 0 && pos.y < maze.GetLength(1) &&

!maze[pos.x, pos.y].IsWall &&

!visited[pos.x, pos.y];

}

}

1. 迷宫生成器类：负责使用多种算法生成随机迷宫。

/// <summary>

/// 迷宫生成器类

/// 负责使用多种算法生成随机迷宫

/// </summary>

/// <remarks>

/// 主要功能：

/// 1. 迷宫生成：

/// - DFS算法：深度优先搜索生成

/// - Prim算法：最小生成树法

/// - Kruskal算法：并查集生成

/// - 递归分割：空间递归划分

/// 2. 迷宫处理：

/// - SetBoundaries()：设置边界

/// - SetStartAndEnd()：设置起终点

/// - InitializeMaze()：初始化迷宫

///

/// 使用方式：

/// - 创建MazeGenerator实例

/// - 调用GenerateMaze()生成迷宫

/// - 随机选择一种生成算法

/// </remarks>

public partial class MazeGenerator

{

private MazeCell[,] maze;

private int width;

private int height;

private System.Random random;

/// <summary>可移动方向数组：上、右、下、左</summary>

private readonly int[,] directions = new int[,] { { -1, 0 }, { 0, 1 }, { 1, 0 }, { 0, -1 } };

// 定义迷宫生成算法类型

private enum GenerationAlgorithm

{

DFS,

Prim,

Kruskal,

RecursiveDivision

}

public MazeGenerator()

{

random = new System.Random();

}

/// <summary>

/// 生成新的迷宫

/// </summary>

/// <param name="mazeData">迷宫数据数组</param>

public void GenerateMaze(MazeCell[,] mazeData)

{

maze = mazeData;

width = maze.GetLength(0);

height = maze.GetLength(1);

InitializeMaze();

// 随机选择一种算法

GenerationAlgorithm algorithm = (GenerationAlgorithm)random.Next(4);

Debug.Log($"Using algorithm: {algorithm}");

switch (algorithm)

{

case GenerationAlgorithm.DFS:

CarvePassagesDFS(1, 1);

break;

case GenerationAlgorithm.Prim:

PrimGeneration();

break;

case GenerationAlgorithm.Kruskal:

KruskalGeneration();

break;

case GenerationAlgorithm.RecursiveDivision:

// 先将所有格子设为通道

for (int x = 0; x < width; x++)

{

for (int y = 0; y < height; y++)

{

// 设置边界为墙

if (x == 0 || x == width - 1 || y == 0 || y == height - 1)

{

maze[x, y].IsWall = true;

}

else

{

maze[x, y].IsWall = false;

}

}

}

RecursiveDivision(1, 1, width - 2, height - 2);

break;

}

SetBoundaries();

SetStartAndEnd();

}

}

### 2.2.2数据结构设计

1. 采用栈（先进后出）配合字典和哈希集合作为数据结构来实现DFS算法演示

private Stack<Vector2Int> stack = new Stack<Vector2Int>();

private Vector2Int currentExploring;

private int exploredCount = 0;

private Dictionary<Vector2Int, Vector2Int> parent = new Dictionary<Vector2Int, Vector2Int>();

private HashSet<Vector2Int> inStack = new HashSet<Vector2Int>();

public DFSPathFinder(MazeCell[,] maze) : base(maze)

{

currentExploring = startPos;

}

（2）采用队列（先进先出）配合字典作为数据结构来实现BFS算法演示

private Queue<Vector2Int> queue = new Queue<Vector2Int>();

private Dictionary<Vector2Int, Vector2Int> parent = new Dictionary<Vector2Int, Vector2Int>();

private Vector2Int currentExploring;

private int exploredCount = 0;

private Queue<Vector2Int> movementPath = new Queue<Vector2Int>();

public BFSPathFinder(MazeCell[,] maze) : base(maze)

{

currentExploring = startPos;

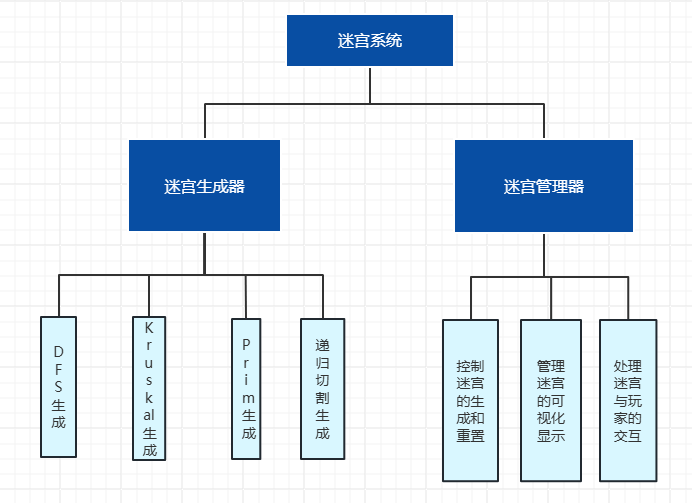
}

（3）设计二维数组存放地图数据，迷宫生成器可根据获得的地图数据实例化迷宫。

private MazeCell[,] maze;

# 3项目详细设计

## 3.1迷宫系统详细设计



**图3-1迷宫系统结构图**

### 3.1.1迷宫生成功能详细设计

#### 3.1.1.1 DFS生成

private void CarvePassagesDFS(int x, int y)

**返回类型：无返回值；**

**是否含参数：是，地图坐标**

**步骤：**

算法流程：

1. 从起点开始递归访问

2. 随机选择未访问的相邻单元格

3. 打通当前单元格与选中单元格的墙

4. 递归处理选中的单元格

5. 回溯处理其他方向

#### 3.1.1.2 Kruskal生成

### private void KruskalGeneration()

**返回类型：无返回值；**

**是否含参数：无,这是部分类的方法，直接使用的类里面的地图数据**

**步骤：**

算法流程：

1. 初始化所有单元格为独立集合

2. 收集所有可能的边

3. 随机打乱边的顺序

4. 依次处理每条边：

- 检查两端单元格是否连通

- 若不连通则合并集合并打通墙壁

#### 3.1.1.3 Prim生成

private void PrimGeneration()

**返回类型：无返回值；**

**是否含参数：无,这是部分类的方法，直接使用的类里面的地图数据**

**步骤：**

算法流程：

1. 从起点开始扩展

2. 维护边界单元格列表

3. 随机选择边界单元格

4. 连接到已访问区域

5. 更新边界列表

### 3.1.1.4 递归切割生成

private void RecursiveDivision(int x1, int y1, int x2, int y2)

**返回类型：无返回值；**

**是否含参数：有，两个点坐标**

**步骤：**

算法流程：

1. 从空房间开始

2. 递归地添加墙壁分割空间：

- 随机选择分割方向

- 在分割线上开通道

- 递归处理子区域

3. 保证通路：

- 检查是否有通道

- 保护起点和终点区域

### 3.1.2迷宫管理功能详细设计

#### 3.1.2.1控制迷宫的生成和重置

public void GenerateMaze()

**返回类型：无返回值；**

**是否含参数：无**

**步骤：**

1. 停止当前协程
2. 清理现有迷宫墙体
3. 清理起点和终点效果
4. 启动迷宫生成协程
5. 初始化迷宫容器和迷宫网格
6. 调用迷宫生成器生成迷宫
7. 生成迷宫视觉效果
8. 设置玩家初始位置
9. 播放迷宫生成音效

#### public void ResetAll()

**返回类型：无返回值；**

**是否含参数：无**

**步骤：**

1. 停止所有协程
2. 重置路径可视化
3. 启用玩家控制，确保重置后玩家可以正常操控
4. 更新算法可视化器状态，通过 UI 告知用户迷宫已经重置，可以重新开始操作。

#### 3.1.2.2管理迷宫的可视化显示

private IEnumerator CreateMazeVisuals()

**返回类型：**返回值类型是一个对象，它表示一个可枚举的集合中的元素**；**

**是否含参数：无**

**步骤：**

1. 计算迷宫的起始位置
2. 遍历迷宫网格创建地板，设置属性并加载和应用地板材质
3. 将生成的地板对象关联到迷宫的相应单元格中
4. 异步创建迷宫的墙壁，设置属性并加载和应用对应的材质

#### 3.1.2.3处理迷宫与玩家的交互

public void StartPathFinding(bool useDFS = true)

**返回类型：无返回值；**

**是否含参数：是，是否使用DFS，否就使用BFS**

**步骤：**

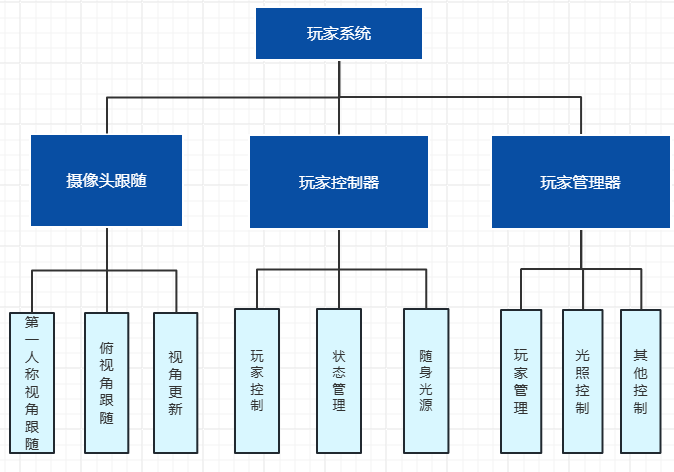
1. 停止当前所有协程，确保路径搜索的操作能够从头开始
2. 计算玩家在迷宫网格中的位置并检查玩家是否在起始位置

- 如果玩家不在起始位置（默认为 1, 1），则输出日志并重置玩家到起始位置，之后延迟启动路径搜索。

- 如果玩家已经在起始位置，则直接调用 StartPathFindingInternal 方法开始路径搜索。

1. 重置路径搜索的可视化效果，清除之前的路径信息
2. 根据选择的算法创建路径查找器
3. 启动路径查找协程
4. 执行路径查找，如果找到路径，开始移动玩家协程
5. 遍历路径并移动玩家

## 3.2 玩家系统详细设计



**图3-2玩家系统结构图**

### 3.2.1 摄像头跟随

负责控制相机跟随玩家移动和视角切换以及更新

#### 3.2.1.1 第一人称视角

public void SetFirstPersonView()

处理流程：在游戏开始时，默认设置摄像机为第一人称视角，切换到第一人称视角时，设置相关的偏移量和移动平滑速度。

#### 3.2.1.2 俯视角

public void SetTopDownView()

处理流程：切换到俯视视角时，设置偏移量和移动平滑速度。

计算俯视角的旋转，保持摄像机位于目标上方，并根据目标的朝向设置旋转，使得目标的前方总是对准屏幕的上方。

确保摄像机始终俯视目标，并且保持玩家的朝向。

#### 3.2.1.3 视角更新

private void LateUpdate()

处理流程：每帧调用，确保摄像机更新发生在其他对象（例如玩家）移动之后，从而避免抖动或位置不一致的问题。更新摄像机位置和旋转：

第一人称视角，摄像机的旋转与目标的旋转相同，使摄像机始终面向目标的方向。

俯视视角，摄像机保持在俯视模式下，使用 topDownRotation 旋转。

### 3.2.2 玩家控制器

负责处理玩家的输入和移动控制

#### 3.2.2.1 玩家控制

private void OnPlayerMove(Vector2 input)

处理流程：玩家移动，根据输入的 Vector2 控制玩家的移动，y 控制前后，x 控制左右。

private void OnMouseMove(Vector2 mouseDelta)

处理流程：鼠标控制，根据鼠标的移动旋转玩家视角，支持第一人称视角和俯视视角的旋转操作。

private void Start()

处理流程：初始化操作，获取 mainCamera 和 Rigidbody 组件，并设置 Rigidbody 的物理参数。锁定并隐藏鼠标，防止玩家在第一人称视角下干扰视角控制。注册 PlayerMove 和 MouseMove 事件，以处理玩家的移动和鼠标移动事件。初始化聚光灯的状态。

private void Update()

处理流程：每帧调用，监听玩家输入

监听 Alt 键，用于显示和隐藏鼠标，暂停或恢复玩家控制。

监听 V 键切换视角模式（第一人称与俯视视角）。

监听 1, 2, 3 键触发不同的迷宫操作：生成迷宫、开始路径搜索（DFS 或 BFS）、重置迷宫。

监听 Esc 键退出游戏，并弹出退出面板。

监听 L 键控制聚光灯的开关。

调用 CheckAndLightFloor() 来检测并照亮玩家脚下的地面，调用 CheckGameFinish() 来检查游戏是否结束。

#### 3.2.2.2 状态管理

public void EnableControl()

public void DisableControl()

玩家启用控制与禁用控制

private void OnCollisionEnter(Collision collision)

处理流程：碰撞检测，与标记的物体发生碰撞时，触发游戏对应事件

#### 3.2.2.3 随身光源

private void CheckAndLightFloor()

处理流程：通过发射一条射线来检测玩家脚下的地面，并根据地面的位置来更新迷宫中的一个“地板”状态。

1.从玩家位置向下发射射线。

2.获取射线碰撞的点。

3.计算该点相对于迷宫中心的网格坐标。

4.根据网格坐标更新迷宫中对应的格子。

### 3.2.3 玩家管理器

负责管理玩家对象的生命周期和状态

#### 3.2.3.1 玩家管理

public void Init()

处理流程：初始化玩家对象

1.加载玩家预制体，通过 ResourcesManager.GetInstance().LoadAsync 异步加载一个预制体（"Prefabs/Player"）并实例化它。加载完成后，player 会被赋值为加载的玩家预制体，并设置 tag 为 "Player"。随后调用 SetupPlayer() 来设置玩家的组件和属性。

2.为玩家添加必要的组件，检查 player 是否已经添加了 PlayerController、Rigidbody、CapsuleCollider 组件，若没有则动态添加。为玩家设置物理属性，如冻结旋转、设置质量、拖拽等。

3.设置相机跟随，获取主摄像机并设置其 CameraFollow 脚本的 target 为玩家对象，使得相机始终跟随玩家。

4.添加聚光灯，在玩家对象上创建一个新的游戏对象（PlayerSpotLight），并为其添加一个聚光灯 (Light)，设置其类型为 Spot，并调整范围、角度、强度和颜色。

private void OnUpdate()

处理流程：处理玩家的输入更新

移动输入，检查 W, S, A, D 键是否按下，通过这些键来控制玩家的水平和垂直移动。如果有移动输入，触发事件中心，发布玩家的移动事件。

鼠标输入，获取鼠标的移动偏移量 Mouse X 和 Mouse Y，并触发事件中心，发布鼠标移动事件。

public void ResetPlayer()

处理流程：重置玩家状态，通过 ResetPlayer 方法启动一个协程 (ResetAnimation)，实现玩家对象的重置动画。

玩家先上升一段距离，然后消失并在起始位置短暂消失，最后下降到原位置，完成动画。

#### 3.2.3.2 光照控制

public void SetSpotLightEnabled(bool enabled)

public void SetSpotLightRange(float range)

public void SetSpotLightAngle(float angle)

public void SetSpotLightIntensity(float intensity)

处理流程：提供了控制聚光灯的开关、范围、角度和强度的方法。

#### 3.2.3.3 其他控制

通过提供的方法控制玩家状态

public void SetPlayerPosition(Vector3 position, bool resetRotation = false)

处理流程：设置玩家位置，允许设置玩家的初始位置，并可以选择是否重置玩家的旋转。如果设置 resetRotation = true，会将玩家的旋转重置为 Quaternion.identity（即没有旋转），并重置摄像机的旋转。

public Vector3 GetPlayerPosition()

提供获取玩家位置的方法

public void SetPlayerRotation(Quaternion rotation)

public Quaternion GetPlayerRotation()

提供设置玩家旋转和获取玩家旋转的方法

public void SetMoveSpeed(float speed)

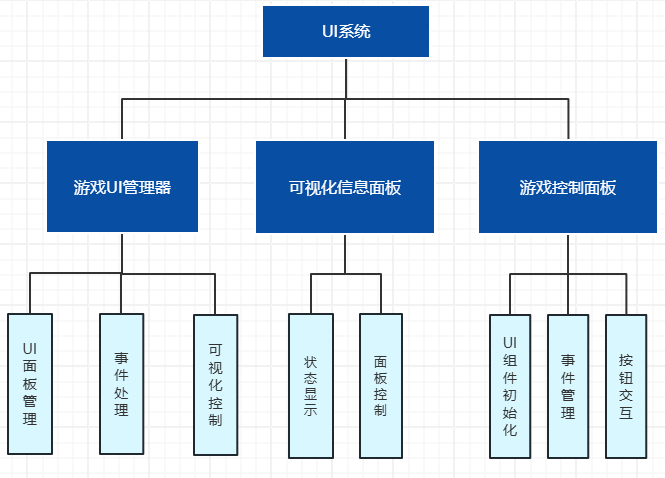
public float GetMoveSpeed()

提供设置玩家速度和获取玩家速度的方法

public GameObject GetPlayer()

提供获取玩家对象的方法

## 3.3 UI系统详细设计



**图3-3 UI系统结构图**

### 3.3.1 游戏UI管理器

负责管理和协调所有游戏UI面板，协调和管理游戏中各个UI面板的显示和交互。使用单例模式 (BaseManager<T>) 来管理UI面板，并处理不同的UI逻辑。实现对UI的集中管理，并通过 UIManager 和 EventCenter 系统来进行面板的显示、状态更新和交互。

#### 3.3.1.1 UI面板管理

public void Init()

处理流程：加载UI面板，通过 UIManager.GetInstance().ShowPanel 异步加载两个UI面板MazeGameUI 和 AlgorithmVisualizerUI，并分别指定其显示层级。

当加载完成后，分别为这两个UI面板设置透明度为0，使其初始时不可见。

private void SetPanelTransparency(GameObject panel, float alpha)

处理流程：通过获取 Image 组件并修改其 color.a 值来调整面板的透明度。

public void SetUITransparency(string panelName, float alpha)

提供按面板名称设置透明度方法，通过 panelName 判断要设置透明度的面板是哪一个，然后调用 SetPanelTransparency 方法调整透明度。

#### 3.3.1.2 事件处理

private void OnGameFinish()

显示完成面板，当游戏完成时，触发 OnGameFinish 方法，显示游戏完成面板并播放完成音效。

public void ShowExitConfirmPanel()

显示退出面板，通过 MazeGameUI 中的 ShowExitConfirmPanel 方法来显示退出确认的面板。

#### 3.3.1.3 可视化控制

public void IncrementSteps()

步骤：调用 AlgorithmVisualizerUI 中的 IncrementSteps 方法来增加步骤。

public void ResetVisualizer()

步骤：调用 AlgorithmVisualizerUI 中的 ResetSteps 方法来重置可视化器。

### 3.3.2 可视化信息面板

负责显示算法执行过程的实时状态和操作说明等信息

#### 3.3.2.1 状态显示

public void ShowDFSInfo()

public void ShowBFSInfo()

public void ShowOperationGuide()

展示深度优先搜索（DFS）和广度优先搜索（BFS）的说明，展示每种算法的工作原理和演示时使用的颜色标识。

public void UpdateStatus(string status)

更新当前执行状态，包括显示当前的执行步骤，并提示用户按R键可以重置演示。

public void ResetInfo()

提供重置所有信息的方法

#### 3.3.2.2 面板控制

private void InitComponents()

初始化了UI面板的各个组件，如：背景面板、显示状态、操作指南、算法信息的文本框。动态创建和初始化UI组件，确保面板和文本能够显示所需内容

private void TogglePanel()

切换面板显示状态（显示或隐藏）。

流程步骤：

切换isPanelVisible的值（从true变为false，或反之）。

如果bgPanel不为空，启动FadePanel()协程，渐变面板的透明度。

private IEnumerator FadePanel(bool fadeIn)

实现透明度渐变效果，背景面板和文本框的透明度都随着渐变变化。

1.初始化渐变的持续时间和已消耗的时间。

2.设置bgPanel的显示状态：如果fadeIn为true，则显示bgPanel，否则不显示。获取bgPanel内所有子文本和背景图片，保存它们的颜色信息。

3.通过循环逐步调整面板和文本的透明度，计算当前透明度alpha，根据fadeIn状态渐变透明度。更新背景色和文本颜色的透明度。等待下一帧。在循环结束时，确保透明度完全调整为目标值。

### 3.3.3 游戏控制面板

负责管理游戏主要面板的UI元素和交互，通过管理按钮、面板、事件和用户交互，协调游戏逻辑的展示和操作，使玩家能够通过UI控制迷宫的生成、算法的执行和游戏的重置，确保UI的互动性和流畅性。

#### 3.3.3.1 UI组件初始化

protected override void Awake()

初始化类实例。

1.调用基类 BasePanel.Awake()。

2.检查是否存在 EventSystem，若不存在则创建新的 EventSystem 对象。

调用 InitComponents() 初始化UI组件。

3.添加事件监听器：

启用UI交互。

禁用UI交互。

迷宫完成时显示完成面板。

private void InitComponents()

初始化UI组件。

步骤：

调用 InitButtonPanel() 初始化按钮面板。

调用 InitFinishPanel() 初始化完成面板。

调用 InitExitConfirmPanel() 初始化退出确认面板。

private void InitButtonPanel()

初始化按钮面板并添加各个功能按钮。

步骤：

1.创建按钮面板 buttonPanel 并设置位置和大小。

2.设置按钮面板的透明度和允许接收射线检测。

3.创建四个按钮：GenerateBtn、DFSBtn、BFSBtn 和 ResetBtn，并设置每个按钮的位置和大小。

private void InitFinishPanel()

初始化完成面板。

步骤：

1.创建完成面板，设置大小、透明度。

2.创建一个按钮来关闭面板。

3.创建两个文本组件，显示“到达终点！”。显示“按R键重置，点击任意处关闭”提示。

4.设置面板初始为隐藏。

private void InitExitConfirmPanel()

初始化退出确认面板。

步骤：

1.创建退出确认面板，设置大小和透明度。

2.创建文本显示“确认退出？”。

3.创建一个按钮容器，并在容器中创建两个按钮：确认退出按钮和取消退出按钮。

4.设置面板初始为隐藏。

#### 3.3.3.2 事件管理

private void OnEnableUIInteraction()

private void OnDisableUIInteraction()

用于启用和禁用UI交互。禁用时，按钮的点击事件会被忽略。

private void OnMazeCompleted()

迷宫完成时显示完成面板

#### 3.3.3.3 按钮交互

protected override void OnClick(string btnName)

处理按钮点击事件。

步骤：

1.输出日志 Button clicked: {btnName}。

2.检查 canInteract 是否为 true（如果为 false，则不处理点击事件）。

3.根据按钮名称执行不同的操作：

GenerateBtn：调用 MazeManager.GetInstance().GenerateMaze() 生成迷宫。

DFSBtn：调用 MazeManager.GetInstance().StartPathFinding(true) 启动深度优先搜索。

BFSBtn：调用 MazeManager.GetInstance().StartPathFinding(false) 启动广度优先搜索。

ResetBtn：调用 MazeManager.GetInstance().ResetAll() 重置迷宫。

ExitConfirmBtn：调用 QuitGame() 退出游戏。

ExitCancelBtn：调用 HideExitConfirmPanel() 隐藏退出确认面板。

public void ShowFinishPanel()

public void HideFinishPanel()

用于显示和隐藏游戏完成面板。完成时显示该面板，提示玩家按 R 键重置游戏。

public void ShowExitConfirmPanel()

public void HideExitConfirmPanel()

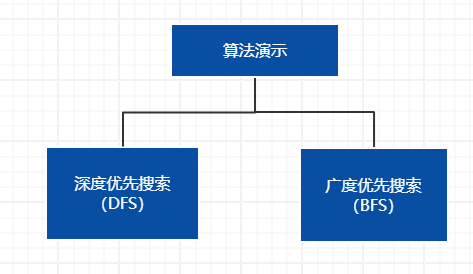
用于显示和隐藏退出确认面板。

private void QuitGame()

退出游戏。

步骤：如果是在 Unity 编辑器中，停止播放。如果是构建版本，退出应用。

## 3.4 算法演示详细设计



**图3-4算法演示模块结构图**

### 3.4.1 深度优先搜索（DFS）

public override IEnumerator<YieldInstruction> FindPathStepByStep()

使用DFS策略在迷宫中寻找路径

算法流程：

1. 从起点开始深度搜索：

标记当前单元格为已访问

递归探索未访问的相邻单元格

回溯处理死路

2. 路径处理：

使用栈记录访问路径

回溯时移除无效路径

找到终点时保存路径

### 3.4.2 广度优先搜索（BFS）

public override IEnumerator<YieldInstruction> FindPathStepByStep()

使用BFS策略在迷宫中寻找最短路径

算法流程：

1. 层级遍历：

使用队列存储待访问单元格

按距离递增顺序访问

记录每个单元格的前驱节点

2. 路径构建：

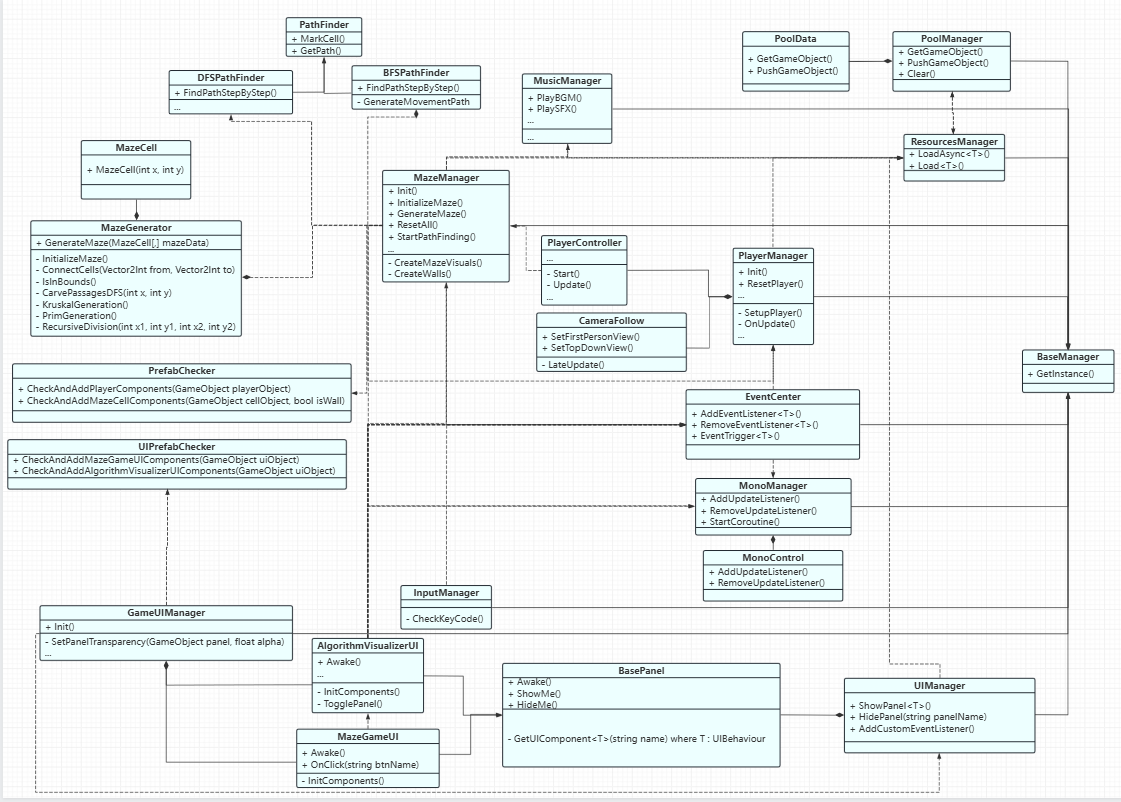
找到终点后回溯构建路径

使用前驱节点信息

反向生成最终路径

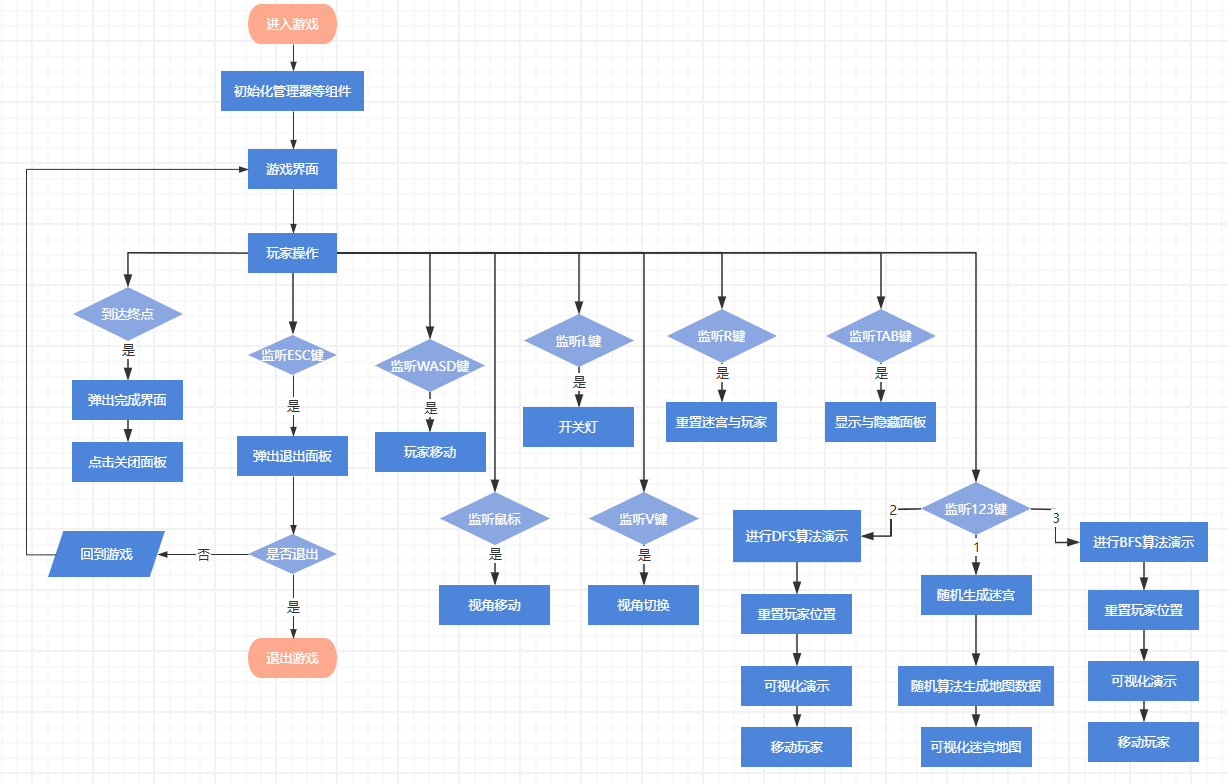
## 3.5主要类图关系

实现箭头表示继承，虚线箭头表示依赖，实线菱形表示组成



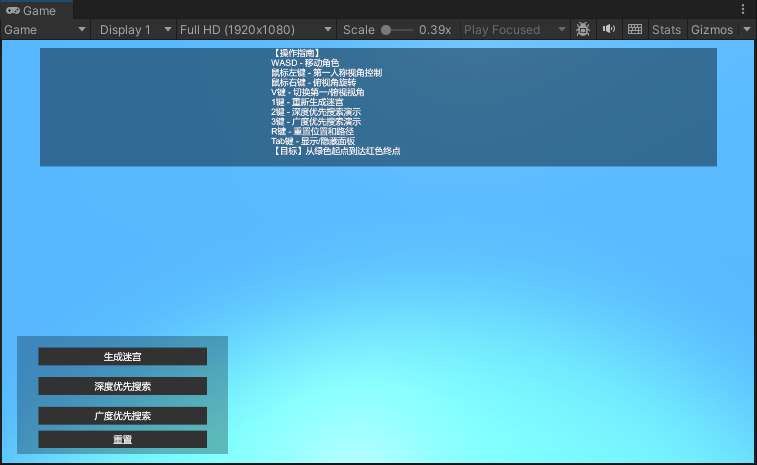
**图3-5 主要类图**

## 3.6 游戏流程详细设计



**图3-6 游戏流程图**

## 3.7界面层设计

本项目是基于unity游戏引擎，使用C#语言进行开发。界面使用UGUI。

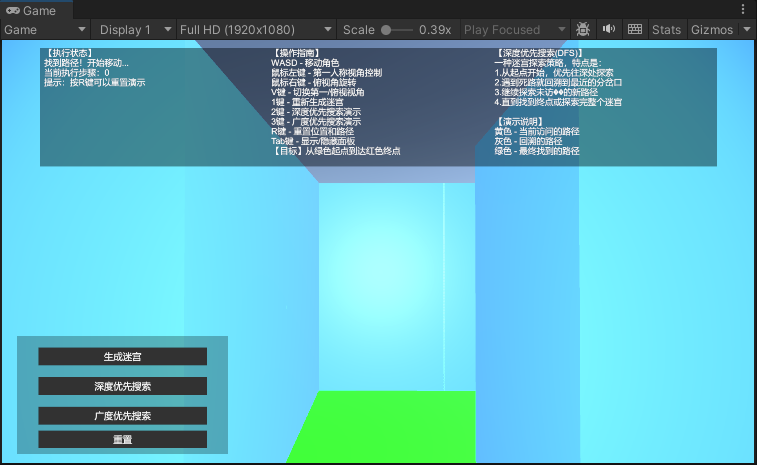
**图3-7-1 初始界面**



**图3-7-2 重置后界面**

# 

**图3-7-3 算法演示界面（俯视角）**



**图3-7-4 算法演示界面（第一人称视角）**



**图3-7-5 结束界面**

# 

**图3-7-6 退出界面**

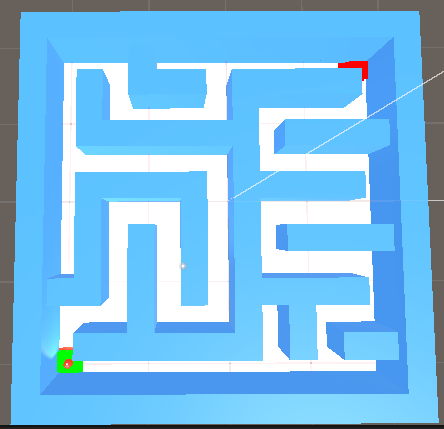
# 4系统实现与测试

## 4.1迷宫生成算法测试

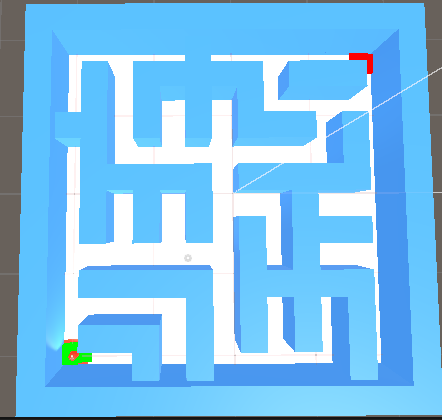
测试用例如表4-1所示。

**表4-1**

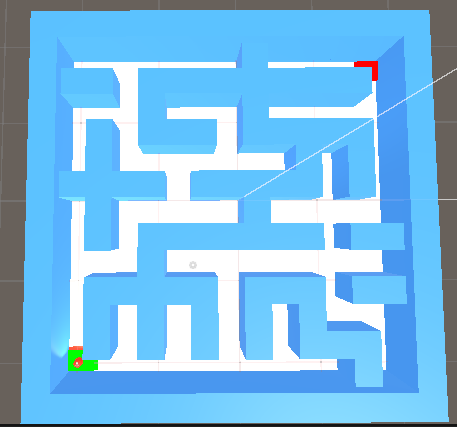
|  |  |  |  |
| --- | --- | --- | --- |
| **测试模块名称** | **输入** | **输出** | **测试结果** |
| DFS生成 | 玩家输入按键1 | 迷宫地图 | 通过 |
| Kurskal算法生成 | 玩家输入按键1 | 迷宫地图 | 通过 |
| Prim算法生成 | 玩家输入按键1 | 迷宫地图 | 通过 |
| 递归切割算法生成 | 玩家输入按键1 | 迷宫地图 | 通过 |



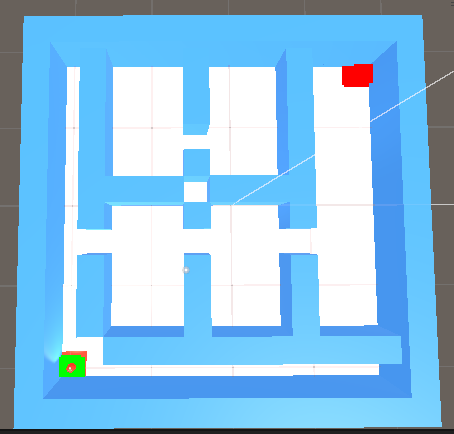
**图4-1 DFS生成迷宫**



**图4-2 Kruskal算法生成迷宫**



**图4-3 Prim算法生成迷宫**



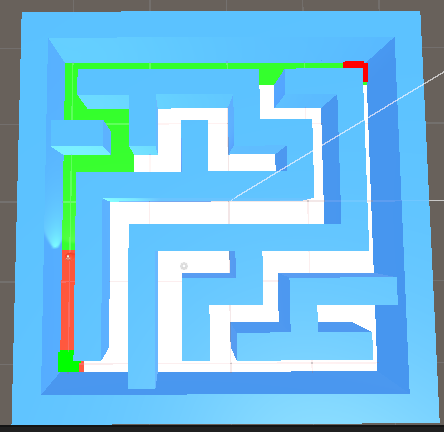
**图4-4 递归切割生成迷宫**

## 4.2可视化算法演示测试

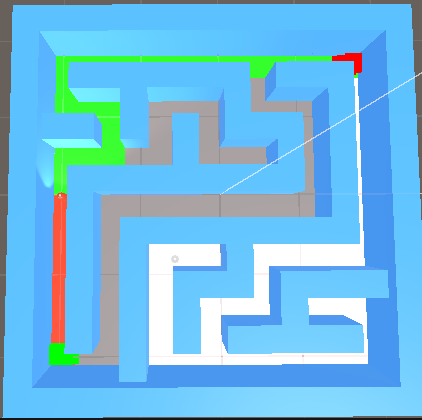
测试用例如表4-2所示。

**表4-2**

|  |  |  |  |
| --- | --- | --- | --- |
| **测试模块名称** | **输入** | **输出** | **测试结果** |
| DFS演示 | 玩家输入按键2 | DFS算法演示 | 通过 |
| BFS演示 | 玩家输入按键3 | BFS算法演示 | 通过 |



**图4-5 DFS算法演示**



**图4-6 BFS算法演示**

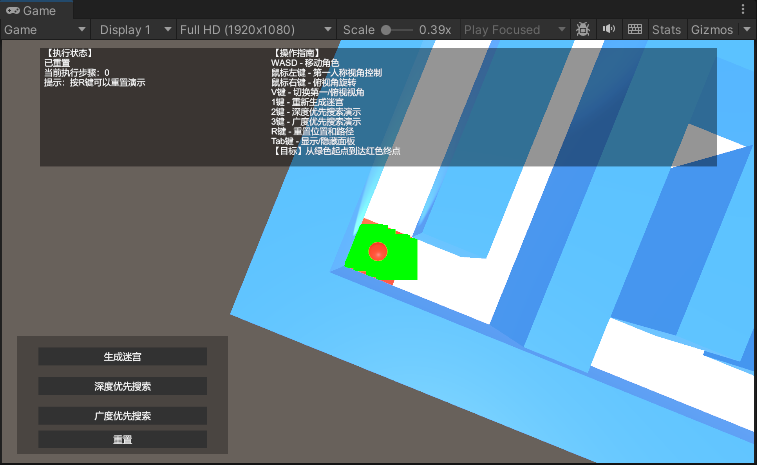
## 4.3 按键监听测试

测试用例如表4-3所示。

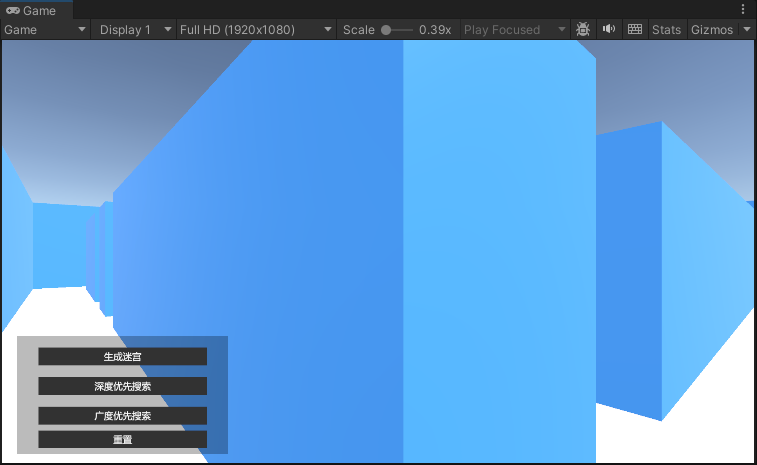
**表4-3**

|  |  |  |  |
| --- | --- | --- | --- |
| **测试模块名称** | **输入** | **输出** | **测试结果** |
| R键重置 | 玩家输入按键R | 重置 | 通过 |
| V键进行视角切换 | 玩家输入按键V | 视角切换 | 通过 |
| L键开关灯源 | 玩家输入按键L | 灯源变化 | 通过 |
| TAB键显隐信息面板 | 玩家输入按键TAB | 显隐面板 | 通过 |
| ALT呼出鼠标 | 玩家长按按键ALT | 呼出鼠标 | 通过 |
| 控制面板按钮监听 | 玩家点击面板的按钮 | 进行对应操作 | 通过 |

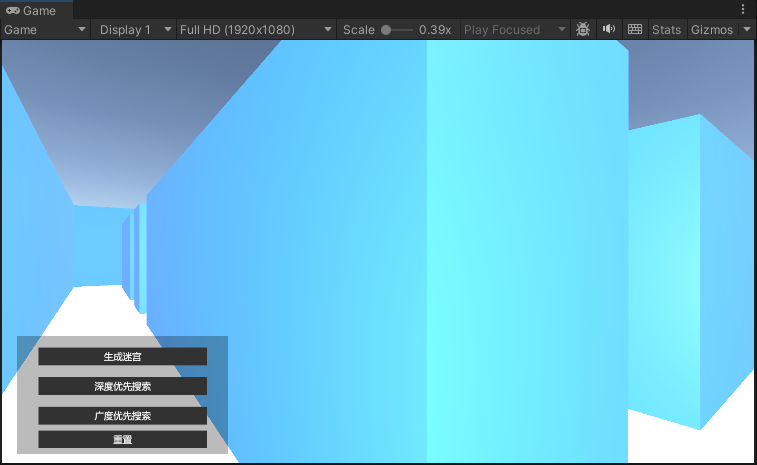
Ps:部分功能在前面的测试中已经通过，不再展示。



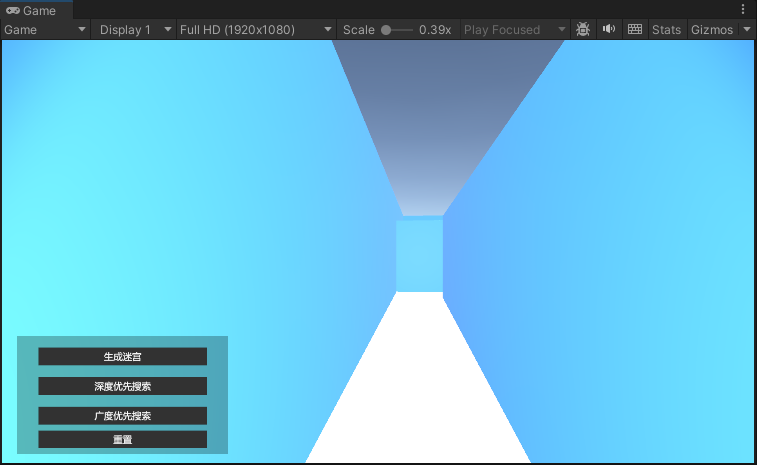
**图4-7 重置功能**



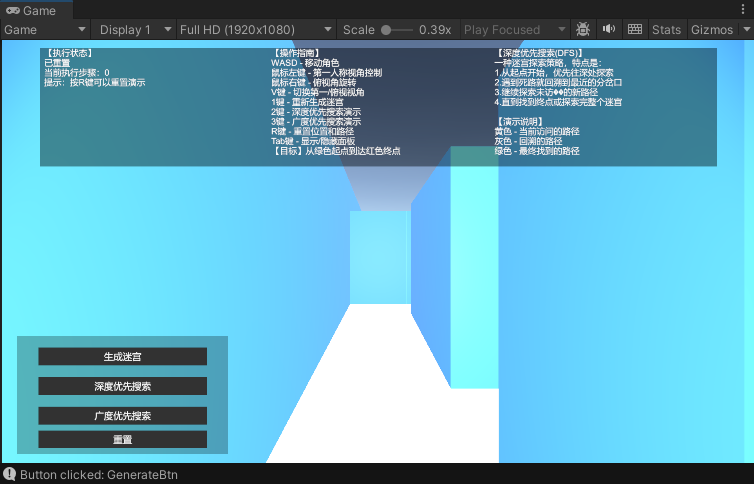
**图4-8 关闭灯光**



**图4-9 打开灯光**



**图4-10 隐藏信息界面**



**图4-11 呼出鼠标**

# 截图的时候无法显示呼出的鼠标。实际上是呼出鼠标了。UI按钮的检测也可以通过输出日志看到该功能已实现。5总结

# 开发过程难点解决方案

# 迷宫生成和重建

# 问题：迷宫重新生成时出现内存泄漏和性能问题

# 解决方案：

# - 实现对象池管理预制体

# - 优化资源加载和卸载时机

# - 分帧处理迷宫生成过程

# 2. 算法可视化

# 问题：需要展示算法执行过程且保持流畅

# 解决方案：

# - 使用协程控制执行节奏

# - 实现分步执行机制

# - 通过颜色系统反馈状态

# 3. 视角切换系统

# 问题：切换视角时相机位置和旋转不平滑

# 解决方案：

# - 实现相机平滑插值

# - 基于玩家朝向计算俯视角

# - 优化相机跟随算法

# 4. UI交互优化

# 问题：UI信息展示与游戏交互存在冲突

# 解决方案：

# - 实现UI显示切换动画

# - 优化UI层级管理

# - 添加渐变过渡效果

# **可扩展方向**

# 1.完善迷宫游戏化元素

2.增加教程系统等

# 3.扩展其他算法以及数据结构

# 4.数据持久化

# 参考文献

论文撰写过程中参考过的期刊、会议论文、毕业论文、书籍、网页等，均需在参考文献中列出，参考文献标号需要在正文中相应位置以上标形式标注。参考文献具体格式要求可参考知网（<https://www.cnki.net/>）的参考文献导出格式：

致谢

独立完成实验及撰写实验报告，不得抄袭。若发现抄袭，抄袭及被抄袭者皆为0分，除非有理由或证据证明非抄袭，或在不知情情况下被抄袭。允许互相学习交流，并在实验报告末尾致谢部分注明和谁讨论过。

感谢与chatgpt的交流。

# 附录：核心代码

/// <summary>

/// 生成新的迷宫

/// 包括清理旧迷宫和创建新迷宫的过程

/// </summary>

public void GenerateMaze()

{

MonoManager.GetInstance().StopAllCoroutines();

if (mazeContainer != null)

{

List<GameObject> wallsToRecycle = new List<GameObject>();

foreach (Transform child in mazeContainer.transform)

{

if (child != null && child.gameObject.name.Contains("Wall"))

{

child.gameObject.SetActive(false);

wallsToRecycle.Add(child.gameObject);

}

}

foreach (var wall in wallsToRecycle)

{

if (wall != null)

{

ResourcesManager.GetInstance().Recycle("Prefabs/Wall", wall);

}

}

if (startPointEffect != null)

{

startPointEffect.Stop();

GameObject.Destroy(startPointEffect.gameObject);

startPointEffect = null;

}

if (endPointEffect != null)

{

endPointEffect.Stop();

GameObject.Destroy(endPointEffect.gameObject);

endPointEffect = null;

}

PoolManager.GetInstance().ClearPool("Prefabs/Wall");

}

MonoManager.GetInstance().StartCoroutine(DelayedMazeGeneration());

}

private IEnumerator DelayedMazeGeneration()

{

yield return null;

if (mazeContainer == null)

{

mazeContainer = new GameObject("MazeContainer");

}

maze = new MazeCell[mazeWidth, mazeHeight];

for (int x = 0; x < mazeWidth; x++)

{

for (int y = 0; y < mazeHeight; y++)

{

maze[x, y] = new MazeCell(x, y);

}

}

mazeGenerator.GenerateMaze(maze);

if (isFirstGeneration)

{

yield return CreateMazeVisuals();

isFirstGeneration = false;

}

else

{

yield return CreateWalls();

}

CreatePointEffects();

float centerX = -mazeWidth \* cellSize / 2f;

float centerZ = -mazeHeight \* cellSize / 2f;

Vector3 startPos = new Vector3(centerX + cellSize, 1.5f, centerZ + cellSize);

PlayerManager.GetInstance().SetPlayerPosition(startPos, true);

MusicManager.GetInstance().PlaySFX("maze\_generate", false);

}