《程序设计实践》实验报告

王浩然2024302121006 侯璎钊2024302121002

石若瀚2024302121332 王柏灿2024302121306

一、实验目的

（1）理解和使用树和图的概念与算法

（2）熟练C++/C#编程/python +pyqt5 (pyside6)

（3）路径规划算法：尝试自主查询资料，选择适 当的算法、确定实现方案

（4）设计实现机器人运动路径规划软件系统

（5）学习软件详细设计的写法

二、实验内容及要求

2.1 实验内容

采用全局路径规划方法实现机器人路径规划，具体内容包括：

（1）环境模拟：设计适当的数据结构来描述机器人工作环境（二维）。

（2）机器人模拟：设计适当的数据结构来描述机器人的状态信息（位置，运动方向）和运动控制（四个方向上的移动、转向等）

（3）最优路径计算：根据给定的起点和终点，设计最短路径搜索算法，根据找到的最短路径，生成机器人运动控制序列。

（4）扩展功能：通过网口接收指令和环境信息，据此实现环境模拟和路径计算。

2.2 实验要求

（1）程序具有良好的用户界面，信息显示详细、清楚；

（2）机器人工作环境（二维）大小可设置，工作环境中的障碍物可设置，并在界面中显示；

（3）起点和终点可设置，并在界面中显示；

（4）搜索到的最短路径可在界面中显示；

（5）机器人沿最短路径的运动过程可在界面中显示。

三、实验原理

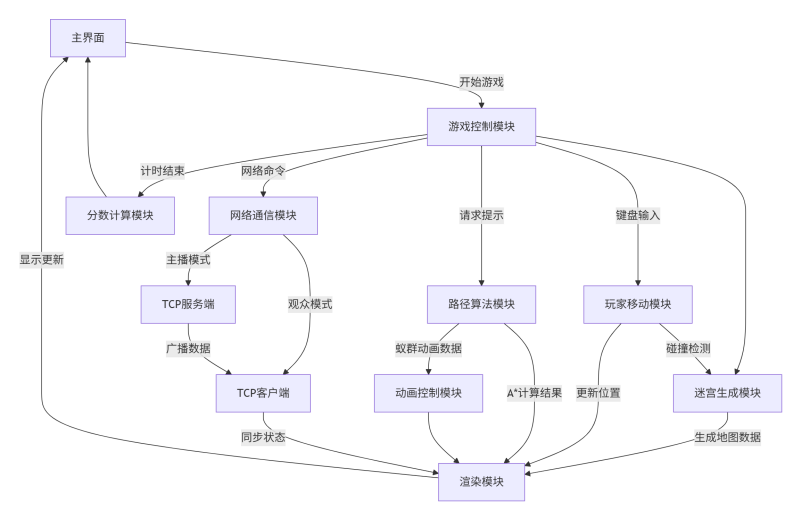


图3.1

如图3.1为本次设计项目的总流程图，在实验目的要求的框架下进行拓展，进而形成了一个以机器人自动巡路功能为核心的，在Qt框架为基础上的，可以借助网口通信实现在线分享和观看其他用户的游戏进度的迷宫游戏系统。其中起到关键作用的模块包括随机迷宫生成模块、路径算法模块、网络通信模块和UI界面模块

3.1 迷宫生成模块

随机Prim算法是一种基于图论的最小生成树算法变体。该算法通过系统化的随机选择机制，在保证迷宫连通性的前提下，构建具有单一解特性的迷宫结构。

算法初始化阶段构建双分辨率网格矩阵，其中奇数坐标位代表潜在通路节点，偶数坐标位表征可穿透边界。该设计将二维迷宫拓扑映射为图结构中的顶点与边，为后续处理提供数学基础。起始节点的随机选定遵循均匀分布原则，确保生成结果的不可预测性。初始节点激活后，其邻接边界被标记为待处理集合，形成算法的初始工作队列。

核心迭代过程体现贪心算法特性：每次从候选边集合中随机抽取元素进行处理。当选定边界连接已访问与未访问区域时，执行边界破除操作，将对应网格值更新为通路标识，并将新访问区域的邻接边界纳入候选集合。值得注意的是，本实现引入概率性环路生成机制，以20%的伯努利试验概率允许连接两个已访问区域，此设计在保证主干路径唯一性的同时，增加迷宫的拓扑复杂度。进而实现了有多个通路的随机迷宫生成，满足了自动巡路模块所需要的环境要求。

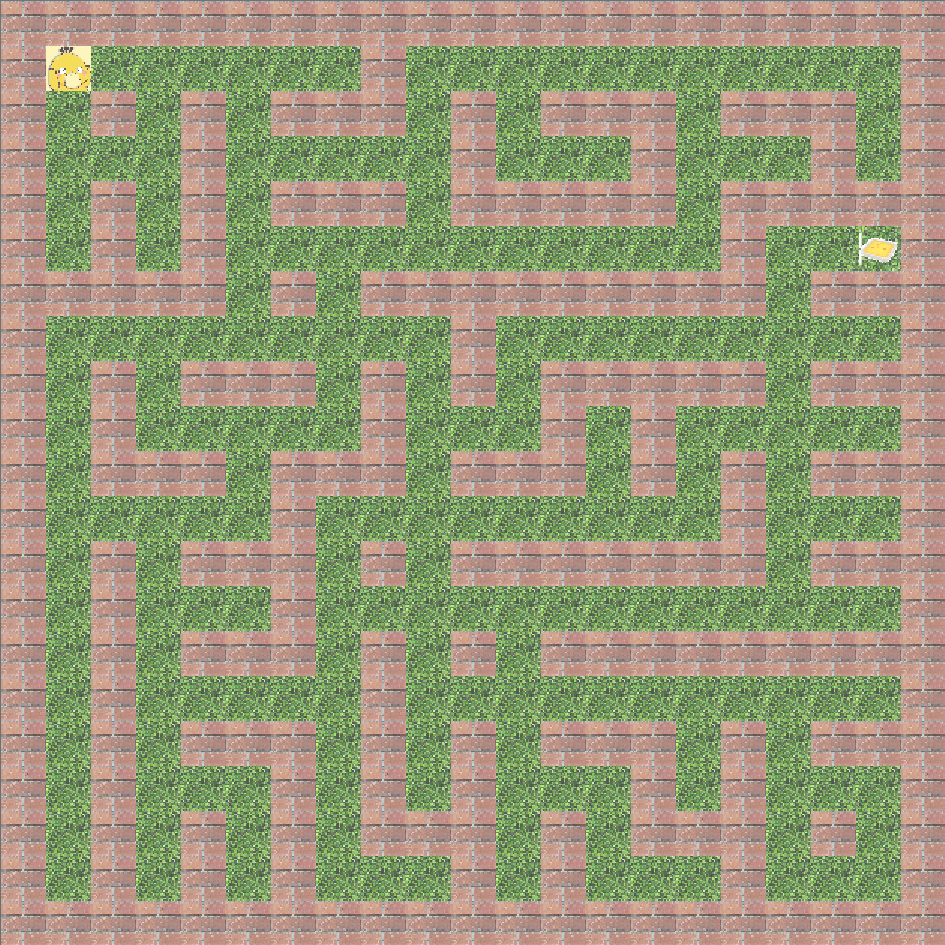


图3.1.1随机生成迷宫实例

3.2 路径算法模块

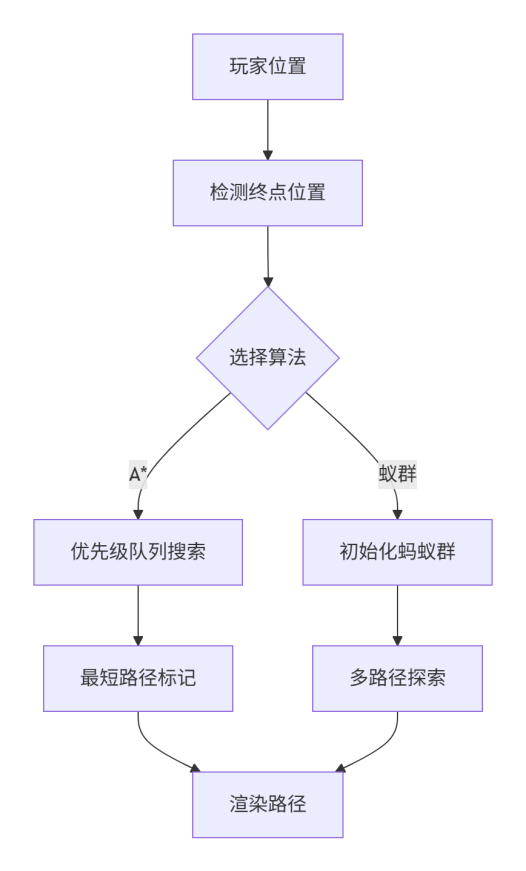


图3.2.1 路径算法模块工作图

如图显示了小组设计项目的路径算法模块，其中包括了A\*算法和蚁群算法两种模式，不仅满足了当前阶段实验目的实验的要求，同时还满足了日后对于更多算法功能拓展的要求，满足游戏性的同时提升整个项目的趣味性。

（1）A\*算法



图3.2.2 A\*算法巡路结果展示

A\*算法作为一种启发式搜索算法，在路径规划领域具有重要的理论意义和广泛的实际应用价值。该算法的核心在于构建了一个综合评估函数f(n)=g(n)+h(n)，其中g(n)表示从起始节点到当前节点的实际路径代价，h(n)表示当前节点到目标节点的启发式估计代价。算法的有效性依赖于启发函数h(n)的设计，当h(n)满足可采纳性（admissible）条件时，即始终不大于真实代价，A\*算法能够保证找到最优路径。

在迷宫求解的具体实现中，算法维护两个关键数据结构：开放集合（open set）和封闭集合（closed set）。开放集合存储待考察的节点，按照f(n)值进行优先级排序；封闭集合记录已处理的节点，避免重复计算。每次迭代从开放集合中提取f(n)值最小的节点进行扩展，计算其相邻节点的g值和h值。若相邻节点未在开放集合中，则加入集合；若已在集合中但发现更优路径，则更新其代价值。这一过程持续进行，直至找到目标节点或开放集合为空。

（2）蚁群算法



图3.2.3 蚁群算法动画展示

蚁群算法是一种模拟自然界蚂蚁觅食行为的群体智能优化算法，其核心思想是通过正反馈机制和分布式计算来寻找最优路径。算法模拟了真实蚂蚁群体在寻找食物过程中释放信息素（pheromone）的行为特征，将优化问题转化为信息素积累与挥发的动态平衡过程。

在迷宫路径搜索的具体应用中，算法初始化多个虚拟蚂蚁个体，每只蚂蚁根据当前网格的信息素浓度和启发式信息（如距离目标的倒数）进行概率性移动。移动过程中，蚂蚁在路径上留下虚拟信息素，其浓度与路径质量成反比。算法通过迭代更新全局信息素矩阵，使得优质路径上的信息素浓度逐渐增强，而劣质路径上的信息素则随时间挥发减弱。这种正反馈机制最终引导蚂蚁群体收敛到最优或次优解。

算法的计算复杂度主要取决于蚂蚁数量、迭代次数和问题规模，其并行搜索特性使其特别适合解决离散空间中的组合优化问题。与传统确定性算法相比，蚁群算法具有更强的鲁棒性和全局搜索能力，但也存在收敛速度较慢的固有缺陷。

（3）算法比较

A\*算法对于单一路径获取最优解具有很好的适用性，而蚁群算法则有更强的全局遍历能力，面对具有多个目标或者协同实现的任务具有更好的适应能力，本次巡路模块同时提供了两种算法

3.3 网络通信模块

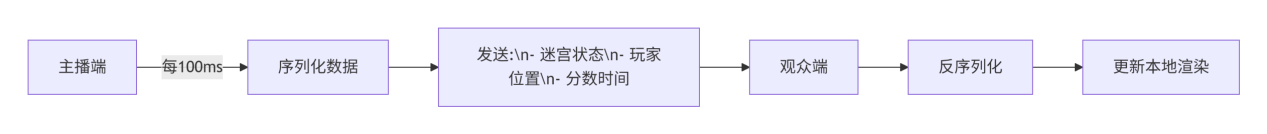


图3.3.1 网络通信模块流程图

实现客户端和服务器两个角色：

（1）服务器端：负责广播游戏状态给多个客户端

（2）客户端：接收并显示服务器发送的游戏状态

网络同步机制在本系统中承担着游戏状态实时共享的关键功能，其设计基于经典的客户端-服务器架构，采用TCP协议作为底层传输保障。该同步系统的核心在于维持分布式节点间的状态一致性，通过序列化技术将游戏状态对象转化为字节流进行网络传输，实现了跨进程的实时数据同步。

系统初始化阶段，主播端实例化QTcpServer对象并绑定至预设端口（如12345），进入监听状态。当观众端通过QTcpSocket发起连接请求时，服务器接受连接并建立全双工通信信道。这一过程严格遵循TCP三次握手协议，确保连接的可靠性。连接建立后，系统进入周期性状态同步阶段，主播端每100毫秒采集当前游戏状态快照，包括迷宫矩阵数据、玩家坐标、得分及剩余时间等关键参数。

状态数据的序列化采用紧凑的二进制格式，通过QDataStream实现结构化数据的线性化。每个数据包首部包含4字节的魔数（Magic Number）0xDEADBEEF用于标识协议，随后是2字节的版本号和4字节的数据长度字段，最后是经过QByteArray压缩的游戏状态数据。这种设计既保证了协议的扩展性，又优化了网络带宽利用率。数据包通过网络层传输至客户端后，接收方首先验证魔数和校验和，确认数据完整性后执行反序列化操作，将字节流重构为内存中的游戏状态对象。

观众端的渲染模块通过观察者模式监听网络数据到达事件，当收到新的状态更新时，立即触发界面重绘流程。为确保平滑的视觉体验，系统采用双缓冲机制：前台缓冲区显示当前帧，后台缓冲区根据网络数据准备下一帧，通过垂直同步技术避免画面撕裂。当检测到网络延迟超过阈值（默认300ms）时，客户端自动启用预测算法，基于历史状态数据插值估算当前可能的状态，待收到实际数据后再进行校正。

异常处理机制是同步系统的重要组成。当检测到网络中断时，客户端尝试指数退避重连策略，初始重试间隔为1秒，最大不超过30秒。在断连期间，系统维持最后已知的有效状态，并在界面显示连接状态提示。主播端则维护心跳检测机制，每5秒验证连接活性，对无响应的客户端执行连接回收，释放系统资源。

该同步系统的创新之处在于其自适应压缩算法，根据当前网络带宽动态调整数据精度。对于迷宫矩阵数据，采用游程编码（RLE）压缩连续重复值；对于坐标信息，使用相对位移量而非绝对位置来减少数据量。测试表明，在典型的10×10迷宫规模下，单个状态包可压缩至200字节左右，即使在带宽受限的移动网络环境下也能保持流畅的同步体验。

系统安全性通过传输层加密保障，采用TLS 1.2协议建立安全通道，防止中间人攻击。身份认证使用基于SHA-256的挑战-响应机制，确保只有合法客户端可以接入。性能优化方面，通过套接字选项调整TCP\_NODELAY禁用Nagle算法，减少小数据包的延迟，同时设置适当的发送/接收缓冲区大小（默认64KB）以适应不同的网络环境。

3.4 Ui界面模块

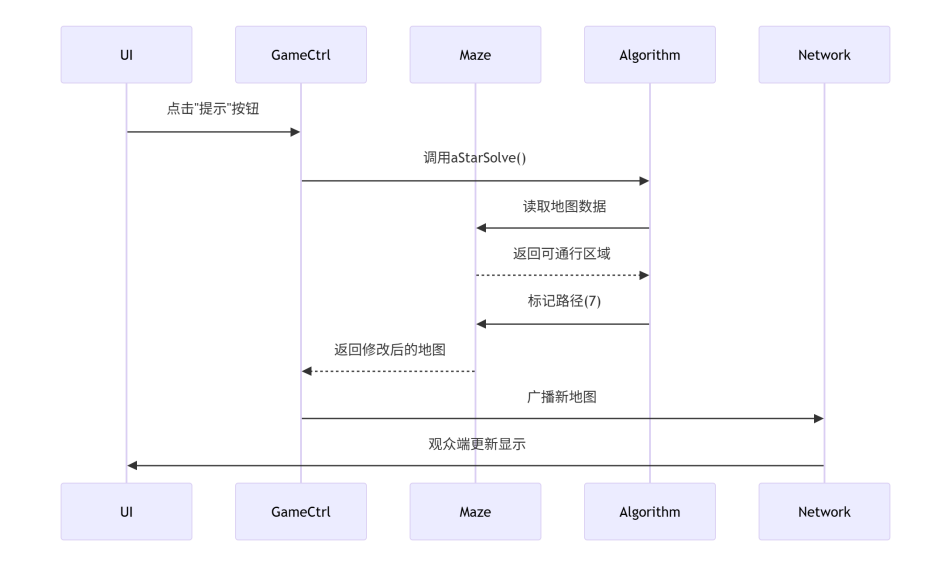


图3.4.1 模块交互流程图

UI界面模块作为系统的人机交互核心，采用MVC（Model-View-Controller）架构实现数据表示与用户操作的解耦。该模块基于Qt框架的图形视图系统构建，通过多层次的渲染管线和事件处理机制，实现了游戏状态的视觉呈现与用户输入的精确响应。

在数据层方面，界面模块与游戏逻辑模块通过观察者模式建立动态绑定关系。当迷宫模型状态发生变化时，通过Qt的信号-槽机制触发界面重绘事件。界面元素的状态管理采用有限状态机模型，定义"菜单"、"游戏"、"暂停"和"结束"四种核心状态，各状态间的转换严格遵循预设的业务规则。渲染管线采用分层绘制策略，从下至上依次处理背景层（渐变填充）、迷宫层（单元格绘制）、实体层（玩家/蚂蚁）和UI控件层（按钮/文字），最后通过合成器输出最终帧。

视觉呈现的核心在于迷宫网格的实时渲染。系统首先计算适应窗体尺寸的单元大小（perblock），确保迷宫始终居中显示。每个网格单元根据其数值状态映射到对应的视觉元素：墙壁使用重复贴图技术，通路采用程序化生成的纹理，起点和终点则加载预制的资源图片。对于动态元素如玩家角色，实现基于关键帧的动画插值，在位置变化时产生平滑的移动效果。光照效果通过叠加半透明层实现，使用高斯模糊算法创建控件面板的毛玻璃特效。

用户交互系统构建在Qt事件模型之上，通过事件过滤器机制实现差异化的输入处理。键盘输入采用状态检测而非事件驱动的方式，每帧检测WASD和IJKL键的状态组合，确保移动响应的实时性。鼠标交互实现精确的坐标转换，将窗体坐标映射到迷宫网格坐标系，支持障碍物编辑模式下的像素级精确操作。触控设备适配通过手势识别模块实现，将触摸事件转化为标准的鼠标/键盘事件。

界面模块的性能优化体现在多个方面：纹理资源使用LRU缓存策略管理，对频繁访问的贴图保留GPU内存驻留；静态元素采用批量绘制技术，通过顶点缓冲对象减少绘制调用；动态元素使用脏矩形算法，仅重绘发生变化的区域。对于高刷新率需求，实现基于垂直同步的自适应帧率控制，在60Hz显示器上保持稳定的16.7ms帧间隔。

可访问性设计是界面模块的重要特性。系统实现屏幕阅读器支持，通过Qt Accessibility框架暴露控件语义信息；色彩方案遵循WCAG 2.0标准，提供足够的对比度；控件尺寸适配系统DPI设置，确保在高分屏上的清晰显示。国际化支持基于Qt Linguist工具链，实现文本资源的动态加载和布局镜像处理。

视觉反馈系统通过多种方式增强用户体验：按钮状态变化采用CSS过渡效果，焦点控件显示发光边框，重要事件触发粒子特效。音效系统与界面操作深度集成，每个交互元素都有对应的声音反馈，通过混音器实现三维音频定位。性能监控组件实时显示帧率和内存占用，帮助开发者优化资源使用。



图3.4.2 彩蛋实例展示

四、实验步骤与过程

（1）项目设计和分工阶段（7月1日）

根据项目设计要求和目的给出了设计迷宫游戏的基本设计方案，同时，根据拓展设计要求，确定了实验拓展的基本方案，即添加了利用网口通讯实现的直播和观战功能。随后实现了基本分工：

王浩然：巡路模块，网口通讯模块实现和实验报告的撰写；

侯璎钊：动画模块和UI界面模块的实现和优化；

石若瀚：项目测试以及优化；

王柏灿：项目规划以及实验报告的形成；

1. 迷宫基本功能的实现阶段（7月2日）

王浩然与侯璎钊同学在已有算法的基础上基本实现了迷宫游戏所需功能的基本算法要求，并在初步的石若瀚同学的测试当中满足了当前阶段的实验要求

1. 直播功能和UI界面的实现阶段（7月3日）

王浩然与侯璎钊同学继续更新算法，基本实现了直播功能和UI界面的优化，王柏灿同学形成了基本报告框架。

1. 添加蚁群算法并完善实验报告（7月4日）

王浩然同学继续优化寻路算法，添加蚁群算法的实现以及动画并进一步完善整体实验报告

五、实验心得

在本次迷宫游戏系统的开发实践中，项目组成员深入理解了游戏引擎架构、路径规划算法和网络同步机制等核心计算机科学概念。通过将理论知识转化为实际可运行的系统，获得了宝贵的工程实践经验。

系统架构方面，采用分层设计模式实现了各模块的高内聚低耦合。MVC架构的严格遵循使得游戏逻辑、界面呈现和用户输入三者清晰分离，这不仅提升了代码的可维护性，也为后续功能扩展奠定了良好基础。

算法实现环节，随机Prim算法、A算法和蚁群算法的对比研究最具启发性。通过实际编码发现，随机Prim算法生成的迷宫具有更好的拓扑随机性，而A算法在路径规划效率上展现出明显优势。蚁群算法的并行搜索特性虽然增加了实现复杂度，但其解决组合优化问题的能力令人印象深刻。在性能调优过程中，算法的时间复杂度从理论分析到实际测量的差异，让项目组成员对算法工程化有了更深认识。

网络同步模块的开发最具挑战性。TCP协议的选择保证了传输可靠性，但实时性优化需要精心设计。通过实现状态压缩、预测算法和断线重连机制，不仅理解了网络游戏开发的核心技术要点，更培养了系统级的异常处理思维。特别是在高延迟环境下的状态插值算法实现，让我对分布式系统的一致性模型有了切身体会。

在UI开发过程中，深刻体会到用户体验设计的重要性。从简单的功能实现到流畅的动画效果，从基本的交互到完整的可访问性支持，每个改进都带来明显的体验提升。Qt强大的图形系统虽然降低了开发难度，但要实现专业级的视觉效果仍需关注细节处理，如纹理过滤、抗锯齿等图形学知识的具体应用。

通过本次实验，项目组成员系统性了解了游戏开发的技术流程，培养了严谨的工程思维。最大的收获在于认识到理论知识与实践能力的辩证关系：优秀的算法需要配合精心的工程实现才能发挥最大效用，而系统设计中的每个决策都需要权衡性能、可维护性和用户体验等多重因素。这些经验将对项目组成员未来的技术学习和项目开发产生深远影响。同时项目组成员还进一步强化了通过多种渠道获取信息的能力，进而解决了在实验过程中所遇到的技术问题。

此外，对于未来代码的优化，我们来有志于进一步提升迷宫游戏相关网口通信链接的健壮性，同时利用在线服务器提供一个在线发排行榜满足游戏基本要求，

**代码附录**

|  |
| --- |
| **CMakeLists.txt** |
| cmake\_minimum\_required(VERSION 3.5)  project(maze VERSION 0.1 LANGUAGES CXX)  set(CMAKE\_AUTOUIC ON)  set(CMAKE\_AUTOMOC ON)  set(CMAKE\_AUTORCC ON)  set(CMAKE\_CXX\_STANDARD 17)  set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)  find\_package(Qt6 REQUIRED COMPONENTS Widgets Network) # Qt6  find\_package(QT NAMES Qt6 REQUIRED COMPONENTS Widgets)  find\_package(Qt**${QT\_VERSION\_MAJOR}** REQUIRED COMPONENTS Widgets)  find\_package(Qt**${QT\_VERSION\_MAJOR}** REQUIRED COMPONENTS Widgets Network) # 添加 Network 模块  find\_package(Qt6 COMPONENTS Core Widgets Multimedia REQUIRED) # 添加Multimedia模块  set(PROJECT\_SOURCES  maze.h  mazewidget.h  main.cpp  maze.cpp  mazewidget.cpp  mazewidget.ui  )  # 链接Qt库  if(**${QT\_VERSION\_MAJOR}** GREATER\_EQUAL 6)  qt\_add\_executable(maze  MANUAL\_FINALIZATION  **${PROJECT\_SOURCES}**  maze.h maze.cpp  mazewidget.ui  )  # Define target properties for Android with Qt 6 as:  # set\_property(TARGET maze APPEND PROPERTY QT\_ANDROID\_PACKAGE\_SOURCE\_DIR  # ${CMAKE\_CURRENT\_SOURCE\_DIR}/android)  # For more information, see https://doc.qt.io/qt-6/qt-add-executable.html#target-creation  else()  if(ANDROID)  add\_library(maze SHARED  **${PROJECT\_SOURCES}**  )  # Define properties for Android with Qt 5 after find\_package() calls as:  # set(ANDROID\_PACKAGE\_SOURCE\_DIR "${CMAKE\_CURRENT\_SOURCE\_DIR}/android")  else()  add\_executable(maze  **${PROJECT\_SOURCES}**  )  endif()  endif()  target\_link\_libraries(maze PRIVATE Qt**${QT\_VERSION\_MAJOR}**::Widgets)  target\_link\_libraries(maze PRIVATE Qt6::Widgets Qt6::Network) # Qt6  target\_link\_libraries(maze PRIVATE Qt6::Widgets Qt6::Multimedia )  # Qt for iOS sets MACOSX\_BUNDLE\_GUI\_IDENTIFIER automatically since Qt 6.1.  # If you are developing for iOS or macOS you should consider setting an  # explicit, fixed bundle identifier manually though.  if(**${QT\_VERSION}** VERSION\_LESS 6.1.0)  set(BUNDLE\_ID\_OPTION MACOSX\_BUNDLE\_GUI\_IDENTIFIER com.example.maze)  endif()  set\_target\_properties(maze PROPERTIES  **${BUNDLE\_ID\_OPTION}**  MACOSX\_BUNDLE\_BUNDLE\_VERSION **${PROJECT\_VERSION}**  MACOSX\_BUNDLE\_SHORT\_VERSION\_STRING **${PROJECT\_VERSION\_MAJOR}**.**${PROJECT\_VERSION\_MINOR}**  MACOSX\_BUNDLE TRUE  WIN32\_EXECUTABLE TRUE  )  include(GNUInstallDirs)  install(TARGETS maze  BUNDLE DESTINATION .  LIBRARY DESTINATION **${CMAKE\_INSTALL\_LIBDIR}**  RUNTIME DESTINATION **${CMAKE\_INSTALL\_BINDIR}**  )  if(QT\_VERSION\_MAJOR EQUAL 6)  qt\_finalize\_executable(maze)  endif() |

|  |
| --- |
| **maze.h** |
| #ifndef MAZE\_H  #define MAZE\_H  #include <iostream>  #include<vector>  #include<queue>  #include<cmath>  #include<QTimer>  #include <QElapsedTimer>  using namespace std;  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* -1:边界 0:墙壁  \* 1:空单元(初始化后未判定的单元)  \* 2:待定墙壁(新建已定单元时建立)  \* 3:已定单元(确定迷宫树已延伸到的单元)  \* 4:打通墙壁(延伸操作)  \* 5.起点  \* 6.终点  \* 7.已经过路径  \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  struct Pos {  int i;  int j;  bool operator==(const Pos& other) const {  return i == other.i && j == other.j;  }  // 建议同时添加!=运算符  bool operator!=(const Pos& other) const {  return !(\*this == other);  }  };  struct AStarNode {  Pos pos;  int f; // f = g + h  int g; // 从起点到当前节点的实际代价  int h; // 启发式估计值  // 优先队列比较函数  bool operator<(const AStarNode& other) const {  return f > other.f; // 小顶堆，f值小的优先  }  };  class maze {  private:  int level; //迷宫阶数  int\*\* map; //地图存储空间  int start\_x, start\_y; //起终点  int end\_x, end\_y;  //生成基础地图（单元格）  void base();  //使点的周围设为待定（2）  void \_2(int i, int j);  //设定迷宫开始延伸的起点  void start();  //循环停止判定（是否存在未判定的区域）  bool judge();  //操作（如果相邻空单元（1）则打通（变为4），如果不相邻空单元则为墙壁（0））  void op(int i, int j);  //随机选择一个待定墙壁判断并操作  void random2();  int heuristic(const Pos& a, const Pos& b); // 启发式函数  std::vector<Pos> getNeighbors(const Pos& current); // 获取相邻节点  void reconstructPath(const std::vector<std::vector<Pos>>& cameFrom, Pos current); // 重建路径  QTimer\* solveTimer; // 解决动画计时器  std::vector<Pos> solutionPath;// 存储解决路径  std::vector<Pos> animatedPath; // 存储已动画展示的路径  int animationIndex; // 当前动画位置  int currentStep; // 当前动画步数  bool forHint;  std::vector<Pos> walkedPath; // 存储已经走过的路径  public:  //构造函数申请内存空间  maze(int in\_level);  ~maze();  //获取地图  int getlevel();  int\*\* getmap();  int getside();  //生成地图  void makemap();  int p\_x, p\_y; //当前位置  Pos x[1000];  int p[100][100];  bool able(int k, int t);  void brush(int k);  void search(int k);  void aStarSolve(bool forHint); // 替换原有的solve函数  //重置地图  void rebuildmap();  int\* operator[](int index);  void startSolveAnimation(); // 开始解决动画  void nextSolveStep(); // 动画下一步  void stopSolveAnimation(); // 停止动画  };  struct Ant {  Pos position;  std::vector<Pos> path; // 已走过的路径  bool active = true;  };  // 蚁群算法类  class AntColony {  public:  explicit AntColony(maze\* mazeMap); // 添加 explicit 避免隐式转换  void update();  ~AntColony();  void start(Pos startPos);  bool foundPath() const { return m\_foundPath; }  int getElapsedTime() const { return m\_elapsedTime; }  const std::vector<Ant>& getAnts() const { return m\_ants; }  void clearAllAnts(); // 新增清除所有蚂蚁的函数  const std::vector<std::vector<bool>>& getVisitedMatrix() const;  private:  maze\* m\_map;  std::vector<Ant> m\_ants;  QElapsedTimer m\_timer;  bool m\_foundPath = false;  int m\_elapsedTime = 0;  std::vector<std::vector<bool>> m\_visited; // 记录所有蚂蚁走过的路径  std::vector<Pos> getPossibleMoves(const Ant& ant);  void moveAnt(Ant& ant, const Pos& moveTo);  };  #endif // MAZE\_H |

|  |
| --- |
| **maze.cpp** |
| #include "maze.h"  #include<queue>  #include <QMessageBox>  #include<QTimer>  //生成基础地图（单元格）  void maze::base() {  for (int i = 0; i < level \* 2 + 1; i++) {  for (int j = 0; j < level \* 2 + 1; j++) {  if (i == 0 || j == 0 || i == level \* 2 + 1 - 1 ||  j == level \* 2 + 1 - 1) {  map[i][j] = -1;  }  else if (i % 2 != 0 && j % 2 != 0) {  map[i][j] = 1;  }  else {  map[i][j] = 0;  }  }  }  }  //使点的周围设为待定（2）  void maze::\_2(int i, int j) {  if (map[i - 1][j] == 0) {  map[i - 1][j] = 2;  }  if (map[i + 1][j] == 0) {  map[i + 1][j] = 2;  }  if (map[i][j - 1] == 0) {  map[i][j - 1] = 2;  }  if (map[i][j + 1] == 0) {  map[i][j + 1] = 2;  }  }  //设定迷宫开始延伸的起点  void maze::start() {  map[start\_x][start\_y] = 5;  \_2(start\_x, start\_y);  }  //循环停止判定（是否存在未判定的区域）  bool maze::judge() {  bool flag = 0;  for (int i = 0; i < level \* 2 + 1; i++) {  for (int j = 0; j < level \* 2 + 1; j++) {  if (map[i][j] == 2) {  flag = 1;  return flag;  }  }  }  return flag;  }  //操作（如果相邻空单元（1）则打通（变为4），如果不相邻空单元则为墙壁（0））  void maze::op(int i, int j) {  // 原条件：向下打通（上侧已访问，下侧未访问）  if ((map[i - 1][j] == 3 || map[i - 1][j] == 5) && map[i + 1][j] == 1) {  map[i][j] = 4;  map[i + 1][j] = 3;  \_2(i + 1, j);  start\_x = i + 1;  start\_y = j;  }  // 原条件：向右打通（左侧已访问，右侧未访问）  else if ((map[i][j - 1] == 3 || map[i][j - 1] == 5) && map[i][j + 1] == 1) {  map[i][j] = 4;  map[i][j + 1] = 3;  \_2(i, j + 1);  start\_x = i;  start\_y = j + 1;  }  // 原条件：向上打通（下侧已访问，上侧未访问）  else if ((map[i + 1][j] == 3 || map[i + 1][j] == 5) && map[i - 1][j] == 1) {  map[i][j] = 4;  map[i - 1][j] = 3;  \_2(i - 1, j);  start\_x = i - 1;  start\_y = j;  }  // 原条件：向左打通（右侧已访问，左侧未访问）  else if ((map[i][j + 1] == 3 || map[i][j + 1] == 5) && map[i][j - 1] == 1) {  map[i][j] = 4;  map[i][j - 1] = 3;  \_2(i, j - 1);  start\_x = i;  start\_y = j - 1;  }  // 新增条件：允许垂直方向打通两个已访问区域（形成环路）  else if ((map[i - 1][j] == 3 || map[i - 1][j] == 5) &&  (map[i + 1][j] == 3 || map[i + 1][j] == 5) &&  (rand() % 100 < 20)) { // 20%概率打通  map[i][j] = 4;  }  // 新增条件：允许水平方向打通两个已访问区域（形成环路）  else if ((map[i][j - 1] == 3 || map[i][j - 1] == 5) &&  (map[i][j + 1] == 3 || map[i][j + 1] == 5) &&  (rand() % 100 < 20)) { // 20%概率打通  map[i][j] = 4;  }  // 默认情况：保持为墙  else {  map[i][j] = 0;  }  }  //随机选择一个待定墙壁判断并操作  void maze::random2() {  int t = 0;  for (int i = 0; i < level \* 2 + 1; i++) {  for (int j = 0; j < level \* 2 + 1; j++) {  if (map[i][j] == 2) {  t++;  }  }  }  int k = rand() % t + 1;  t = 0;  for (int i = 0; i < level \* 2 + 1; i++) {  for (int j = 0; j < level \* 2 + 1; j++) {  if (map[i][j] == 2) {  t++;  if (t == k) {  op(i, j);  goto loopout;  }  }  }  }  loopout:  if (!judge()) {  map[start\_x][start\_y] = 6;  }  }  //构造函数申请内存空间  maze::maze(int in\_level) : level(in\_level) {  map = new int\* [level \* 2 + 1];  for (int i = 0; i < level \* 2 + 1; i++) {  map[i] = new int[level \* 2 + 1];  }  start\_x = 1, start\_y = 1; //起点设置为(1,1)  }  maze::~maze() {  stopSolveAnimation(); // 确保动画停止  for (int i = 0; i < level \* 2 + 1; i++) {  delete[] map[i];  }  delete[] map;  }  //获取地图  int maze::getlevel() {  return maze::level;  }  int\*\* maze::getmap() {  return map;  }  int maze::getside() {  return level \* 2 + 1;  }  //生成地图  void maze::makemap() {  p\_x = start\_x;  p\_y = start\_y;  base();  start();  int a = 0;  while (judge()) {  a++;  random2();  // if (a % 30 == 0) {  // printarr(map, level);  // system("PAUSE");  // }  }  }  int\* maze::operator[](int index) {  return map[index];  }  //重置地图  void maze::rebuildmap() {  start\_x = 1;  start\_y = 1;  makemap();  }  bool maze::able(int k, int t) {  Pos last = x[k - 1];  Pos cur = last;  if (t == 1 && p[cur.i][cur.j + 1] != 0)return 0;  if (t == 2 && p[cur.i + 1][cur.j] != 0)return 0;  if (t == 3 && p[cur.i][cur.j - 1] != 0)return 0;  if (t == 4 && p[cur.i - 1][cur.j] != 0)return 0;  return 1;  }  void maze::brush(int k) {  for (int m = 0; m < k; m++) {  map[x[m].i][x[m].j] = 7;  }  map[x[k - 1].i][x[k - 1].j] = 6;  }  void maze::search(int k) {  if (x[k - 1].i == end\_x && x[k - 1].j == end\_y) {  brush(k);  }  else {  for (int t = 1; t <= 4; t++) {  if (able(k, t)) {  Pos cur = x[k - 1];  if (t == 1)cur.j += 1;  if (t == 2)cur.i += 1;  if (t == 3)cur.j -= 1;  if (t == 4)cur.i -= 1;  x[k] = cur;  p[cur.i][cur.j] = 2;  search(k + 1);  p[cur.i][cur.j] = 0;  }  }  }  }  int maze::heuristic(const Pos& a, const Pos& b) {  return abs(a.i - b.i) + abs(a.j - b.j);  }  // 获取可通行的相邻节点  std::vector<Pos> maze::getNeighbors(const Pos& current) {  std::vector<Pos> neighbors;  int dirs[4][2] = { {0,1}, {1,0}, {0,-1}, {-1,0} }; // 右、下、左、上  for (auto& dir : dirs) {  int ni = current.i + dir[0];  int nj = current.j + dir[1];  // 检查边界和可通行性  if (ni >= 0 && ni < level \* 2 + 1 && nj >= 0 && nj < level \* 2 + 1 &&  (map[ni][nj] == 3 || map[ni][nj] == 4 || map[ni][nj] == 5 || map[ni][nj] == 6 || map[ni][nj] == 7)) {  neighbors.push\_back({ ni, nj });  }  }  return neighbors;  }  // 重建路径并标记  void maze::reconstructPath(const std::vector<std::vector<Pos>>& cameFrom, Pos current) {  // 先清除之前的路径标记(7)  for (int i = 0; i < level \* 2 + 1; i++) {  for (int j = 0; j < level \* 2 + 1; j++) {  if (map[i][j] == 7) {  map[i][j] = 3; // 恢复为已定单元  }  }  }  // 标记新路径  while (!(current.i == p\_x && current.j == p\_y)) {  if (map[current.i][current.j] != 5 && map[current.i][current.j] != 6) {  map[current.i][current.j] = 7;  }  current = cameFrom[current.i][current.j];  // 防止无限循环  if (current.i == -1 || current.j == -1) break;  }  }  // A\*算法主函数  void maze::aStarSolve(bool forHint) {  // 首先确保终点位置正确设置  bool foundEnd = false;  for (int i = 0; i < level \* 2 + 1 && !foundEnd; i++) {  for (int j = 0; j < level \* 2 + 1 && !foundEnd; j++) {  if (map[i][j] == 6) {  end\_x = i;  end\_y = j;  foundEnd = true;  }  }  }  if (!foundEnd) {  QMessageBox::information(nullptr, "错误", "未找到终点位置");  return;  }  // 初始化优先队列  std::priority\_queue<AStarNode> openSet;  // 记录节点来源  std::vector<std::vector<Pos>> cameFrom(level \* 2 + 1, std::vector<Pos>(level \* 2 + 1, { -1,-1 }));  // 记录g值  std::vector<std::vector<int>> gScore(level \* 2 + 1, std::vector<int>(level \* 2 + 1, INT\_MAX));  // 起点初始化  Pos start = { p\_x, p\_y };  Pos end = { end\_x, end\_y };  gScore[p\_x][p\_y] = 0;  int h = heuristic(start, end);  openSet.push({ start, h, 0, h });  while (!openSet.empty()) {  AStarNode current = openSet.top();  openSet.pop();  // 到达终点后的处理  if (current.pos == end) {  // 存储完整路径  solutionPath.clear();  Pos node = current.pos;  while (node != start) {  solutionPath.push\_back(node);  node = cameFrom[node.i][node.j];  // 安全保护，防止意外无限循环  if (solutionPath.size() > level \* level \* 4) {  qDebug() << "Path reconstruction too long, aborting";  solutionPath.clear();  return;  }  }  solutionPath.push\_back(start);  std::reverse(solutionPath.begin(), solutionPath.end());  // 只有提示模式立即标记路径  if (forHint) {  for (const auto& pos : solutionPath) {  if (map[pos.i][pos.j] != 5 && map[pos.i][pos.j] != 6) {  map[pos.i][pos.j] = 7;  }  }  }  return;  }  // 遍历邻居  for (Pos& neighbor : getNeighbors(current.pos)) {  // 计算临时g值  int tentativeG = gScore[current.pos.i][current.pos.j] + 1;  // 找到更优路径  if (tentativeG < gScore[neighbor.i][neighbor.j]) {  cameFrom[neighbor.i][neighbor.j] = current.pos;  gScore[neighbor.i][neighbor.j] = tentativeG;  int f = tentativeG + heuristic(neighbor, end);  openSet.push({ neighbor, f, tentativeG, f - tentativeG });  }  }  }  // 如果没有找到路径  QMessageBox::information(nullptr, "提示", "未找到可行路径");  }  void maze::startSolveAnimation() {  // 清除之前的状态  walkedPath.clear();  // 如果没有路径，先计算路径  if (solutionPath.empty()) {  aStarSolve(true);  }  currentStep = 0;  solveTimer = new QTimer();  QObject::connect(solveTimer, &QTimer::timeout, [this]() {  this->nextSolveStep();  });  solveTimer->start(100); // 每100毫秒移动一步  }  void maze::nextSolveStep() {  if (currentStep < solutionPath.size()) {  // 移动当前位置  p\_x = solutionPath[currentStep].i;  p\_y = solutionPath[currentStep].j;  // 记录走过的路径（不包括起点和终点）  if (currentStep > 0 && currentStep < solutionPath.size() - 1) {  walkedPath.push\_back(solutionPath[currentStep]);  map[p\_x][p\_y] = 7; // 走过的路径变黄  }  currentStep++;  }  else {  stopSolveAnimation();  }  }  void maze::stopSolveAnimation() {  /\*if (solveTimer) {  solveTimer->stop();  delete solveTimer;  solveTimer = nullptr;  }\*/  solutionPath.clear();  }  AntColony::AntColony(maze\* mazeMap)  : m\_map(mazeMap), m\_foundPath(false), m\_elapsedTime(0) {  if (m\_map) {  int side = m\_map->getside();  m\_visited.resize(side, std::vector<bool>(side, false));  }  }  AntColony::~AntColony() {  m\_map = nullptr; // 防止悬空指针  }  void AntColony::start(Pos startPos) {  clearAllAnts(); // 这会重置访问矩阵  m\_timer.start();  Ant initialAnt;  initialAnt.position = startPos;  initialAnt.path.push\_back(startPos);  m\_ants.push\_back(initialAnt);  // 标记起点为已访问  m\_visited[startPos.i][startPos.j] = true;  }  void AntColony::update() {  if (!m\_map) return;  if (m\_foundPath || m\_ants.empty()) return;  std::vector<Ant> newAnts;  for (auto& ant : m\_ants) {  if (!ant.active) continue;  // 获取当前可走方向  std::vector<Pos> possibleMoves;  Pos current = ant.position;  // 检查四个方向  int dirs[4][2] = { {0,1}, {1,0}, {0,-1}, {-1,0} };  for (auto& dir : dirs) {  Pos next = { current.i + dir[0], current.j + dir[1] };  // 检查是否可通行且未走过  if (next.i >= 0 && next.i < m\_map->getside() &&  next.j >= 0 && next.j < m\_map->getside()) {  int cellValue = (\*m\_map)[next.i][next.j];  if ((cellValue == 3 || cellValue == 4 || cellValue == 6||cellValue==7) &&  std::find(ant.path.begin(), ant.path.end(), next) == ant.path.end()) {  possibleMoves.push\_back(next);  }  }  }  // 处理移动  if (possibleMoves.empty()) {  ant.active = false; // 死路，蚂蚁消失  }  else if (possibleMoves.size() == 1) {  // 单一方向移动  ant.position = possibleMoves[0];  ant.path.push\_back(possibleMoves[0]);  // 检查是否到达终点  if ((\*m\_map)[ant.position.i][ant.position.j] == 6) {  m\_foundPath = true;  m\_elapsedTime = m\_timer.elapsed();  return;  }  }  else {  // 分叉路：克隆蚂蚁  for (size\_t i = 1; i < possibleMoves.size(); ++i) {  Ant newAnt = ant;  newAnt.position = possibleMoves[i];  newAnt.path.push\_back(possibleMoves[i]);  newAnts.push\_back(newAnt);  // 检查是否到达终点  if ((\*m\_map)[newAnt.position.i][newAnt.position.j] == 6) {  m\_foundPath = true;  m\_elapsedTime = m\_timer.elapsed();  return;  }  }  // 原蚂蚁走第一个方向  ant.position = possibleMoves[0];  ant.path.push\_back(possibleMoves[0]);  // 检查是否到达终点  if ((\*m\_map)[ant.position.i][ant.position.j] == 6) {  m\_foundPath = true;  m\_elapsedTime = m\_timer.elapsed();  return;  }  }  }  // 添加新克隆的蚂蚁  m\_ants.insert(m\_ants.end(), newAnts.begin(), newAnts.end());  // 移除不活跃的蚂蚁  m\_ants.erase(std::remove\_if(m\_ants.begin(), m\_ants.end(),  [](const Ant& a) { return !a.active; }), m\_ants.end());  }  void AntColony::clearAllAnts() {  m\_ants.clear();  m\_foundPath = false;  m\_elapsedTime = 0;  // 重置访问矩阵  if (m\_map) {  int side = m\_map->getside();  m\_visited.assign(side, std::vector<bool>(side, false));  }  }  std::vector<Pos> AntColony::getPossibleMoves(const Ant& ant) {  std::vector<Pos> possibleMoves;  int dirs[4][2] = { {0,1}, {1,0}, {0,-1}, {-1,0} };  for (auto& dir : dirs) {  Pos next = { ant.position.i + dir[0], ant.position.j + dir[1] };  if (next.i >= 0 && next.i < m\_map->getside() &&  next.j >= 0 && next.j < m\_map->getside()) {  int cellValue = (\*m\_map)[next.i][next.j];  // 检查是否是可行走区域且未被任何蚂蚁过  if ((cellValue == 3 || cellValue == 4 || cellValue == 6 || cellValue == 7) &&  !m\_visited[next.i][next.j] && // 新增检查  std::find(ant.path.begin(), ant.path.end(), next) == ant.path.end()) {  possibleMoves.push\_back(next);  }  }  }  return possibleMoves;  }  void AntColony::moveAnt(Ant& ant, const Pos& moveTo) {  // 标记新位置为已访问  m\_visited[moveTo.i][moveTo.j] = true;  ant.position = moveTo;  ant.path.push\_back(moveTo);  if ((\*m\_map)[ant.position.i][ant.position.j] == 6) {  m\_foundPath = true;  m\_elapsedTime = m\_timer.elapsed();  }  }  const std::vector<std::vector<bool>>& AntColony::getVisitedMatrix() const {  return m\_visited;  } |

|  |
| --- |
| **mazewidget.h** |
| #ifndef MAZEWIDGET\_H  #define MAZEWIDGET\_H  #include <QWidget>  #include <QMessageBox> //Qt信息窗口头文件  #include <QPainter> //Qt绘图头文件  #include <QDebug> //QtDebug头文件  #include <QKeyEvent> //Qt按键事件头文件  #include <QTimer> //Qt计时器头文件  #include <QInputDialog> //Qt输入对话框头文件  #include<QTcpSocket> // 客户端Socket  #include <QTcpServer> // 服务端  #include <QNetworkInterface>  #include <QHostAddress>  #include <QMediaPlayer> // 多媒体播放器  #include <QAudioOutput> // 音频输出  #include "maze.h" //迷宫类头文件  QT\_BEGIN\_NAMESPACE  namespace Ui {  class mazeWidget;  }  QT\_END\_NAMESPACE  class mazeWidget : public QWidget {  Q\_OBJECT  public:  mazeWidget(QWidget\* parent = nullptr);  ~mazeWidget();  protected:  void paintEvent(QPaintEvent\*); //绘图事件  void keyPressEvent(QKeyEvent\*); //按键按下事件  void mousePressEvent(QMouseEvent\* event) override;  private slots:  void on\_start\_btn\_clicked(); //|  void on\_stop\_ptn\_clicked(); //|  void on\_end\_btn\_clicked(); //|  void on\_rule\_btn\_clicked(); //|各按钮点击槽函数  void on\_setting\_btn\_clicked(); //|  void time\_update(); //时间更新槽函数  void on\_solve\_btn\_clicked();  void on\_stupid\_clicked();  void on\_disease\_clicked(); // 设障按钮点击槽函数  void on\_hostMode\_clicked(); // 主播模式按钮  void on\_audienceMode\_clicked(); // 观众模式按钮  void on\_startWatch\_clicked(); // 开始观战按钮  void newConnection(); // 主播端有新连接  void readyRead(); // 有数据可读  void on\_antColony\_clicked();  private:  Ui::mazeWidget\* ui; //ui对象  maze\* map;//迷宫对象  bool obstacleMode; // 标记是否处于设障模式  bool painting\_switch; //绘图开关  bool timing\_switch; //计时开关  bool keybord\_switch; //键盘响应开关  bool stop\_switch; //暂停按钮状态  QPixmap menuImage; // 菜单图片  bool showMenu; // 是否显示菜单  int grade; //分数  int time; //时间  QTimer\* timer;  QTcpServer\* server; // 主播端使用的服务器  QTcpSocket\* clientSocket; // 观众端使用的客户端socket  QTcpSocket\* serverSocket; // 主播端使用的客户端socket(用于发送数据)  void startBroadcasting(); // 开始直播  void stopBroadcasting(); // 停止直播  void startWatching(); // 开始观看  void stopWatching(); // 停止观看  void sendGameState(); // 发送游戏状态  void readGameState(); // 读取游戏状态  QMediaPlayer\* bgmPlayer; // BGM播放器  QAudioOutput\* audioOutput; // 音频输出  AntColony\* antColony;  QTimer\* antAnimationTimer;  void startAntAnimation();  void stopAntAnimation();  void updateAntAnimation();  void updateAntsRegion();  int m\_aStarTime;  void clearAntColony(); // 新增清除蚁群的函数  };  #endif // MAZEWIDGET\_H |

|  |
| --- |
| **mazewidget.cpp** |
| #include "mazewidget.h"  #include "ui\_mazewidget.h"  #include<QRect>  #include<QPainter>  #include <QElapsedTimer>  const double stupid\_num = 0.2;  bool if\_stupid = false;  mazeWidget::mazeWidget(QWidget\* parent)  : QWidget(parent)  , ui(new Ui::mazeWidget), map(new maze(20)) // 将 maze 实例传递给 solve 的构造函数  , painting\_switch(false), timing\_switch(false)  , keybord\_switch(false), stop\_switch(false), grade(0), time(0), obstacleMode(false), showMenu(true),antColony(nullptr),antAnimationTimer(nullptr) {  //TODO:状态栏  ui->setupUi(this);  ui->progressBar->setVisible(false); //初始隐藏进度条  ui->end\_btn->setEnabled(false); //设置终止按钮禁用  ui->stop\_ptn->setEnabled(false); //设置暂停按钮禁用  ui->grade\_value->setText(" "); //设置分数值显示为空  ui->time\_value->setText(" "); //设置时间值显示为空  map->makemap(); //生成地图  timer = new QTimer(this); //初始化计时器  connect(timer, &QTimer::timeout, this, &mazeWidget::time\_update); //链接时间更新信号与槽  connect(ui->disease, &QPushButton::clicked, this, &mazeWidget::on\_disease\_clicked);  ui->plaque\_time->setText(" ");  ui->plaque\_grade->setText(" ");  ui->label->setVisible(true);  ui->solve\_btn->setEnabled(false);  // 主窗口背景（改为暖灰到浅沙色渐变）  this->setStyleSheet(  "background: qlineargradient(x1:0 y1:0, x2:1 y2:1, stop:0 #f5f7fa, stop:1 #c3cfe2);"  "color: #333;"  );  server = nullptr;  clientSocket = nullptr;  serverSocket = nullptr;  audioOutput = new QAudioOutput(this);  bgmPlayer = new QMediaPlayer(this);  // 设置音频输出  bgmPlayer->setAudioOutput(audioOutput);  // 设置音量（0.0-1.0）  audioOutput->setVolume(0.5); // 50%音量  // 加载BGM文件（确保文件路径正确）  bgmPlayer->setSource(QUrl::fromLocalFile("bgm/duck.wav"));  // 设置循环播放  bgmPlayer->setLoops(QMediaPlayer::Infinite);  // 连接错误信号  connect(bgmPlayer, &QMediaPlayer::errorOccurred, [](QMediaPlayer::Error error, const QString& errorString) {  qDebug() << "BGM Error:" << errorString;  });  // 连接按钮信号槽  connect(ui->hostMode, &QPushButton::clicked, this, &mazeWidget::on\_hostMode\_clicked);  connect(ui->audienceMode, &QPushButton::clicked, this, &mazeWidget::on\_audienceMode\_clicked);  connect(ui->startWatch, &QPushButton::clicked, this, &mazeWidget::on\_startWatch\_clicked);  // 游戏区域面板（毛玻璃效果）  ui->frame->setStyleSheet(  "background-color: rgba(255, 255, 255, 0.3);"  "border-radius: 15px;"  "border: 1px solid rgba(0, 0, 0, 0.1);"  "backdrop-filter: blur(5px);" // 毛玻璃效果（需Qt 5.12+）  );  menuImage.load("painting/start.png"); // 替换为你的菜单图片路径  if (menuImage.isNull()) {  qDebug() << "Failed to load menu image";  // 创建一个默认的菜单背景  menuImage = QPixmap(400, 300);  menuImage.fill(Qt::white);  QPainter painter(&menuImage);  painter.setPen(Qt::black);  painter.setFont(QFont("Arial", 20));  painter.drawText(menuImage.rect(), Qt::AlignCenter, "迷宫游戏\n点击开始");  }  ui->solve\_btn->setStyleSheet(  "QPushButton {"  " background: #e67e22;" // 温暖的橙色  " color: white;"  " border: none;"  " padding: 8px 16px;"  " border-radius: 8px;"  " font-weight: 500;"  " min-width: 80px;"  "}"  "QPushButton:hover { background: #d35400; }"  "QPushButton:pressed { background: #ba4a00; }"  );  // 其他按钮（自然中性色）  QString naturalButtonStyle =  "QPushButton {"  " background: rgba(255, 255, 255, 0.7);"  " color: #5d4037;" // 深咖啡色文字  " border: 1px solid rgba(0, 0, 0, 0.1);"  " padding: 6px 12px;"  " border-radius: 6px;"  "}"  "QPushButton:hover {"  " background: rgba(255, 255, 255, 0.9);"  " border-color: rgba(0, 0, 0, 0.2);"  "}";  ui->start\_btn->setStyleSheet(naturalButtonStyle);  ui->end\_btn->setStyleSheet(naturalButtonStyle);  }  mazeWidget::~mazeWidget() {  delete ui;  delete map;  delete timer;  delete bgmPlayer;  delete audioOutput;  stopAntAnimation(); // 先停止动画  delete antColony; // 再删除 AntColony  delete map; // 最后删除 maze  }  void mazeWidget::paintEvent(QPaintEvent\*) {  QPainter painter(this); //画笔对象  painter.setRenderHint(QPainter::Antialiasing); // 这行让图形更平滑  if (showMenu) {  QRect frameRect = ui->frame->geometry();  painter.drawPixmap(frameRect, menuImage);  return;  }  // 否则绘制迷宫  if (!painting\_switch) return;  //绘图逻辑：  QPixmap playerImage("painting/duck.png");  if (playerImage.isNull()) {  qDebug() << "Failed to load player image";  playerImage = QPixmap(20, 20); // 创建空图片作为后备  playerImage.fill(Qt::red);  }  QPixmap wallImage("painting/wall.png"); // 假设墙的贴图放在资源文件的images目录下  if (wallImage.isNull()) {  qDebug() << "Failed to load wall image";  wallImage = QPixmap(20, 20); // 创建空图片作为后备  wallImage.fill(Qt::gray);  }  QPixmap footImage("painting/Foot.png"); // 假设墙的贴图放在资源文件的images目录下  if (footImage.isNull()) {  qDebug() << "Failed to load foot image";  footImage = QPixmap(20, 20); // 创建空图片作为后备  footImage.fill(QColor(255, 183, 77));  }  QPixmap foodImage("painting/Food.png"); // 假设墙的贴图放在资源文件的images目录下  if (foodImage.isNull()) {  qDebug() << "Failed to load foot image";  foodImage = QPixmap(20, 20); // 创建空图片作为后备  foodImage.fill(QColor(183, 28, 28));  }  QPixmap roadImage("painting/grass.png"); // 假设墙的贴图放在资源文件的images目录下  if (roadImage.isNull()) {  qDebug() << "Failed to load foot image";  roadImage = QPixmap(20, 20); // 创建空图片作为后备  roadImage.fill(QColor(250, 250, 245));  }  QPixmap homeImage("painting/home.png"); // 假设墙的贴图放在资源文件的images目录下  if (homeImage.isNull()) {  qDebug() << "Failed to load foot image";  homeImage = QPixmap(20, 20); // 创建空图片作为后备  homeImage.fill(QColor(46, 125, 50));  }  int perblock = (std::min(ui->frame->width(), ui->frame->height()) - 20) / (map->getside());  int start\_x = ui->frame->x() + (ui->frame->width() - (ui->frame->x() + (map->getside()) \* perblock)) / 2;  int strat\_y = ui->frame->y() + (ui->frame->height() - (ui->frame->y() + (map->getside()) \* perblock)) / 2;  for (int i = 0; i < map->getlevel() \* 2 + 1; i++) {  for (int j = 0; j < map->getlevel() \* 2 + 1; j++) {  QRect rect(start\_x + i \* perblock,  strat\_y + j \* perblock,  perblock, perblock);  // 玩家位置（保持醒目的珊瑚色）  if (i == map->p\_x && j == map->p\_y) {  // 缩放图片以适应格子大小  QPixmap scaled = playerImage.scaled(perblock, perblock, Qt::KeepAspectRatio);  painter.drawPixmap(rect, scaled);  }  // 解决方案路径（琥珀色）  else if (map->getmap()[i][j] == 7) {  QPixmap scaledFoot = footImage.scaled(perblock, perblock, Qt::KeepAspectRatioByExpanding);  painter.drawPixmap(rect, scaledFoot);  }  // 普通路径（乳白色）  else if (map->getmap()[i][j] == 3 || map->getmap()[i][j] == 4) {  QPixmap scaledroad = roadImage.scaled(perblock, perblock, Qt::KeepAspectRatio);  painter.drawPixmap(rect, scaledroad);  }  // 起点（深绿色）  else if (map->getmap()[i][j] == 5) {  QPixmap scaledhome = homeImage.scaled(perblock, perblock, Qt::KeepAspectRatio);  painter.drawPixmap(rect, scaledhome);  }  // 终点（深红色）  else if (map->getmap()[i][j] == 6) {  QPixmap scaledfood = foodImage.scaled(perblock, perblock, Qt::KeepAspectRatio);  painter.drawPixmap(rect, scaledfood);  }  // 墙壁（浅灰色）  else {  // 缩放墙的贴图以适应格子大小  QPixmap scaledWall = wallImage.scaled(perblock, perblock, Qt::KeepAspectRatioByExpanding);  painter.drawPixmap(rect, scaledWall);  }  }  }  if (antColony) {  QPixmap antImage("painting/ant.png");  QPixmap visitedImage("painting/antvisited.png"); // 新增：被访问过的路径  for (const Ant& ant : antColony->getAnts()) {  if (ant.active) {  QRect rect(start\_x + ant.position.i \* perblock,  strat\_y + ant.position.j \* perblock,  perblock, perblock);  painter.drawPixmap(rect, antImage.scaled(perblock, perblock));  }  }  // 绘制被访问过的路径  if (antColony) {  for (int i = 0; i < antColony->getVisitedMatrix().size(); i++) {  for (int j = 0; j < antColony->getVisitedMatrix()[i].size(); j++) {  if (antColony->getVisitedMatrix()[i][j]) {  QRect rect(start\_x + i \* perblock,  strat\_y + j \* perblock,  perblock, perblock);  painter.drawPixmap(rect, visitedImage.scaled(perblock, perblock));  }  }  }  }  }  }  void mazeWidget::keyPressEvent(QKeyEvent\* event) {  if (!keybord\_switch) return;  int x = map->p\_x;  int y = map->p\_y;  //键盘移动逻辑：  if (event->key() == Qt::Key\_I || event->key() == Qt::Key\_W) {  if ((\*map)[x][y - 1] == 3 || (\*map)[x][y - 1] == 4 || (\*map)[x][y - 1] == 5 || (\*map)[x][y - 1] == 6 || (\*map)[x][y - 1] == 7) {  map->p\_y--;  }  }  else if (event->key() == Qt::Key\_K || event->key() == Qt::Key\_S) {  if ((\*map)[x][y + 1] == 3 || (\*map)[x][y + 1] == 4 || (\*map)[x][y + 1] == 5 || (\*map)[x][y + 1] == 6 || (\*map)[x][y + 1] == 7) {  map->p\_y++;  }  }  else if (event->key() == Qt::Key\_J || event->key() == Qt::Key\_A) {  if ((\*map)[x - 1][y] == 3 || (\*map)[x - 1][y] == 4 || (\*map)[x - 1][y] == 5 || (\*map)[x - 1][y] == 6 || (\*map)[x - 1][y] == 7) {  map->p\_x--;  }  }  else if (event->key() == Qt::Key\_L || event->key() == Qt::Key\_D) {  if ((\*map)[x + 1][y] == 3 || (\*map)[x + 1][y] == 4 || (\*map)[x + 1][y] == 5 || (\*map)[x + 1][y] == 6 || (\*map)[x + 1][y] == 7) {  map->p\_x++;  }  }  //经过路径  if ((\*map)[map->p\_x][map->p\_y] != 5 && (\*map)[map->p\_x][map->p\_y] != 6)(\*map)[map->p\_x][map->p\_y] = 7;  repaint();  //到达终点  if ((\*map)[map->p\_x][map->p\_y] == 6) {  map->makemap();  repaint();  if (if\_stupid == false)  {  grade += pow(map->getlevel(), 2);  }  else if (if\_stupid == true)  {  grade += stupid\_num\*pow(map->getlevel(), 2);  if\_stupid = false;  }  ui->grade\_value->setText(QString::number(grade));  }  }  void mazeWidget::time\_update() {  if (time != 0) {  //计时中  time--;  ui->time\_value->setText(QString::number(time));  ui->progressBar->setValue(time / 2);  }  else {  timer->stop(); //停止计时器  ui->progressBar->setVisible(false); //隐藏进度条  keybord\_switch = false; //设置键盘响应、  painting\_switch = false; //绘图响应、  timing\_switch = false; //计时响应为关闭状态  repaint(); //清除画布  ui->start\_btn->setEnabled(true); //|  ui->time\_value->setText(" "); //|  ui->grade\_value->setText(" "); //|  ui->stop\_ptn->setEnabled(false); //|设置各按钮与标签状态  ui->end\_btn->setEnabled(false); //|  ui->setting\_btn->setEnabled(true); //|  QMessageBox scoreBox;  scoreBox.setWindowTitle("游戏结束");  // 2. 设置文字内容  scoreBox.setText(QString("您的得分: %1").arg(grade));  // 3. 加载图片并添加到弹窗  QPixmap resultImage("painting/winning.png"); // 确保图片路径正确  if (!resultImage.isNull()) {  scoreBox.setIconPixmap(resultImage.scaled(200, 200, Qt::KeepAspectRatio));  }  else {  qDebug() << "Failed to load result image";  }  // 4. 添加确定按钮  scoreBox.addButton(QMessageBox::Ok);  // 5. 显示弹窗  scoreBox.exec();  //分数重置  grade = 0;  }  }  void mazeWidget::on\_start\_btn\_clicked() {  showMenu = false; // 隐藏菜单  ui->label->setVisible(false);  ui->solve\_btn->setEnabled(true);  painting\_switch = true;  timing\_switch = true;  keybord\_switch = true;  time = 200;  timer->start(1000);  if (bgmPlayer->playbackState() != QMediaPlayer::PlayingState) {  bgmPlayer->play();  }  ui->progressBar->setVisible(true);  ui->progressBar->setValue(100);  repaint();  ui->time\_value->setText(QString::number(time));  ui->grade\_value->setText(QString::number(grade));  ui->start\_btn->setEnabled(false);  ui->stop\_ptn->setEnabled(true);  ui->end\_btn->setEnabled(true);  ui->setting\_btn->setEnabled(false);  ui->plaque\_time->setText("时间");  ui->plaque\_grade->setText("分数");  }  void mazeWidget::on\_stop\_ptn\_clicked() {  if (stop\_switch) {  timing\_switch = false;  keybord\_switch = false;  timer->stop();  ui->stop\_ptn->setText("继续");  stop\_switch = false;  bgmPlayer->pause();  }  else {  timing\_switch = true;  keybord\_switch = true;  timer->start();  ui->stop\_ptn->setText("暂停");  stop\_switch = true;  bgmPlayer->play();  }  }  void mazeWidget::on\_end\_btn\_clicked() {  timing\_switch = false;  painting\_switch = false;  keybord\_switch = false;  stop\_switch = false;  timer->stop();  time = 0;  grade = 0;  ui->plaque\_time->setText(" ");  ui->plaque\_grade->setText(" ");  ui->progressBar->setVisible(false);  ui->grade\_value->setText(" ");  ui->time\_value->setText(" ");  ui->stop\_ptn->setText("暂停");  ui->stop\_ptn->setEnabled(false);  ui->end\_btn->setEnabled(false);  ui->start\_btn->setEnabled(true);  ui->setting\_btn->setEnabled(true);  map->rebuildmap();  ui->label->setVisible(true);  ui->solve\_btn->setEnabled(false);  bgmPlayer->stop();  showMenu = true; // 游戏结束时重新显示菜单  repaint();  }  void mazeWidget::on\_rule\_btn\_clicked() {  QMessageBox rule(QMessageBox::NoIcon, "规则", "计时200秒，根据迷宫等级与经过关卡记分。\n操作方式：WASD或者IJKL控制方向。\n放弃后不得分，使用提示后得分降为对应分段的20%", QMessageBox::Ok);  rule.exec();  }  void mazeWidget::on\_setting\_btn\_clicked() {  QStringList difficultys;  difficultys << tr("小朋友难度(5阶迷宫)") << tr("简单难度(10阶迷宫)") << tr("普通难度(20阶迷宫)") << tr("困难难度(40阶迷宫)");  QString difficulty = QInputDialog::getItem(this, tr("选择难度"),  tr("请选择一个条目"), difficultys, 0, false);  if (difficulty == tr("小朋友难度(5阶迷宫)")) {  delete map;  map = new maze(5);  map->makemap();  }  else if (difficulty == tr("简单难度(10阶迷宫)")) {  delete map;  map = new maze(10);  map->makemap();  }  else if (difficulty == tr("普通难度(20阶迷宫)")) {  delete map;  map = new maze(20);  map->makemap();  }  else if (difficulty == tr("困难难度(40阶迷宫)")) {  delete map;  map = new maze(40);  map->makemap();  }  }  void mazeWidget::on\_solve\_btn\_clicked() {  ui->solve\_btn->setEnabled(false);  map->aStarSolve(true);  // 先清除之前的路径  for (int i = 0; i < map->getside(); i++) {  for (int j = 0; j < map->getside(); j++) {  if (map->getmap()[i][j] == 7) {  map->getmap()[i][j] = 3; // 恢复为可通行区域  }  }  }  // 开始动画  map->startSolveAnimation();  // 使用QTimer定期重绘  QTimer\* animationTimer = new QTimer(this);  connect(animationTimer, &QTimer::timeout, [this]() {  this->repaint();  });  animationTimer->start(100); // 每100毫秒重绘一次  // 动画结束后恢复状态  QTimer::singleShot(100 \* (map->getside() \* 2), [this, animationTimer]() {  animationTimer->stop();  animationTimer->deleteLater();  map->stopSolveAnimation();  map->makemap();  repaint();  ui->grade\_value->setText(QString::number(grade));  ui->solve\_btn->setEnabled(true);  });  }  void mazeWidget::on\_stupid\_clicked()  { // 清除旧路径  for (int i = 0; i < map->getside(); i++) {  for (int j = 0; j < map->getside(); j++) {  if (map->getmap()[i][j] == 7) {  map->getmap()[i][j] = 3;  }  }  }  map->aStarSolve(true);  repaint();  if\_stupid = true;  }  void mazeWidget::on\_disease\_clicked()  {  obstacleMode = true;  }  // 鼠标点击事件处理  void mazeWidget::mousePressEvent(QMouseEvent\* event)  {  if (!obstacleMode || !painting\_switch) {  QWidget::mousePressEvent(event);  return;  }  // 计算点击位置对应的迷宫坐标  int perblock = (std::min(ui->frame->width(), ui->frame->height()) - 20) / (map->getside());  int start\_x = ui->frame->x() + (ui->frame->width() - (ui->frame->x() + (map->getside()) \* perblock)) / 2;  int start\_y = ui->frame->y() + (ui->frame->height() - (ui->frame->y() + (map->getside()) \* perblock)) / 2;  int mazeX = (event->pos().x() - start\_x) / perblock;  int mazeY = (event->pos().y() - start\_y) / perblock;  // 检查坐标是否有效  if (mazeX >= 0 && mazeX < map->getside() &&  mazeY >= 0 && mazeY < map->getside())  {  // 检查是否是可行走区域（不是起点、终点或墙壁）  if ((\*map)[mazeX][mazeY] == 3 || (\*map)[mazeX][mazeY] == 4 || (\*map)[mazeX][mazeY] == 7)  {  (\*map)[mazeX][mazeY] = 0; // 设置为墙壁  QMessageBox::information(this, "提示", "设置障碍成功！！！");  repaint();  }  }  obstacleMode = false; // 一次设障后退出设障模式  QWidget::mousePressEvent(event);  }  void mazeWidget::on\_hostMode\_clicked() {  if (server == nullptr) {  server = new QTcpServer(this);  connect(server, &QTcpServer::newConnection, this, &mazeWidget::newConnection);  if (server->listen(QHostAddress::Any, 12345)) {  // 获取本地IP地址  QString ipAddress;  QList<QHostAddress> ipAddressesList = QNetworkInterface::allAddresses();  for (const QHostAddress& address : ipAddressesList) {  if (address != QHostAddress::LocalHost && address.toIPv4Address()) {  ipAddress = address.toString();  break;  }  }  if (ipAddress.isEmpty()) {  ipAddress = QHostAddress(QHostAddress::LocalHost).toString();  }  QMessageBox::information(this, "提示",  "已开启主播模式，等待观众连接...\n你的IP地址是: " + ipAddress);  ui->hostIP->setText("IP: " + ipAddress);  startBroadcasting();  }  else {  QMessageBox::critical(this, "错误", "无法启动服务器: " + server->errorString());  delete server;  server = nullptr;  }  }  else {  QMessageBox::information(this, "提示", "已经是主播模式");  }  }  // 观众模式按钮点击  void mazeWidget::on\_audienceMode\_clicked() {  if (clientSocket == nullptr) {  clientSocket = new QTcpSocket(this);  connect(clientSocket, &QTcpSocket::readyRead, this, &mazeWidget::readyRead);  connect(clientSocket, &QTcpSocket::disconnected, this, [this]() {  QMessageBox::information(this, "提示", "与主播断开连接");  stopWatching();  });  QMessageBox::information(this, "提示", "已切换到观众模式，请输入主播IP开始观看");  }  else {  QMessageBox::information(this, "提示", "已经是观众模式");  }  }  // 开始观战按钮点击  void mazeWidget::on\_startWatch\_clicked() {  if (clientSocket == nullptr) {  QMessageBox::critical(this, "错误", "请先切换到观众模式");  return;  }  bool ok;  QString hostIP = QInputDialog::getText(this, "输入主播IP", "请输入主播IP地址:",  QLineEdit::Normal, "127.0.0.1", &ok);  if (ok && !hostIP.isEmpty()) {  clientSocket->connectToHost(hostIP, 12345);  if (clientSocket->waitForConnected(10000)) {  QMessageBox::information(this, "提示", "已连接到主播");  showMenu = false; // 游戏结束时重新显示菜单  startWatching();  }  else {  QMessageBox::critical(this, "错误", "连接失败: " + clientSocket->errorString());  stopWatching();  }  }  }  // 开始直播  void mazeWidget::startBroadcasting() {  if (serverSocket == nullptr && server != nullptr && server->hasPendingConnections()) {  serverSocket = server->nextPendingConnection();  connect(serverSocket, &QTcpSocket::disconnected, this, [this]() {  QMessageBox::information(this, "提示", "观众已断开连接");  stopBroadcasting();  });  // 定时发送游戏状态  QTimer\* broadcastTimer = new QTimer(this);  connect(broadcastTimer, &QTimer::timeout, this, &mazeWidget::sendGameState);  broadcastTimer->start(100); // 每100ms发送一次  }  }  // 停止直播  void mazeWidget::stopBroadcasting() {  if (serverSocket != nullptr) {  serverSocket->disconnectFromHost();  serverSocket->deleteLater();  serverSocket = nullptr;  }  if (server != nullptr) {  server->close();  server->deleteLater();  server = nullptr;  }  ui->hostIP->setText("IP: 未开启");  }  // 开始观看  void mazeWidget::startWatching() {  // 禁用游戏控制按钮  ui->start\_btn->setEnabled(false);  ui->stop\_ptn->setEnabled(false);  ui->end\_btn->setEnabled(false);  ui->solve\_btn->setEnabled(false);  // 启用绘图  painting\_switch = true;  showMenu = false; // 游戏结束时重新显示菜单  repaint();  }  // 停止观看  void mazeWidget::stopWatching() {  if (clientSocket != nullptr) {  clientSocket->disconnectFromHost();  clientSocket->deleteLater();  clientSocket = nullptr;  // 恢复游戏控制按钮  ui->start\_btn->setEnabled(true);  ui->solve\_btn->setEnabled(true);  // 清除绘图  showMenu = true; // 游戏结束时重新显示菜单  painting\_switch = false;  repaint();  }  // 发送游戏状态  void mazeWidget::sendGameState() {  if (serverSocket != nullptr && serverSocket->state() == QAbstractSocket::ConnectedState) {  QByteArray block;  QDataStream out(&block, QIODevice::WriteOnly);  out.setVersion(QDataStream::Qt\_5\_15);  // 发送迷宫数据  out << map->getside();  for (int i = 0; i < map->getside(); i++) {  for (int j = 0; j < map->getside(); j++) {  out << map->getmap()[i][j];  }  }  // 发送玩家位置  out << map->p\_x << map->p\_y;  // 发送分数和时间  out << grade << time;  serverSocket->write(block);  }  }  // 读取游戏状态  void mazeWidget::readGameState() {  if (clientSocket != nullptr && clientSocket->bytesAvailable() > 0) {  QDataStream in(clientSocket);  in.setVersion(QDataStream::Qt\_5\_15);  int side;  in >> side;  // 如果迷宫大小不匹配，重新创建  if (map->getside() != side) {  delete map;  map = new maze((side - 1) / 2);  }  // 读取迷宫数据  for (int i = 0; i < side; i++) {  for (int j = 0; j < side; j++) {  int value;  in >> value;  map->getmap()[i][j] = value;  }  }  // 读取玩家位置  in >> map->p\_x >> map->p\_y;  // 读取分数和时间  in >> grade >> time;  // 更新UI  ui->grade\_value->setText(QString::number(grade));  ui->time\_value->setText(QString::number(time));  // 重绘  repaint();  }  }  // 主播端有新连接  void mazeWidget::newConnection() {  if (serverSocket == nullptr) {  startBroadcasting();  }  else {  // 已经有观众连接，拒绝新的连接  QTcpSocket\* newSocket = server->nextPendingConnection();  newSocket->disconnectFromHost();  newSocket->deleteLater();  QMessageBox::information(this, "提示", "已有观众连接，拒绝新的连接");  }  }  // 有数据可读  void mazeWidget::readyRead() {  readGameState();  }  void mazeWidget::startAntAnimation() {  stopAntAnimation(); // 确保先停止之前的动画  antColony = new AntColony(map);  antColony->start({ map->p\_x, map->p\_y });  antAnimationTimer = new QTimer(this);  connect(antAnimationTimer, &QTimer::timeout, this, &mazeWidget::updateAntAnimation);  antAnimationTimer->start(50); // 降低刷新频率到200ms  }  void mazeWidget::stopAntAnimation() {  if (antAnimationTimer) {  antAnimationTimer->stop();  delete antAnimationTimer;  antAnimationTimer = nullptr;  }  if (antColony) {  delete antColony;  antColony = nullptr;  }  repaint(); // 清除残留的蚂蚁显示  }  void mazeWidget::updateAntAnimation() {  if (!antColony) return;  antColony->update();  updateAntsRegion();  if (antColony->foundPath()) {  stopAntAnimation();  clearAntColony();  antColony = new AntColony(map); // 重新创建  repaint();  // 比较两种算法的时间  int antTime = antColony->getElapsedTime();  QMessageBox::information(this, "算法比较",  QString("蚁群算法耗时: %1 ms\nA\*算法耗时: %2 ms")  .arg(antColony->getElapsedTime()) // int  .arg(m\_aStarTime)); // int  }  }  void mazeWidget::on\_antColony\_clicked() {  if (!map) return; // 防御性检查  stopAntAnimation(); // 停止之前的动画  antColony = new AntColony(map); // 重新创建  startAntAnimation();  // 1. 清除旧路径  for (int i = 0; i < map->getside(); i++) {  for (int j = 0; j < map->getside(); j++) {  if (map->getmap()[i][j] == 7) {  map->getmap()[i][j] = 3;  }  }  }  // 2. 运行A\*并计时  QElapsedTimer aStarTimer;  aStarTimer.start();  for (int i = 0; i < 100; i++) { // 运行100次取平均值  map->aStarSolve(true);  }  m\_aStarTime = aStarTimer.elapsed() / 100; // 计算平均耗时  const int aStarTime = m\_aStarTime;  // 3. 运行蚁群算法  if (antColony) {  delete antColony;  antColony = nullptr;  }  antColony = new AntColony(map);  antColony->start({ map->p\_x, map->p\_y });  // 4. 设置动画定时器  if (antAnimationTimer) {  antAnimationTimer->stop();  delete antAnimationTimer;  }  antAnimationTimer = new QTimer(this);  // 使用lambda捕获aStarTime  connect(antAnimationTimer, &QTimer::timeout, [this, aStarTime]() {  if (!antColony) return;  antColony->update();  repaint();  if (antColony->foundPath()) {  antAnimationTimer->stop();  QMessageBox::information(  this,  "算法比较",  QString("蚁群算法耗时: %1 ms\nA\*算法耗时: %2 ms")  .arg(antColony->getElapsedTime())  .arg(aStarTime) // 这里使用已捕获的aStarTime  );  }  });  antAnimationTimer->start(100);  }  void mazeWidget::updateAntsRegion() {  // 计算需要重绘的区域  int perblock = (std::min(ui->frame->width(), ui->frame->height()) - 20) / (map->getside());  int start\_x = ui->frame->x() + (ui->frame->width() - (ui->frame->x() + (map->getside()) \* perblock)) / 2;  int start\_y = ui->frame->y() + (ui->frame->height() - (ui->frame->y() + (map->getside()) \* perblock)) / 2;  // 只更新蚂蚁所在区域  if (antColony) {  for (const Ant& ant : antColony->getAnts()) {  if (ant.active) {  QRect rect(start\_x + ant.position.i \* perblock,  start\_y + ant.position.j \* perblock,  perblock, perblock);  update(rect);  }  }  }  }  void mazeWidget::clearAntColony() {  if (antColony) {  antColony->clearAllAnts(); // 清除所有蚂蚁  stopAntAnimation(); // 停止动画  repaint(); // 重绘界面  }  } |