BOMB IT 项目说明文档

庄子懿 宋新悦

December 28, 2025

Abstract

本文档详细阐述了基于 Cocos2d-x 3.17.2 引擎开发的休闲竞技游戏 “Q 版泡泡堂” 的设计与实现。文档涵盖了项目背景、技术栈、整体架构设计、场景流转逻辑以及核心类的职责划分。本项目实现了单人闯关、本地双人对战及迷雾模式，并集成了基于 A* 算法与热力图机制的高级 AI 系统。

Contents

1 项目概述

1.1 项目简介

本项目是一款复刻经典游戏“Q 版泡泡堂”玩法的 2D 休闲游戏。玩家通过在网格地图中放置炸弹摧毁障碍物、获取增强道具，并利用战术策略击败对手（AI 或其他玩家）。游戏支持高分辨率适配，拥有完整的游戏循环、胜负判定及数据持久化功能。

1.2 技术栈

- **开发引擎**: Cocos2d-x 3.17.2
- **编程语言**: C++ (C++11/14 Standard)
- **开发环境**: Visual Studio 2022
- **图形渲染**: OpenGL ES (Cocos2d-x 底层封装)
- **音频支持**: Cocos2d Experimental AudioEngine

1.3 核心玩法模式

游戏包含以下几种主要模式：

- **单人模式 (SINGLE)**: 玩家对战 3 个不同难度的 AI 机器人。
- **本地双人 (LOCAL_2P)**: 两名玩家在同一设备上分别使用 WASD 和方向键进行对战，同时混入 2 个 AI。
- **迷雾模式 (FOG)**: 引入“战争迷雾”机制，视野受限，强调探索与反应。
- **在线模式 (ONLINE)**: (预留接口，当前版本暂未实现)。

2 系统架构设计

2.1 整体架构

游戏采用标准的 Cocos2d-x 场景管理机制，通过 `Director` 进行场景堆栈管理。主要流转逻辑如下：

1. **StartScene**: 游戏主入口，负责资源预加载、背景音乐播放及菜单导航（开始、说明、排行榜）。
2. **moduleScene**: 模式选择中枢，分发用户请求至单人、双人或迷雾模式。
3. **SelectScene / SelectScene_2Player**:
   - **SelectScene**: 用于单人或迷雾模式的角色选择，随后跳转至难度选择。
   - **SelectScene_2Player**: 专门处理双人同时输入的选人逻辑，解决输入冲突。
4. **SelectDifficultyScene**: 配置 AI 难度（简单/困难），最终完成参数注入并启动游戏场景。

5. **GameScene**: 核心战斗场景，包含游戏主循环、物理碰撞、AI 调度等所有战斗逻辑。

6. **GameOverLayer**: 战斗结算层，展示胜负结果，支持“再来一局”或“返回主页”。

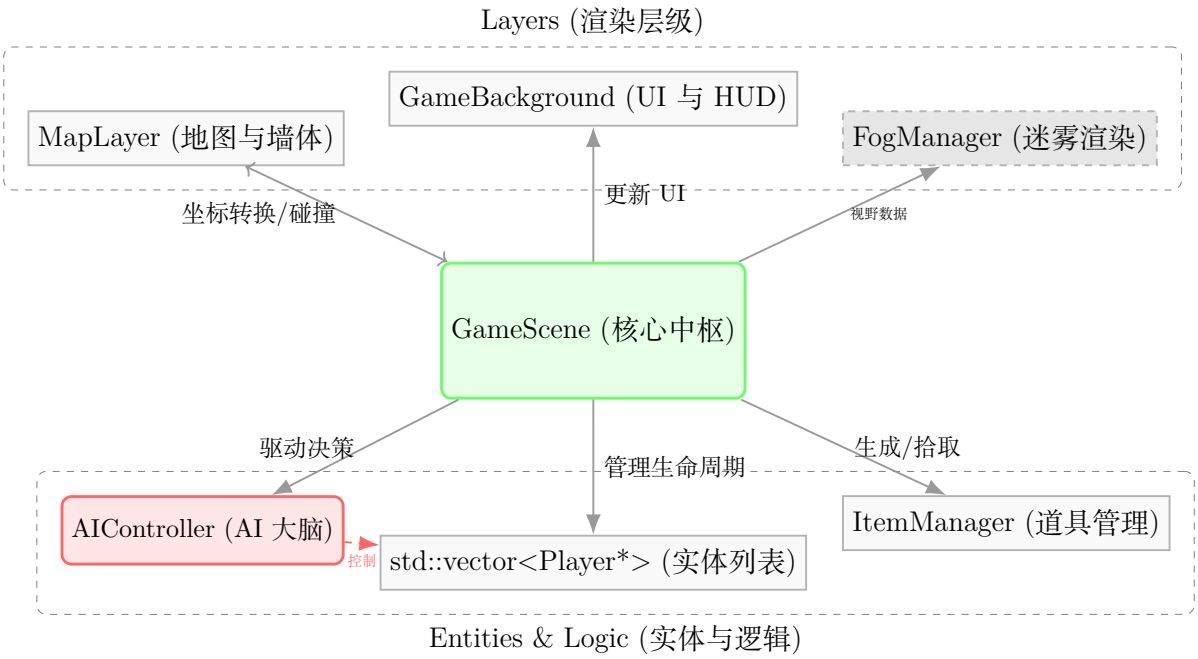

Figure 1: GameScene 核心组件架构图

2.2 核心类架构

游戏采用了典型的面向对象设计，核心类图关系如下：

2.2.1 游戏主控

`GameScene` 是游戏运行时的核心容器，它聚合了所有关键的管理器与层级：

- **Layer 层级管理**:
  - `GameBackground` (Z=0): 负责 UI 侧边栏、倒计时、暂停逻辑及 HUD 显示。
  - `MapLayer` (Z=1): 负责瓦片地图渲染、坐标转换及障碍物管理。
  - `FogManager` (Z=100): 仅在迷雾模式启用，负责视野遮罩渲染。
- **实体管理**:
  - `std::vector<Player*> _players`: 管理所有玩家和 AI 实体。
  - `ItemManager`: 管理地图上的道具生成与拾取逻辑。
- **控制器**:
  - `AIController`: 独立的 AI 决策模块，驱动非玩家实体的行为。

2.2.2 实体类

- **Player**: 继承自 `cocos2d::Sprite`。
  - 职责：处理移动输入、播放动画、放置炸弹、属性管理（速度、威力、血量）及死亡逻辑。
  - 状态：包含 `isDead`, `isMoving`, `isAI`, `invincible` 等标志位。
- **Bomb**: 继承自 `cocos2d::Sprite`。
  - 职责：处理倒计时、爆炸范围计算 (`willExplodeGrid`) 及生成火焰。
  - 交互：注册到 `GameScene` 的 `_bombDangers` 列表中，供 AI 避险参考。
- **Item**: 继承自 `cocos2d::Sprite`。
  - 职责：定义道具类型（加速、威力、护盾等）及拾取后的特效反馈。

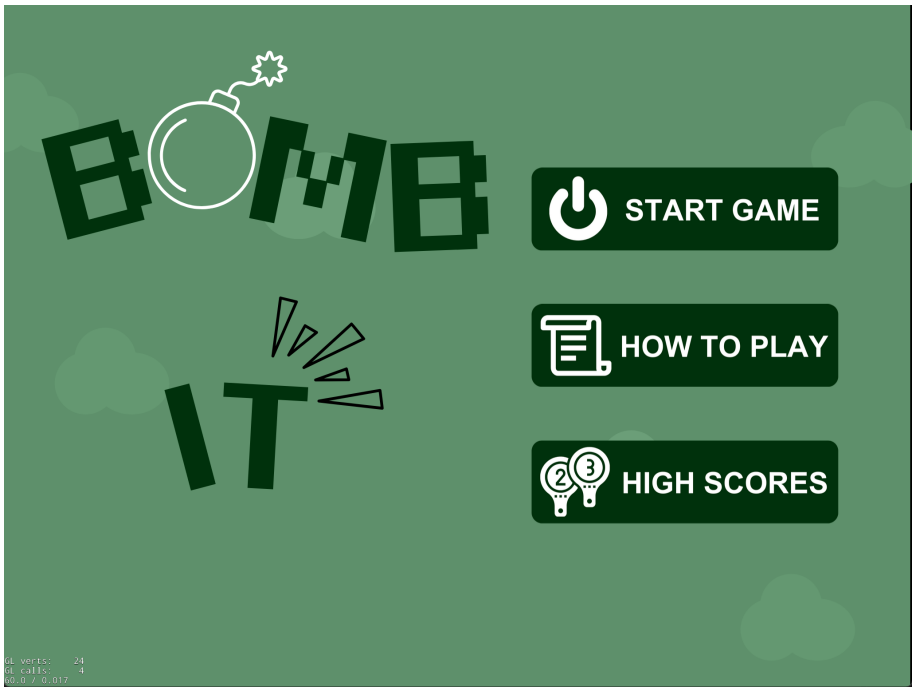

Figure 2: 游戏开始界面

2.3 游戏主循环

游戏的核心逻辑在 `GameScene::update(float dt)` 中每一帧被调用：

1. **输入处理**: 响应键盘事件，更新玩家位移向量。
2. **AI 思考**: `AIController::updateAI` 计算所有存活 AI 的下一步决策。
3. **视野更新**: 计算玩家视野半径，若开启迷雾则更新 `FogManager` 渲染。
4. **碰撞检测**:
   - 玩家 vs 火焰 (伤害判定)。

- 玩家 vs 道具 (拾取判定)。

5. **状态更新**: 更新炸弹倒计时、危险区域列表、游戏胜负条件检测。

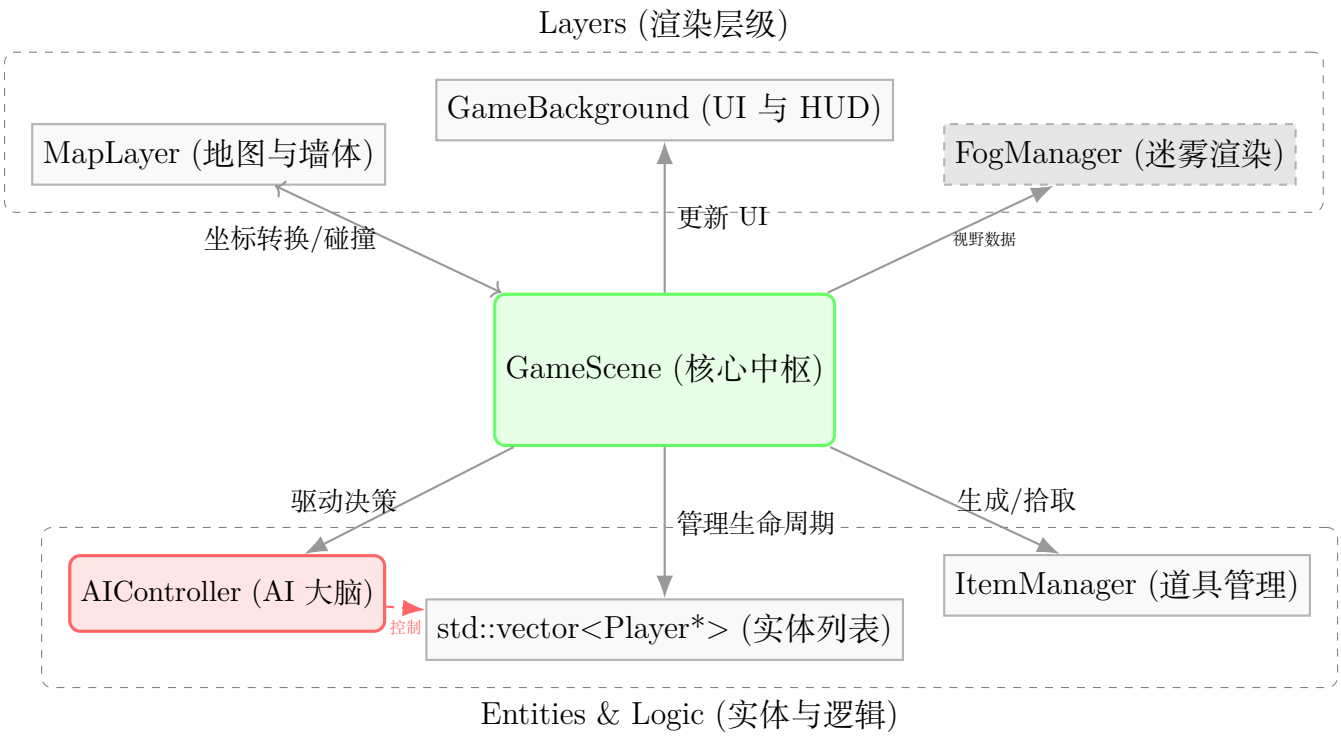

Figure 3: GameScene 核心组件架构图

3 核心模块详细实现

本章将深入解析游戏中各个核心子系统的技术实现细节，包括地图生成算法、物理与移动系统、炸弹爆炸逻辑以及特色的迷雾渲染技术。

3.1 地图系统

地图系统是游戏逻辑的基础，采用基于网格的设计方案。

3.1.1 数据结构与坐标系

地图定义为 13×13 的网格结构，单块瓦片尺寸 (`TILE_SIZE`) 设定为 108 像素。底层数据通过二维数组 `mapData[WIDTH][HEIGHT]` 维护，其数值映射如下：

- **0**: 空地 (Walkable)
- **1**: 硬墙 (Indestructible, 铁墙)
- **2**: 软墙 (Destructible, 土墙)
- **300**: 火焰 (Fatal Danger)

为了实现逻辑坐标与屏幕渲染坐标的交互，MapLayer 提供了双向转换接口：

```
// 世界坐标转网格坐标
Vec2 MapLayer::worldToGrid(const Vec2& pos) {
  Vec2 local = pos - this->getPosition();
  int gx = local.x / TILE_SIZE;
  int gy = local.y / TILE_SIZE;
  return Vec2(gx, gy);
}

```

3.1.2 地图生成与出生点保护

地图初始化采用程序化生成。算法首先遍历全图，依据设定的概率（硬墙 20%，软墙 40%）随机填充障碍物。

为了防止随机生成的墙体将玩家困死在出生点，系统引入了出生点保护算法。在生成数据后，fixSpawnArea 函数会被调用，强制清除地图四个角落（(1,1), (1,11), (11,1), (11,11)）及其相邻区域的障碍物，并确保至少有两条连通路径。

3.2 角色移动与物理判定

3.2.1 平滑移动与贴墙辅助

角色移动并非基于简单的网格跳变，而是基于速度向量的连续移动，以提供流畅的手感。Player::move 方法计算下一帧的预估位置，并调用 canMoveTo 进行碰撞检测。

为了优化由于像素级对齐导致的“卡墙”问题，系统实现了贴墙滑动辅助。当玩家试图转向但稍微错位时（例如碰到墙角边缘），算法会自动检测相邻轴向的可通行性，并在允许的情况下微调角色坐标，使其“滑”入通道。

3.2.2 状态管理

Player 类内部维护了多种状态标志位以处理复杂的战斗逻辑：

- **Invincible**: 受伤后的短暂无敌时间（闪烁效果），或拾取护盾后的持续无敌。
- **Stunned**: 被特定道具（如 Block）击中后的定身状态，期间无法响应输入。
- **isDead**: 死亡标记，触发死亡动画并在延迟后隐藏实体。

3.3 炸弹与爆炸算法

3.3.1 炸弹生命周期

炸弹放置后会向 GameScene 注册 BombDanger 预警信息，供 AI 避险使用。炸弹拥有约 2.0 秒的倒计时，倒计时结束后触发 explode()。

3.3.2 十字蔓延算法

爆炸效果采用“十字蔓延”逻辑。爆炸中心产生火焰后，向上下左右四个方向进行射线检测：

- **循环遍历**: 根据炸弹威力 (range)，在每个方向延伸 $1 \dots N$ 格。
- **硬墙判定 (Iron Wall)**: 遇到硬墙立即停止扩散。

- **软墙判定 (Soft Wall)**: 遇到软墙时，生成火焰销毁该墙体，触发道具掉落逻辑，并**停止**该方向的后续扩散（能量被墙体吸收）。
- **空地判定**: 生成火焰对象并标记地图数据为 `TILE_FLAME`。

Figure 4: 游戏界面

3.4 道具系统

3.4.1 生成机制

道具由 `ItemManager` 统一管理。当软墙被销毁时，系统根据设定的概率（默认 35%）决定是否掉落道具。

3.4.2 模式特化逻辑

系统支持根据游戏模式调整掉落策略。在迷雾模式下，为了平衡视野限制，道具生成器会强制或高概率产出 **Light (灯泡)** 道具，而在普通模式下则会屏蔽该道具。

3.4.3 效果实现

拾取道具后，`Player::pickItem` 会立即应用效果：

- **SpeedUp**: 修改 `moveSpeed` 并启动延时回调复原。
- **PowerBomb**: 增加 `bombRange`。
- **Block**: 遍历 `GameScene` 寻找最近的敌人并设置其 `stunned` 状态。

道具系统是增加游戏策略性的核心机制。所有道具均继承自 `Item` 基类，通过 `ItemType` 枚举进行区分。当玩家拾取道具时，`Player::pickItem` 函数会根据类型分发处理逻辑，修改玩家的属性或状态。

3.4.4 道具概览表

表 1 列出了游戏中所有可拾取道具及其技术参数。

Table 1: 游戏道具功能与数值设定表

道具名称	枚举类型 (ItemType)	功能与技术实现逻辑
威力加强	PowerBomb	**增加爆炸范围**。拾取后，bombRange 增加 2 格，且设置 _enhancedBombsLeft = 3。这意味着该强化仅对接下来放置的 3 枚炸弹有效。
生命恢复	Heal	**恢复血量**。若当前 hp 小于 maxHp (5)，则 hp 加 1。屏幕会弹出 “+1 HP” 提示。
无敌护盾	Shield	**无敌状态**。拾取后 invincible 和 hasShield 设为 true，持续 **5.0 秒**。期间玩家免疫炸弹火焰伤害。
强力路障	Block	**定身敌人**。搜索距离当前玩家最近的一名敌对目标，将其 stunned 状态设为 true，持续 **3.0 秒**。被定身的玩家无法移动或放弹。
极速跑鞋	SpeedUp	**移动加速**。将 moveSpeed 提升至基础速度的 **1.5 倍**，持续 **3.0 秒**。加速结束后自动恢复原速。
强光灯泡	Light	**视野扩大**。专为迷雾模式设计。将 _visionRadius 提升至 **350.0** 像素，持续 **10.0 秒**。在非迷雾模式下通常会被替换为加速道具。

3.4.5 特殊机制说明

1. 道具掉落算法道具并非随机生成在地图上，而是隐藏在软墙之中。

- **触发条件**: 当软墙被炸弹销毁时，调用 ItemManager::dropItem。
- **概率判定**: 默认掉落概率为 35%。
- **模式修正**: 在迷雾模式 (FOG) 下，道具生成算法会进行劫持，强制或高概率生成 Light 道具以辅助玩家开视野；而在非迷雾模式下，若随机到了 Light，则会强制转换为 SpeedUp 以避免产生无用道具。

2. 视觉反馈为了增强交互感,道具拾取时实现了复合动画效果 playPickAnimationEffect:

- **拾取瞬间**: 道具 Sprite 执行 “放大并淡出” (ScaleTo + FadeOut) 动画。
- **持续状态**:
  - **护盾**: 玩家身上会出现一个半透明的 shield(2).png 护罩跟随移动。
  - **加速**: 玩家脚下出现闪烁的 speedup(2).png 特效。
  - **文字飘字**: 加血或增加视野时，会在角色头顶生成向上飘动的彩色 Label (如绿色 “+1 HP”)。

3.5 迷雾系统

3.5.1 实现原理

迷雾效果通过 `RenderTexture` 和 OpenGL 混合模式实现，而非简单的贴图覆盖。

1. **底色填充**: 每一帧开始时，使用黑色（Alpha=0.96）填充全屏纹理。
2. **反向擦除**: 使用混合模式 `{GL_ZERO, GL_ONE_MINUS_SRC_ALPHA}`。
3. **柔光笔刷**: 在玩家当前位置绘制一个中心透明、边缘半透明的径向渐变 Sprite。该 Sprite 的 Alpha 通道值会“扣除”底色纹理的 Alpha 值，从而实现平滑的透视效果。

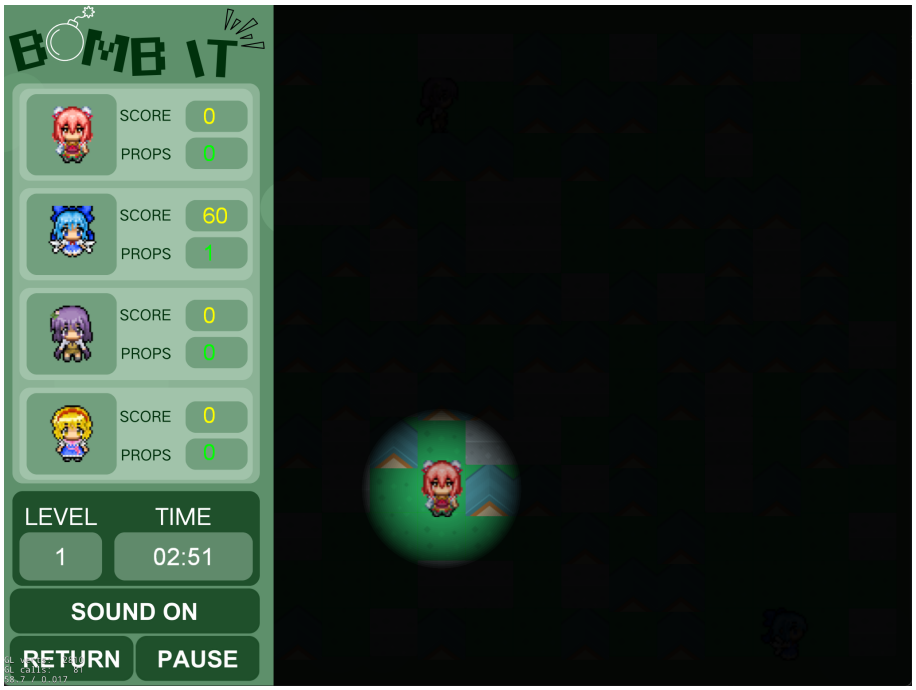

Figure 5: 迷雾系统演示（使用道具前）

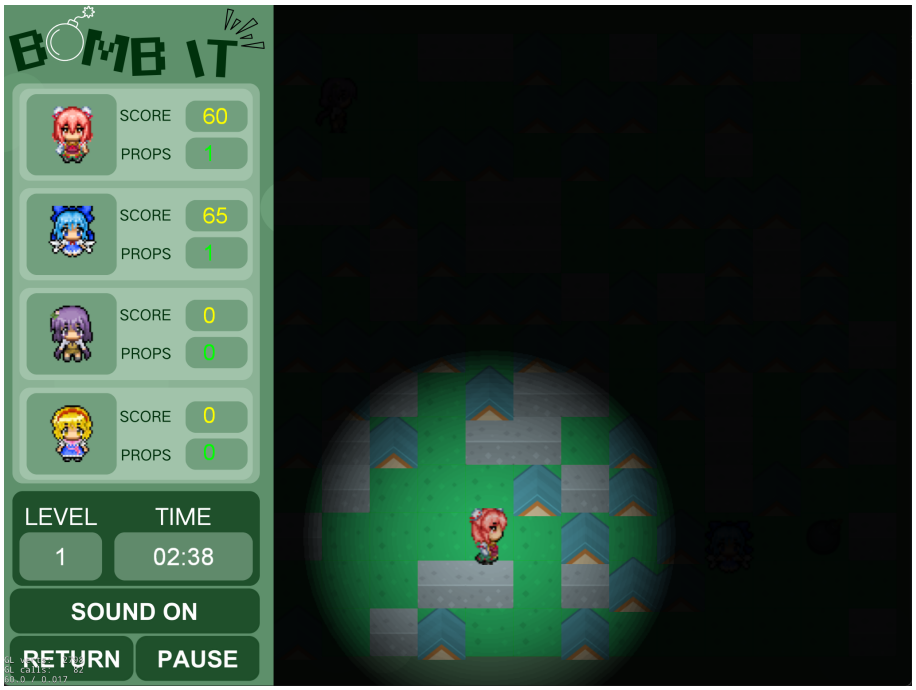

Figure 6: 迷雾系统演示（使用道具后）

c

3.5.2 动态视野

笔刷的缩放比例（Scale）与玩家的属性 `visionRadius` 绑定。当玩家拾取 Light 道具时，`visionRadius` 增大，`FogManager` 随即计算更大的笔刷尺寸，实时扩大可视区域。

4 AI 系统与核心算法

本项目实现了一套基于有限状态机与启发式搜索算法相结合的高级 AI 系统。该系统不仅能完成基本的移动与攻击，还能展现出“趋利避害”的智能行为，如躲避炸弹、预判死胡同以及战术性放置炸弹。

4.1 行为决策模型

AI 的行为控制由 `AIController` 类驱动，核心逻辑基于一个分层优先级的有限状态机。

4.1.1 状态定义

AI 的内部状态定义如下：

- **EscapeDanger (最高优先级)**: 当检测到当前位置或预定路径处于危险中（如炸弹即将爆炸），立即中断当前行为进行避险。
- **ChasePlayer (攻击)**: 锁定最近的敌人进行追踪，并在合适距离放置炸弹。
- **PickItem (贪婪)**: 搜索地图上的道具并前往拾取。
- **BreakWall (破坏)**: 寻找软墙进行炸毁，旨在获取道具或打通路径。
- **Idle (兜底)**: 当无事可做时进行随机漫步。

4.1.2 决策循环

在每一帧的 updateAI 中，系统按照以下逻辑链进行决策：

1. **冷却检查**：通过 thinkCooldown 模拟反应延迟，避免 AI 反应过于非人类地迅速。
2. **生存优先**：调用 tryEscapeDanger。若当前位置热力值过高，强制切换至逃生状态。
3. **性格判定**：若环境安全，根据 aggressiveness (攻击欲) 和 curiosity (好奇心) 等随机参数，决定是去吃道具、炸墙还是攻击玩家。

4.2 热力图系统

为了量化地图上的“危险程度”，系统引入了热力值算法。函数 getHeatValue(grid) 会返回指定网格的危险评分，评分越高代表该区域越危险。

危险分值的计算公式近似为：

$$Score = S_{flame} + S_{bomb} + S_{terrain}$$

其中：

- S_{flame}：若该格有火焰，固定 $+100$（极度危险）。
- S_{bomb}：若该格在炸弹爆炸范围内，基础分 $+70$。此外，炸弹剩余时间越短，分数越高（$TimeFactor$），迫使 AI 优先远离即将爆炸的炸弹。
- $S_{terrain}$：若该格是死胡同（出口 ≤ 1），$+25$，防止 AI 钻入死角被堵杀。

4.3 路径规划算法

项目实现了多种寻路策略以应对不同情境，所有寻路逻辑均封装在 GameScene 中。

4.3.1 基于权重的 A* 算法

用于常规寻路（如追踪玩家、找道具）。与传统 A* 不同，本项目的代价函数 $f(n)$ 引入了热力值：

$$f(n) = g(n) + h(n) + (Heat(n) \times 0.5)$$

通过将 getHeatValue 叠加到移动代价 $g(n)$ 中，AI 会自动规划出一条“既能到达目标，又能避开危险区域”的路径。例如，如果直线路经上有炸弹，AI 会绕路行走。

4.3.2 BFS 逃生搜索

当 AI 处于危险状态（EscapeDanger）时，需要以最快速度找到最近的安全点。此时采用广度优先搜索 (BFS) 算法：

1. 以当前 AI 坐标为起点进行层序遍历。
2. 检查每个邻居节点的 getHeatValue。
3. 一旦找到热力值 < 0.1 的安全格，立即停止搜索并生成路径。

4.4 战术预判

AI 在攻击时不会盲目放置炸弹，而是包含了一层自我保护逻辑。

4.4.1 安全放弹检测

在 `tryAttackPlayer` 或 `tryDestroyWall` 状态下，AI 准备放炸弹前会调用 `hasSafeEscape`：

- **模拟演练**：假设当前位置已经放置了炸弹。
- **路径验证**：检查在该虚拟炸弹的爆炸倒计时内，是否存在一条通往安全区域的路径。
- **决策执行**：只有当“逃生路径存在”时，AI 才会真正执行放弹操作，否则会放弃攻击以保全自身。

4.4.2 敌我识别

`findNearestPlayer` 函数在搜索目标时增加了阵营判断，AI 会忽略同阵营的 AI 队友，优先锁定人类玩家或敌对阵营的 AI，实现了基础的团队协作雏形。

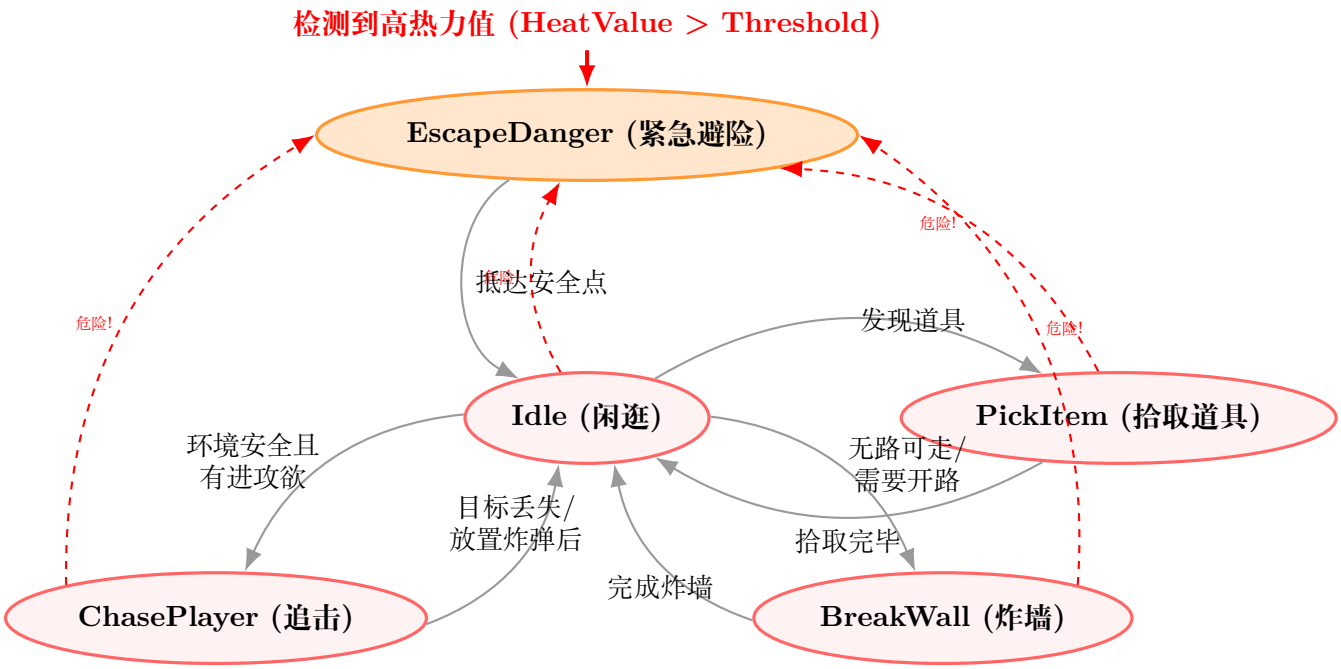

Figure 7: AI 有限状态机 (FSM) 转移图

5 数据持久化与音频系统

为了提升用户体验，游戏集成了完整的数据存储机制用于保存排行榜与用户偏好，并构建了稳健的音频管理模块以处理背景音乐与音效的播放状态。

5.1 数据持久化

项目利用 Cocos2d-x 提供的 `UserDefault` 轻量级数据库（基于 XML/Plist）来实现本地数据的读写操作。

5.1.1 排行榜存储机制

排行榜系统旨在记录并展示历史最高的 5 次得分。其核心逻辑封装在 `HighScoresScene::saveScore` 中，具体流程如下：

1. **读取历史数据**：遍历键值 `"HighScore0"` 至 `"HighScore4"`，将已存在的非负分数加载至内存容器中。
2. **插入与排序**：将当前局的新分数加入容器，并使用 `std::sort` 进行降序排列。
3. **截断与回写**：
   - 若容器大小超过 5，则截断多余数据，仅保留前 5 名。
   - 遍历容器，将更新后的分数重新写入 `UserDefault`。
4. **强制刷新**：调用 `flush()` 确保数据立即写入物理存储设备，防止意外退出导致数据丢失。

```
// 核心保存逻辑片段
std::sort(scores.begin(), scores.end(), [](int a, int b) {
  return a > b;
});
if (scores.size() > 5) scores.resize(5);

for (int i = 0; i < scores.size(); ++i) {
  std::string key = StringUtils::format("HighScore%d", i);
  UserDefault::getInstance()->setIntegerForKey(key.c_str(), scores[i]);
}
UserDefault::getInstance()->flush();

```

5.1.2 用户偏好设置

除了分数，系统还负责持久化双人模式下的角色选择状态。在 `SelectScene` 中，一旦 P1 和 P2 确认选角，系统会将 `SelectedCharacter1` 和 `SelectedCharacter2` 写入本地，以便在后续流程中读取。

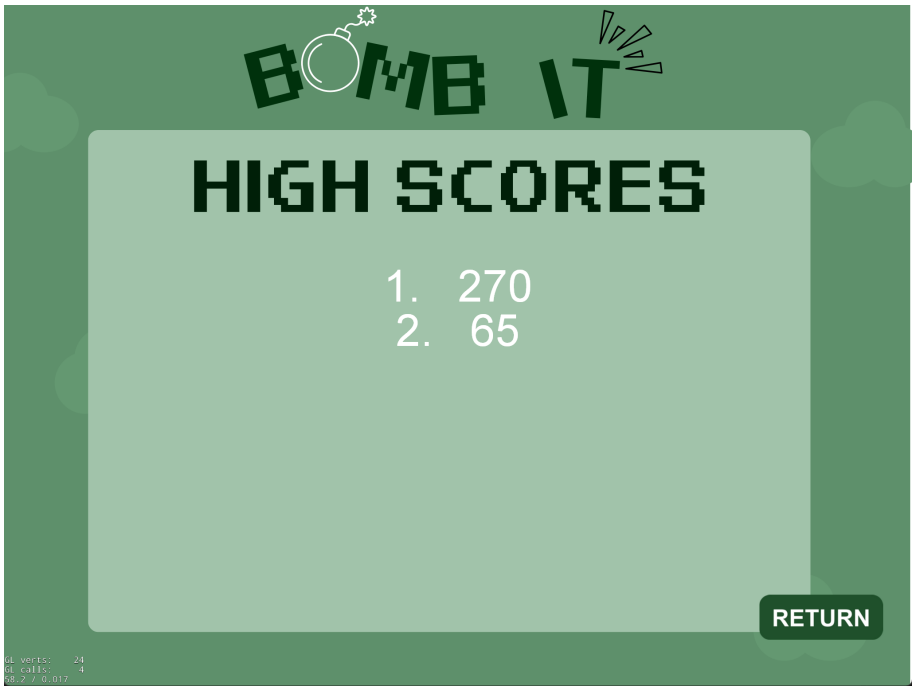

Figure 8: 分数界面

c

5.2 音频系统

音频模块基于 `cocos2d::experimental::AudioEngine` 实现，支持多声道混音、循环播放及全局状态管理。

5.2.1 全局音频管控

为了在不同场景间保持音效状态的一致性，`GameScene` 维护了静态变量 `s_isAudioOn` 作为全局开关。

- **状态同步**：在 `GameBackground` 的设置按钮中切换静音时，会直接修改此静态变量，并调用 `AudioEngine::pauseAll()` 或 `resumeAll()`。
- **场景切换**：新场景初始化时（如 `StartScene::init`），会首先检查此开关，若处于静音状态，则自动暂停刚初始化的背景音乐。

5.2.2 生命周期管理

音频系统与应用生命周期紧密绑定，以确保应用切入后台时不会产生噪音干扰。在 `AppDelegate` 中实现了以下回调：

- **进入后台**：触发 `applicationDidEnterBackground`，调用 `Director::stopAnimation()` 暂停游戏循环，并执行 `AudioEngine::pauseAll()` 挂起所有音频。
- **返回前台**：触发 `applicationWillEnterForeground`，恢复动画循环及音频播放。

5.2.3 音效分类处理

系统对背景音乐 (BGM) 和音效 (SE) 进行了逻辑区分：

- **BGM**: 如 `GameBackgroundSound.mp3`，设置为 `loop=true`，并通过持久化的 `audioID` (如 `s_gameAudioID`) 进行句柄管理，防止场景重入时重复播放。
- **SE**: 如 `bomb.mp3` 或 `stepsSound.mp3`，通常为一次性播放。对于脚步声等持续性音效，采用了 ID 追踪机制，在角色停止移动时通过 ID 强制停止特定音效。

6 项目总结与未来展望

6.1 项目成果概述

本项目基于 **Cocos2d-x 3.17.2** 引擎与 **C++** 语言，成功构建了一款功能完备的“Q 版泡泡堂”休闲竞技游戏。项目在保证核心玩法（炸弹放置、障碍销毁、道具拾取）还原度的基础上，实现了流畅的 60FPS 运行体验。

系统集成了多种游戏模式，包括经典的单人闯关、本地双人对战以及极具策略性的迷雾模式，并通过 `UserDefault` 实现了基于 XML 的数据持久化，保证了游戏进度的可保存性。

6.2 核心技术亮点

回顾开发过程，本项目的技术实现具有以下显著特征：

- **智能化的 AI 决策系统**: 摒弃了简单的随机游走，采用了有限状态机结合热力图的设计。AI 能够识别炸弹爆炸范围、计算死胡同风险，并利用 A* 算法规划最优路径，实现了“趋利避害”的高级智能行为。
- **高性能的迷雾渲染**: 在迷雾模式中，利用 `RenderTexture` 与 OpenGL 混合模式实现了动态视野擦除。相比传统的静态遮罩，该方案支持任意形状的笔刷与平滑的边缘过渡，且性能开销极低。
- **模块化的架构设计**: 采用了“管理者模式”对系统进行解耦。`GameScene` 作为总控，协调道具、视觉、逻辑与数据协同工作，各模块职责单一，便于后续维护与扩展。
- **稳健的地图生成算法**: 通过程序化生成算法 (Procedural Generation) 构建关卡，并引入“出生点保护机制”与“连通性修正”，有效解决了随机地图可能导致的死局问题。

6.3 存在的问题与未来展望

尽管项目已完成核心功能闭环，但在功能扩展性与性能深度优化上仍有改进空间：

1. **网络联机功能的实现**: 目前主菜单的“Online”入口仅作为 UI 展示，底层网络模块尚未实装。未来计划引入 BSD Socket 或 WebSocket，采用帧同步技术实现局域网或广域网的多人在线对战。
2. **对象池技术的引入**: 当前游戏中的炸弹 (`Bomb`) 与火焰 (`Flame`) 对象在销毁时直接从内存移除。在激烈战斗中，高频的 `new/delete` 操作可能导致内存碎片。未来应实现 `ObjectPool` 机制，复用这些高频对象以降低 CPU 开销。

3. **可视化地图编辑器**: 目前的地图完全依赖随机生成，缺乏设计感。未来可开发基于 Tiled Map Editor 的加载器，允许设计者手动编辑关卡地形与敌人配置，丰富游戏内容。

4. **AI 行为的多样化**: 当前的 AI 性格参数（如攻击欲、好奇心）虽有区分，但战术风格趋同。未来可扩展状态机，增加“伏击”、“协作包抄”等高级战术状态，使 PVE 体验更加丰富。