贪吃蛇游戏软件分析与设计说明书

1. 引言

1.1 项目背景

贪吃蛇游戏作为20世纪70年代诞生的经典街机游戏，凭借其简单的操作规则和渐进式难度机制，历经数十年仍保持着广泛的影响力。传统版本以单机模式为主，玩家通过方向键操控蛇体移动并吞食随机生成的食物，蛇身长度随得分增加而延长，游戏失败条件通常为蛇头触碰边界或自身躯体。然而，在互联网技术与用户需求快速迭代的背景下，传统玩法已难以满足现代用户对数据留存、社交互动及个性化体验的诉求。为此，本系统在保留核心玩法的基础上，深度融合用户管理与游戏日志分析功能，构建了一套具备可扩展性的现代化游戏框架。通过引入用户注册登录机制，系统支持多账户独立存储游戏数据；游戏日志表记录每局游戏的时长、得分及操作轨迹，为后续用户行为分析与难度自适应调整提供数据支撑。项目开发过程中采用团队协作模式，依托ProcessOn在线工具完成功能模块设计与流程图绘制，既保证了设计规范性，又通过实时协作提升了开发效率。此外，本项目的实施不仅是对经典游戏的重构实践，更可作为计算机专业学生理解软件工程全流程的综合性案例。

1.2 设计目的

本文档的核心目标在于系统化阐述贪吃蛇游戏软件的开发全貌，为不同角色的参与者提供精准指导。首先，针对开发团队，文档详细解构了功能模块层次方框图（任务1.1），明确划分游戏控制、用户系统、数据存储三大核心模块的交互逻辑，避免开发过程中出现职责重叠或功能遗漏。其次，通过定义标准化的用户表与游戏日志表结构（任务1.2），规范数据库设计范式，确保数据一致性与查询效率。在界面优化层面（任务1.3），文档提供高保真原型图与交互逻辑说明，指导开发人员实现响应式布局与视觉美化。针对算法优化（任务1.4），流程图与性能对比数据直观展示碰撞检测逻辑改进方案，帮助开发人员理解优化前后的技术差异。此外，文档还承担项目管理职能：GitHub仓库的规范化配置说明（任务2）为版本控制提供操作指南，WBS分解结构（任务3）则通过任务编码与时间分配表，确保项目里程碑可控。对于维护阶段，文档记录的测试用例集、监控指标定义（如帧率、崩溃率）以及用户反馈处理流程，为后续功能迭代与故障排查提供完整技术依据，最终形成"开发-交付-维护"的全生命周期管理闭环。

1.3 适用范围

本文档的适用对象涵盖软件开发全流程的关键参与者：

开发团队：前端工程师可依据界面原型图（任务1.3）进行组件化开发；后端工程师需遵循数据库表结构定义（任务1.2）实现API接口；算法工程师则需基于优化后的流程图（任务1.4）重构移动控制逻辑。

测试团队：集成测试人员需参考功能模块方框图（任务1.1）设计端到端测试用例，性能测试团队需根据算法优化说明（任务1.4）验证响应时间与资源占用率是否符合预期。

项目管理层：技术经理可通过WBS分解表（任务3）跟踪任务进度，质量保障主管需依据GitHub提交记录（任务2）审计代码规范性，运维团队则依赖监控方案（维护计划2.1）配置告警规则。

外部关联方：高校指导教师可将本文档作为软件工程实践课程的评分依据，通过对比WBS时间记录（任务4.3）与代码提交日志，评估团队协作效率；第三方审计机构则可核查文档中声明的功能点（如用户数据加密存储）是否与实现代码一致。此外，文档中ProcessOn绘制的设计图和GitHub仓库管理规范，亦可为同类项目的开发提供方法论参考。

2. 需求分析

2.1 功能需求

1. 用户管理功能

用户注册

用户登录

2. 游戏核心功能

- 游戏地图绘制

- 蛇的移动控制

- 食物生成与碰撞检测

- 游戏速度调节

- 游戏暂停/继续

- 游戏结束判定

3. 游戏日志功能

- 游戏记录存储

- 游戏记录查询

2.2 非功能需求

1. 性能需求：游戏运行流畅，响应时间<100ms

2. 可靠性：游戏数据不丢失

3. 兼容性：Windows平台控制台应用

4. 可扩展性：便于添加新功能

3. 系统设计

3.1 系统架构

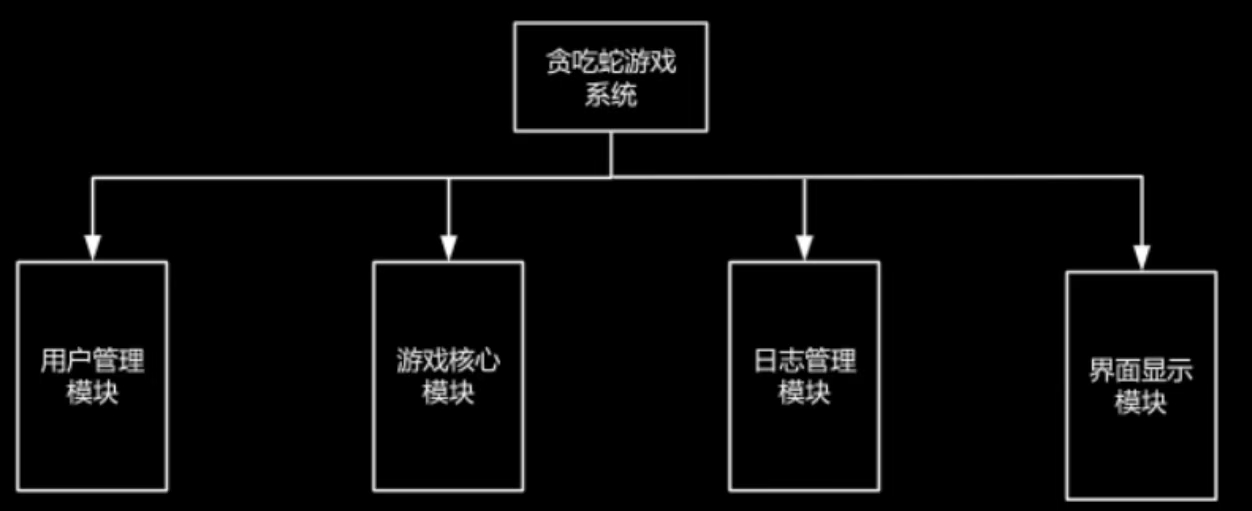


图1 系统架构图

3.2 功能模块设计

3.2.1 用户管理模块

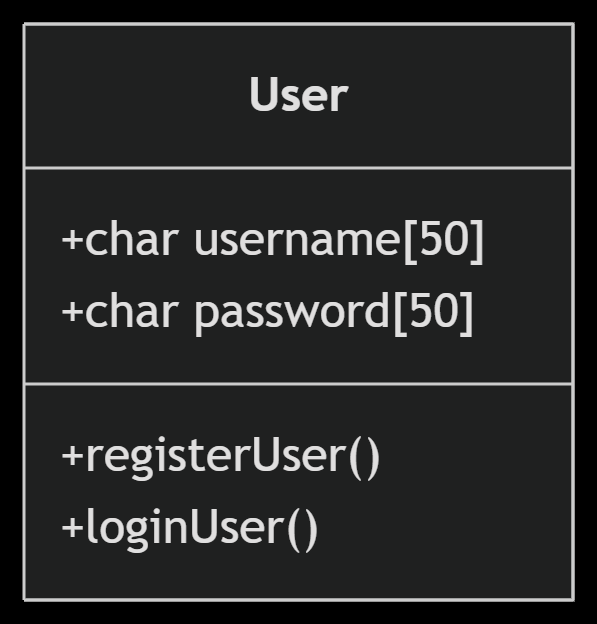


图2 管理模块

3.2.2 游戏核心模块

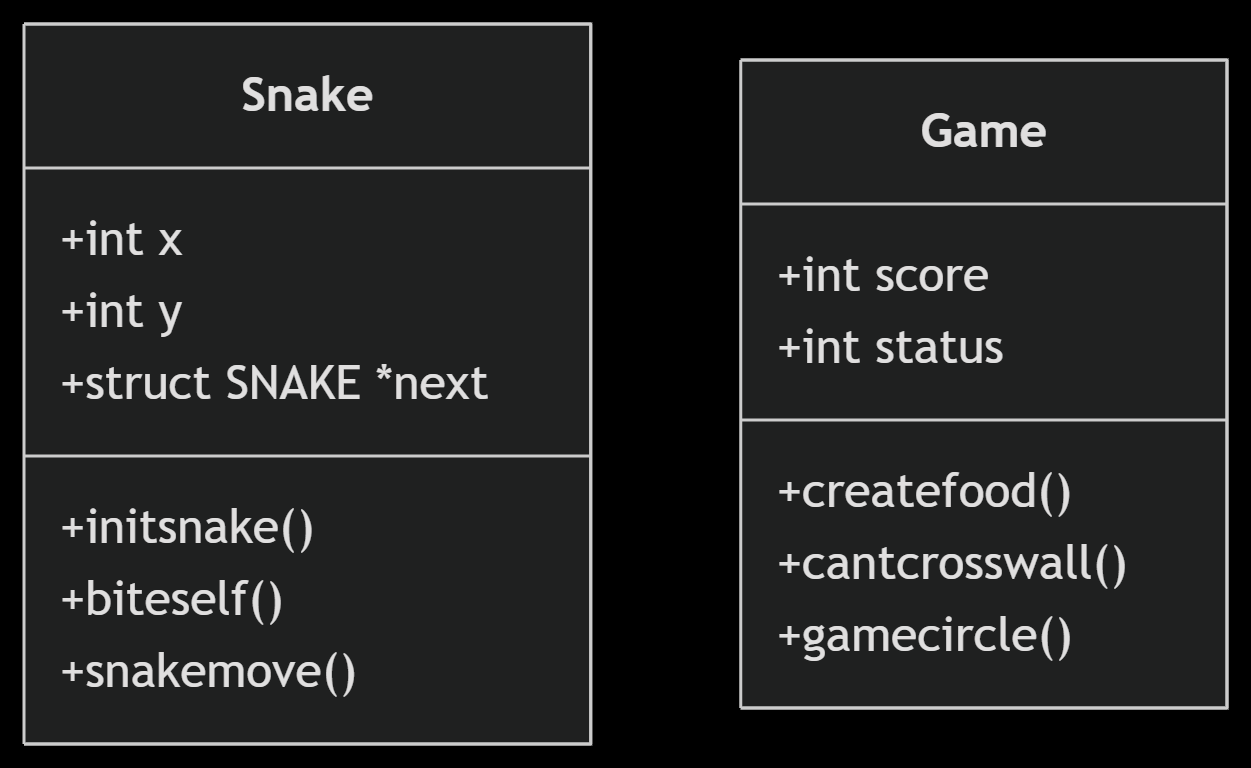


图3 核心模块

3.2.3 日志管理模块



图4 日志管理模块

3.3 数据结构设计

3.3.1 蛇身节点结构

typedef struct SNAKE {

int x; // x坐标

int y; // y坐标

struct SNAKE \*next; // 指向下一个节点的指针

} snake;

typedef struct {

char username[50]; // 用户名

char password[50]; // 密码

} User;

3.3.3 游戏日志结构

typedef struct {

char username[50]; // 用户名

int score; // 得分

time\_t startTime; // 开始时间

int duration; // 持续时间(秒)

} GameLog;

3.4 关键算法设计

3.4.1 蛇移动算法

1. 根据当前方向计算新蛇头位置

蛇前进,上 U,下 D,左 L,右 R，此处举例向下的情况

    snake \*nexthead;

    cantcrosswall();

    nexthead = (snake \*)malloc(sizeof(snake));

    if (status == U)

    {

        nexthead->x = head->x;

        nexthead->y = head->y - 1;

        if (nexthead->x == food->x && nexthead->y == food->y)

        { // 如果下一个有食物//

            nexthead->next = head;

            head = nexthead;

            q = head;

            while (q != NULL)

            {

                Pos(q->x, q->y);

                printf("■");

                q = q->next;

            }

            score = score + add;

            createfood();

        }

        else

        { // 如果没有食物//

            nexthead->next = head;

            head = nexthead;

            q = head;

            while (q->next->next != NULL)

            {

                Pos(q->x, q->y);

                printf("■");

                q = q->next;

            }

            Pos(q->next->x, q->next->y);

            printf(" ");

            free(q->next);

            q->next = NULL;

        }

    }

2. 检查是否吃到食物

- 吃到食物：增加蛇身长度，生成新食物

- 未吃到食物：移动蛇身，删除尾部节点

if (nexthead->x == food->x && nexthead->y == food->y)

        { // 有食物

            nexthead->next = head;

            head = nexthead;

            q = head;

            while (q != NULL)

            {

                Pos(q->x, q->y);

                printf("■");

                q = q->next;

            }

            score = score + add;

            createfood();

        }

        else

        { // 没有食物

            nexthead->next = head;

            head = nexthead;

            q = head;

            while (q->next->next != NULL)

            {

                Pos(q->x, q->y);

                printf("■");

                q = q->next;

            }

            Pos(q->next->x, q->next->y);

            printf(" ");

            free(q->next);

            q->next = NULL;

        }

3. 检查碰撞情况

int biteself()

{ // 判断是否咬到了自己

    snake \*self;

    self = head->next;

    while (self != NULL)

    {

        if (self->x == head->x && self->y == head->y)

        {

            return 1;

        }

        self = self->next;

    }

    return 0;

}

3.4.2 食物生成算法

1. 随机生成食物坐标

2. 检查是否与蛇身重叠

3. 如重叠则重新生成

void createfood()

{

    int isOverlapping;

    snake \*food\_1;

    srand((unsigned)time(NULL));

    food\_1 = (snake \*)malloc(sizeof(snake));

    do

    {

        isOverlapping = 0;

        food\_1->x = rand() % 52 + 2;

        food\_1->y = rand() % 24 + 1;

        if ((food\_1->x % 2) != 0)

            food\_1->x--; // 确保 x 坐标为偶数，与蛇头对齐

        for (q = head; q != NULL; q = q->next)

        {

            if (q->x == food\_1->x && q->y == food\_1->y)

            {

                isOverlapping = 1;

                break;

            }

        }

    } while (isOverlapping); // 如果食物与蛇身重合，重新生成

    Pos(food\_1->x, food\_1->y);

    food = food\_1;

    printf("■");

}

3.4.3 碰撞检测算法

墙壁碰撞检测：检查蛇头是否碰到地图边界

void cantcrosswall()

{ // 不能穿墙

    if (head->x == 0 || head->x == 56 || head->y == 0 || head->y == 26)

    {

        endgamestatus = 1;

        endgame();

    }

}

4. 详细设计

4.1 主程序流程图

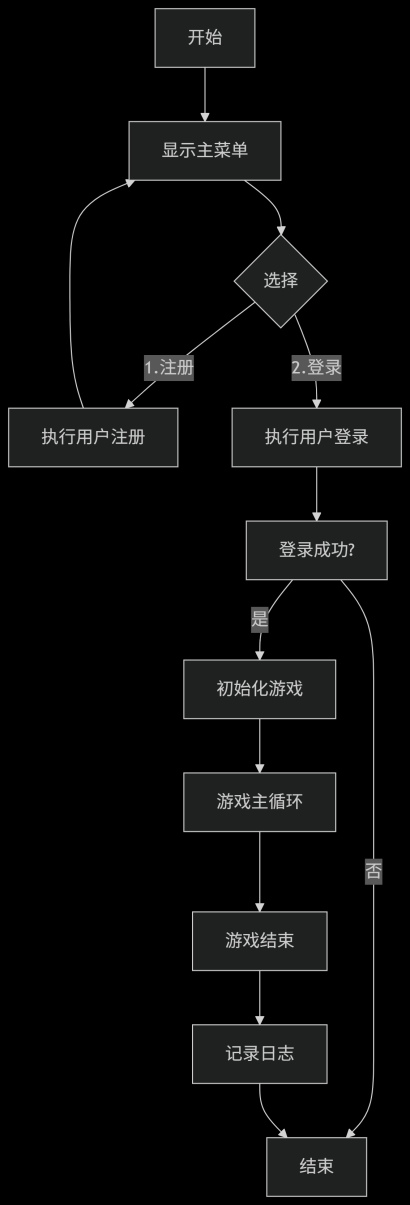


图5 主程序流程图

4.2 游戏主循环流程图

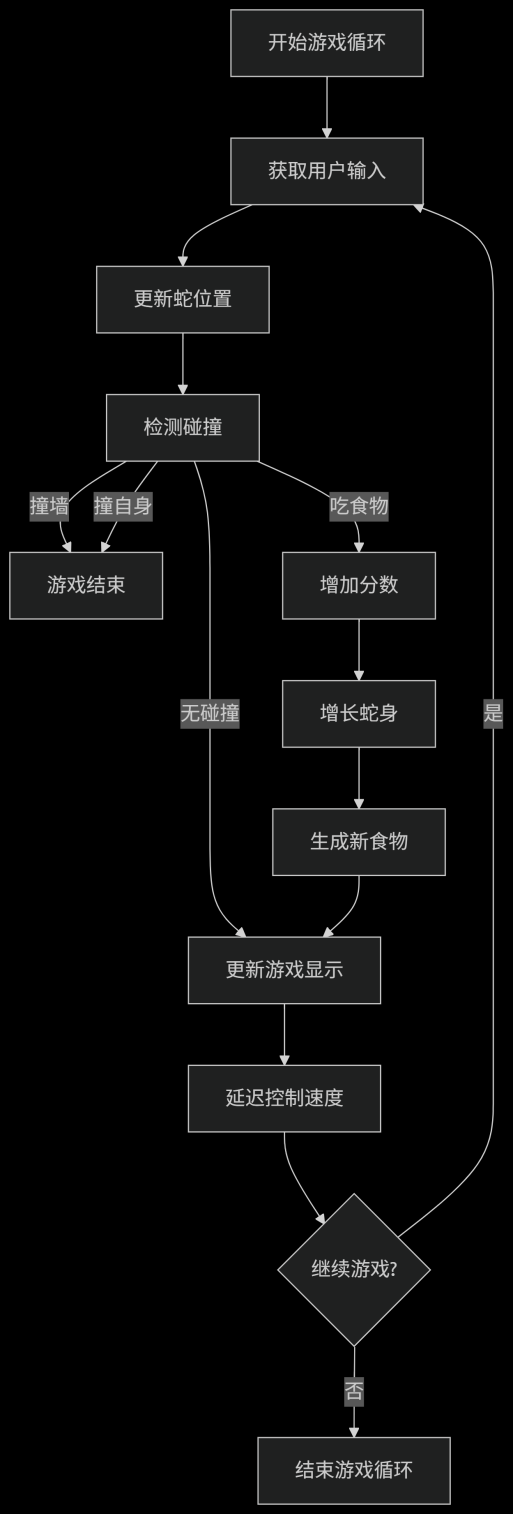


图6 主循环流程图

5. 用户界面设计

5.1 登录/注册界面

1. 注册

2. 登录

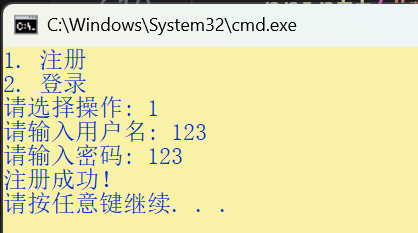


图7 登录注册

5.2 游戏主界面

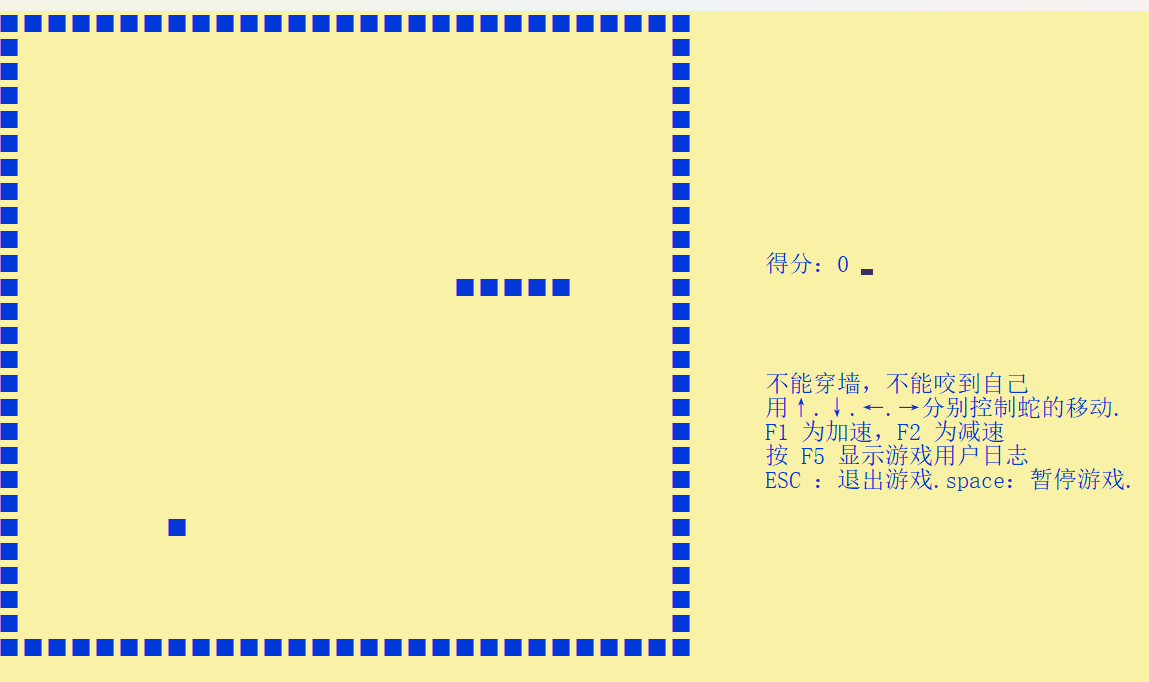


图8 游戏主界面

6. 数据库设计

6.1 用户表(users)

表1 用户表

|  |  |  |
| --- | --- | --- |
| 字段名 | 类型 | 说明 |
| username | VARCHAR(50) | 用户名(主键) |
| password | VARCHAR(50) | 密码 |

6.2 游戏日志表(game\_logs)

表2 游戏日志表

|  |  |  |
| --- | --- | --- |
| 字段名 | 类型 | 说明 |
| username | VARCHAR(50) | 用户名 |
| score | INT | 游戏得分 |
| startTime | DATETIME | 游戏开始时间 |
| duration | INT | 游戏持续时间(秒) |
| end\_status | TINYINT | 游戏结束状态 |

7. 测试计划

7.1 测试用例

1. 用户注册测试

- 输入有效用户名密码

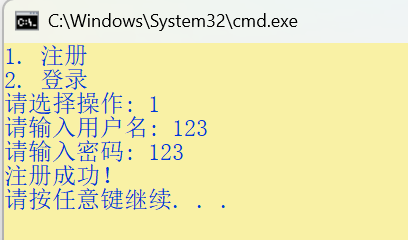


图9 注册测试

2. 用户登录测试

- 正确用户名密码

- 错误用户名或密码

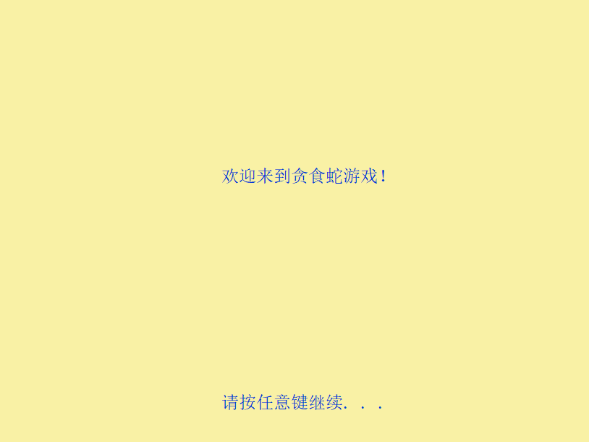


图10 正确输入密码测试

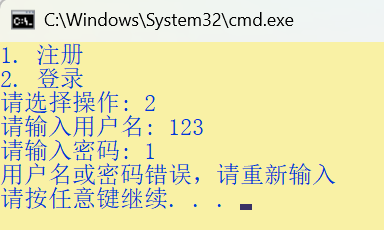


图11 错误输入密码测试

3. \*\*游戏功能测试\*\*

- 蛇移动方向控制

- 食物生成位置

- 碰撞检测

- 分数计算

- 游戏速度调整

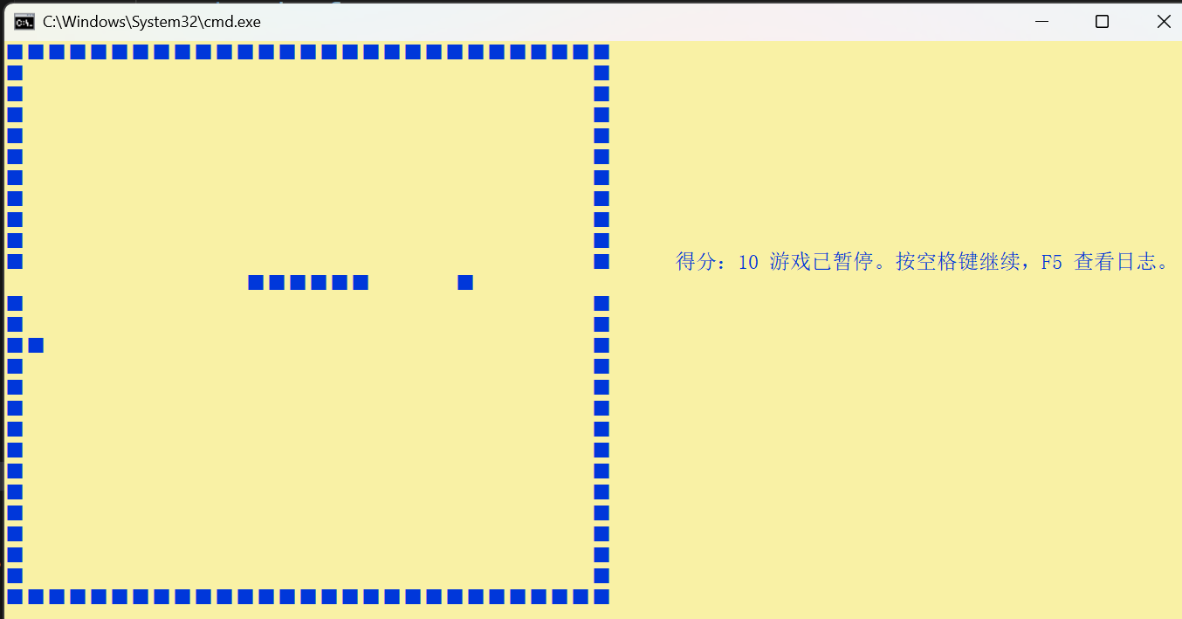


图11 游戏功能测试

4. 日志功能测试

- 游戏记录存储

- 游戏记录查询

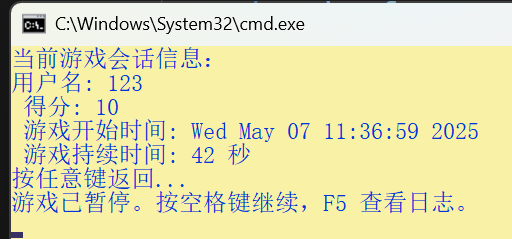


图12 日志功能测试

8. 部署与维护

8.1 系统部署

1. 编译环境：Windows + GCC/MinGW

2. 运行环境：Windows控制台

3. 依赖库：windows.h, time.h, stdio.h等标准库

8.2 维护计划

1. 定期备份用户数据和游戏日志

为确保系统数据的安全性与可恢复性，维护计划首先制定了详细的数据备份策略。基于任务1（2）中定义的用户表与游戏日志表结构，用户表采用每日全量备份与6小时增量备份相结合的方式，确保账户信息实时存档；游戏日志表则按周归档历史数据并保留最近3个月记录，平衡存储成本与数据可用性。通过编写Python自动化脚本或利用数据库工具（如mysqldump），备份文件将同步至云端存储（如阿里云OSS），避免本地硬件故障导致的数据丢失。此外，每月进行备份数据完整性验证，模拟恢复流程以确认备份有效性，并将备份日志纳入实验报告（任务4）中，形成闭环管理。该方案直接关联实验任务中的数据库设计与GitHub文档管理（任务2），备份脚本代码存放于仓库的scripts/backup目录，确保团队可追溯与协作维护。

2. 监控系统运行状态

系统监控旨在实时掌握软件性能与稳定性，其设计紧密结合任务1的算法优化与功能测试成果。通过部署Prometheus与Grafana开源工具，监控核心指标包括游戏帧率、算法响应时间及崩溃率等稳定性参数。异常阈值设置（如CPU占用率超80%触发告警）可及时通知团队PM，避免运行中断。监控数据每日生成报告，存档于GitHub仓库的docs/monitoring目录，并与实验报告中的测试结果交叉分析，持续优化系统性能。该模块不仅保障了当前版本的可靠运行，还为后续迭代提供数据支撑，例如通过对比优化前后的帧率变化，验证算法改进的实际收益。

3. 根据用户反馈进行功能更新

功能更新机制以用户需求为导向，形成“收集-分析-迭代”的闭环。在任务1（3）优化的界面中嵌入反馈入口，结合游戏日志表的行为数据分析高频痛点，优先处理界面交互问题（如按钮响应延迟）及算法性能调优（如碰撞检测精度）。版本管理依托GitHub仓库（任务2），通过Pull Request审核代码变更，使用git tag标记版本号（如v1.1.0），确保更新过程可追溯。每次更新同步修订《分析与设计说明书》，并通过回归测试（复用任务1（5）的测试用例）验证兼容性。此流程将用户反馈直接转化为开发行动，强化了实验成果的实用性，形成从开发到维护的完整生命周期管理。