深 圳 大 学 实 验 报 告

课程名称 计算机游戏开发

项目名称 实验 4　游戏中的人工智能

学　　院 数学科学学院

专　　业 信息与计算科学（数学与计算机实验班）

指导教师 储 颖

报 告 人 王曦 学号 2021192010

实验时间 2024 年 05 月 14 日

提交时间 2024 年 05 月 14 日

教务处制

目录

一、实验目的与要求

1. 理解 A*寻路算法原理。
2. 进一步熟悉地图编辑器的使用。
3. 实现游戏中的人工智能。

二、实验内容与方法

1．完成游戏编译（5 分)

成功编译并运行教材 P200“游戏 AI 实例-迷宫寻宝”。

2．完成修改内容一 （5 分）

修改游戏代码，实现修改内容一，即修改窗口大小。

3．完成修改内容二 （10 分）

修改游戏代码，实现修改内容二，即增加 DANGEROUS 区域。

4．完成修改内容三 （10 分）

修改游戏代码，实现修改内容三，即增加按键监听。

5．完成修改内容四 （10 分）

修改游戏代码，实现修改内容四，即增加地图层次。

6．完成 Bug 修改（5 分）

通过更改游戏代码的方式修复 BUG，如：点击笑脸出现“No Way” 和“Found Treasure”的叠加。

7．添加配乐、音效（5 分）

为游戏添加配乐、音效。

8．增加粒子特效（5 分）

为游戏添加粒子特效。

9. 游戏优化升级（5 分）

自行发挥想象力，优化游戏功能。

10．完成实验报告 （40 分）

截图记录关键步骤，分析实验结果，撰写心得体会。

三、实验步骤与过程

（记录关键步骤/设计过程/设计结果的截图）

1. 重构与编译

1.1 A*算法

1.1.1 A*算法简介

A* 算法类似于 Dijkstra 算法，是对 BFS 的优化。朴素 BFS 中直接从起点搜到终点可能会经过很多状态，而 A* 算法中加入启发函数(估价函数)，使得只需搜较少的状态即可找到从起点到终点的一条最短路。A* 算法只在搜索空间很大时才有明显的优化效果。A* 算法和 Dijkstra 算法都能解决边权非负的图中的最短路。Dijkstra 算法可看作从每个点到终点的估计距离都是 0 的最短路。

A* 算法中每个点可能会被扩展多少次，这与 BFS 和 Dijkstra 算法不同: ① BFS 入队时判重; ② Dijkstra 算法出队时判重; ③ A*算法不判重。

【A* 算法的成立条件】估计距离 ≤ 真实距离。

【A* 算法的应用场景】确定有解。若无解时，A* 算法会将整个搜索空间都搜索一遍，效率低于朴素 BFS 。但实际应用中未必知道是否有解，也可用 A* 算法，大部分时候比朴素 BFS 快。

【定理】当终点第一次出队时已确定最短距离。

【证】设终点 u 当前出队，此时 u 与起点的距离为 d(u) 。

设起点到 u 的真实距离为 dis(u) ，u 到终点的估计距离和真实距离分别为 f(u) 和 g(u) 。

设真实的最短路径为 d ，则 d=dis(u)+g(u)≥ dis(u)+f(u) 。

若 u 出队时不是最短距离，则 d(u)>d≥ dis(u)+f(u) ，且是最短距离的节点在队列中。

这表明: 当前出队的不是队列中的最小值，矛盾。

【注】A* 算法只能保证终点出队时是最短距离，不能保证其他节点。

A* 算法将 BFS 中的队列换为小根堆，队列中不仅存起点到当前点的真实距离，还存该点到终点的估计距离。每次选择与终点的估计距离最小的点扩展。

1.1.2 A*算法的实现

【题意】输入一个 n * m 的字符矩阵，其中字符 '.' 表示空地，'#'表示障碍。某人从迷宫的左上角 (1, 1) 出发，要到达迷宫的右下角 (n, m) 。他每次只能上、下、左、右走，且不能走到障碍物中。问他是否存在从起点到终点的路径？若不存在，输出“No”；否则第一行输出“Yes”，第二行输出最短路径的长度，第三行输出最短路径。

【输入】

```
3 5
.##.#
.#...
```

```
...#.
```

【输出】

```
Yes
(1, 1)
(2, 1) -> (3, 1) -> (3, 2) -> (3, 3) -> (2, 3) -> (2, 4) -> (2, 5) -> (3, 5) ->
```

【C++实现】

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

const int INF = 0x3f3f3f3f;

using pii = pair<int, int>;
#define x first
#define y second

using T = tuple<int, int, pii>;   // { 到终点的估计距离, 到起点的实际距离, 节点坐标 }

const vector<int> dx = { -1, 0, 0, 1 };
const vector<int> dy = { 0, -1, 1, 0 };

int n, m;   // 迷宫大小
vector<string> a;   // 迷宫
vector<vector<pii>> pre;   // pre[x][y] 表示点 (x, y) 的前驱

// 求点 A 到点 B 的 Manhattan 距离
int getDistance(pii A, pii B) {
    return abs(A.x - B.x) + abs(A.y - B.y);
}

// 检查坐标 (x, y) 是否能走
bool check(int x, int y) {
    return 1 <= x && x <= n && 1 <= y && y <= m && a[x][y] != '#';
}

vector<vector<int>> Astar(pii st, pii ed) {
    vector<vector<int>> dis(n + 5, vector<int>(m + 5, INF));
    pre = vector<vector<pii>>(n + 5, vector<pii>(m + 5));
    dis[st.x][st.y] = 0;
    pre[st.x][st.y] = pii(0, 0);

    priority_queue<T, vector<T>, greater<T>> heap;
    heap.push({ getDistance(st, ed), 0, st });
    while (heap.size()) {
```

```
            auto [_, real, cur] = heap.top();
            heap.pop();

            auto [curX, curY] = cur;

            // 到达终点
            if (curX == ed.x && curY == ed.y) {
                break;
            }

            for (int i = 0; i < 4; i++) {
                int tmpX = curX + dx[i], tmpY = curY + dy[i];

                // 扩展
                if (check(tmpX, tmpY)) {
                    if (dis[tmpX][tmpY] > real + 1) {
                        dis[tmpX][tmpY] = real + 1;
                        heap.push({ dis[tmpX][tmpY] + getDistance(pii(tmpX, tmpY), ed), dis[tmpX][tmpY], pii(tmpX, tmpY) });
                        pre[tmpX][tmpY] = cur;
                    }
                }
            }
        }

        return dis;
}

void solve() {
        cin >> n >> m;

        a = vector<string>(n + 5);
        for (int i = 1; i <= n; i++) {
            cin >> a[i];
            a[i] = " " + a[i];
        }

        pii st = pii(1, 1), ed = pii(n, m);
        auto ans = Astar(st, ed);
        bool ok = (ans[n][m] != INF);
        cout << (ok ? "Yes" : "No") << '\n';

        if (!ok) {
            return;
```

```
    }

    // 输出最短路径长度
    cout << "minDistance = " << ans[n][m] << '\n';

    // 输出最短路径
    auto cur = ed;
    vector<pii> stk;
    while (cur != pii(0, 0)) {
        stk.push_back(cur);
        cur = pre[cur.x][cur.y];
    }
    for (; stk.size(); stk.pop_back()) {
        auto [x, y] = stk.back();
        cout << "(" << x << ", " << y << ")";
        cout << (stk.back() == st ? "\n" : " -> ");
    }
}

int main() {
    cin.tie(0)->sync_with_stdio(false);
    // int t; cin >> t; while (t--)
    solve();
    return 0;
}
```

1.2 迷宫的加载与转换

1.2.1 加载迷宫

实现一个瓦块地图类 TiledMap，用于加载迷宫地图，并得到可通过矩阵 accessibleMatrix。

TiledMap 类继承于 Layer 类，主要的成员变量和成员函数如下。

```
class TiledMap : public cocos2d::Layer {
private:
    TMXTiledMap* tiledMap;
    int mapUnit;
    int mapWidth;
    int mapHeight;

    TMXLayer* wallLayer;   // 墙壁层

    std::vector<std::vector<bool>> accessibleMatrix;   // 可通过矩阵

public:
    …
```

```
    bool init(int mapWidth, int mapHeight, std::string mapPath, int mapUnit = 32, int tag = -1);

    …

    // 将迷宫地图转化为是否可通行的 bool 矩阵 (0-indexed)
    void transformIntoBooleanMatrix();
};
```

其中 init()函数加载瓦片地图，并设置锚点位置为中心点。

```
bool TiledMap::init(int _mapWidth, int _mapHeight, std::string mapPath, int mapUnit, int tag) {
    mapWidth = _mapWidth;
    mapHeight = _mapHeight;

    auto tiledMap = TMXTiledMap::create(mapPath);
    // Size mapSize = tiledMap->getContentSize();
    tiledMap->setAnchorPoint(Point(0.5, 0.5));
    this->addChild(tiledMap);

    wallLayer = tiledMap->getLayer("Wall");

    transformIntoBooleanMatrix();

    return true;
}
```

上述的 wallLayer 为迷宫中墙壁所在层。如**图 1.2.1.1** 所示。

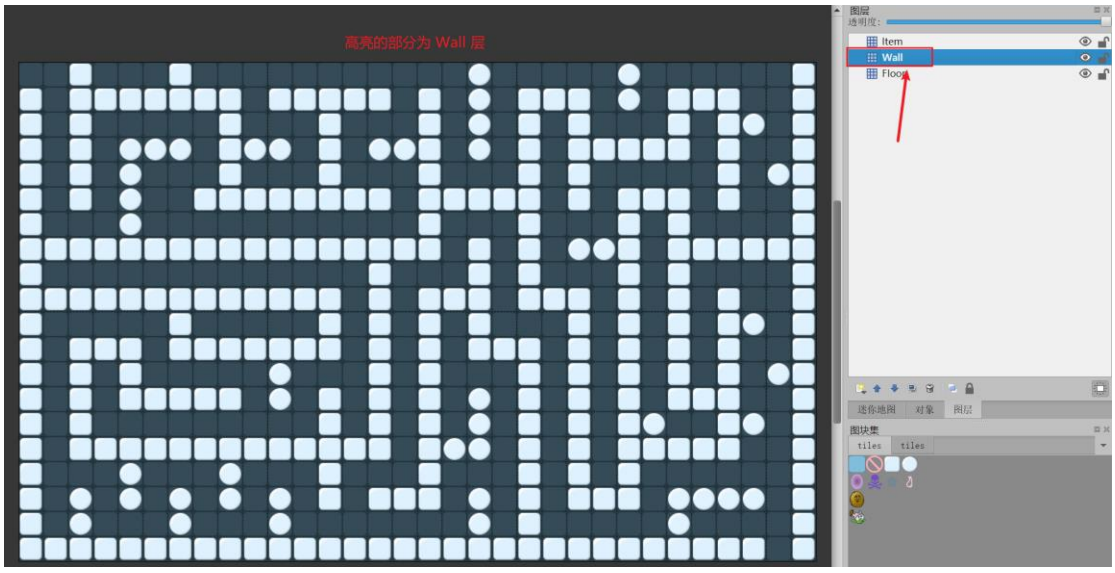


图 1.2.1.1：迷宫地图的 Wall 层

init()函数调用将迷宫地图转化为是否可通行的 bool 矩阵 (0-indexed)的函数 transformIntoBooleanMatrix()，其实现如下。

```
// 将迷宫地图转化为是否可通行的 bool 矩阵 (0-indexed)
void TiledMap::transformIntoBooleanMatrix() {
    accessibleMatrix = std::vector<std::vector<bool>>(mapWidth,
```

```
std::vector<bool>(mapHeight));
    for (int i = 0; i < mapWidth; i++) {
        for (int j = 0; j < mapHeight; j++) {
            int gid = wallLayer->getTileGIDAt(Point(i, j));
            accessibleMatrix[i][j] = (std::find(INACCESSIBLE_TILES.begin(),
INACCESSIBLE_TILES.end(), gid) == INACCESSIBLE_TILES.end());
        }
    }
}
```

Config.h 中的 INACCESSIBLE_TILES 指定为不可通过的 tile 的编号。

```
// 迷宫
...
static const std::vector<int> INACCESSIBLE_TILES = { 3, 4 };
```

注意 Tiled 中的 tile 是 0-indexed，而 Cocos2d-x 中是 1-indexed，即上述的在 Cocos2d-x 中编号 3、4 表示的 tile 如**图 1.2.1.2** 所示。

图 1.2.1.2：在 Cocos2d-x 中编号 3、4 表示的 tile

1.2.2 转换迷宫

Config.h 中指定迷宫相关的参数。

```
// 迷宫
static const int MAP_UNIT = 32;
static const int MAP_WIDTH = 32;
static const int MAP_HEIGHT = 20;
static const std::string TILED_MAZE_1_PATH = "mazes/tiledMaze1.tmx";
static const std::string TILED_MAZE_2_PATH = "mazes/tiledMaze2.tmx";
static const std::string WALL_LAYER_NAME = "Wall";
static const std::vector<int> ACCESSIBLE_TILES = { 1 };
static const std::vector<int> INACCESSIBLE_TILES = { 3, 4 };
```

第一层的场景类 Level1Scene 实现加载迷宫的函数 loadTiledMaze()。

```
// 加载迷宫地图
void Level1Scene::loadTiledMaze(int mapWidth, int mapHeight, Point position, int z, std::string
mapPath, int mapUnit, int tag) {
    tiledMaze = TiledMap::create(mapWidth, mapHeight, mapPath, mapUnit, tag);
    tiledMaze->setPosition(position);
    if (~tag) {
        tiledMaze->setTag(tag);
    }
    this->addChild(tiledMaze, z);
```

```
    // 预处理可用点
    accessibleMatrix = tiledMaze->getAccessibleMatrix();
    for (int i = 0; i < MAP_WIDTH; i++) {
        for (int j = 0; j < MAP_HEIGHT; j++) {
            if (accessibleMatrix[i][j]) {
                availablePoints.insert({ i, j });
            }
        }
    }
}
```

Level1Scene 类实现将屏幕坐标转化为地图坐标（不考虑越界）的函数screenCoordinate2MapCoordinate()和将地图坐标转化为屏幕坐标（对应 Tile 的中心）的函数mapCoordinate2ScreenCoordinate()。

```
// 屏幕坐标转地图坐标
std::pair<int, int> Level1Scene::screenCoordinate2MapCoordinate(Point screenPoint) {
    return screenCoordinate2MapCoordinate(screenPoint.x, screenPoint.y);
}

// 屏幕坐标转地图坐标
std::pair<int, int> Level1Scene::screenCoordinate2MapCoordinate(float screenX, float screenY) {
    int mapX = screenX / MAP_UNIT;
    int mapY = (visibleSize.height - screenY) / MAP_UNIT;
    return std::pair<int, int>(mapX, mapY);
}

// 地图坐标转屏幕坐标 (tile 的中心)
Point Level1Scene::mapCoordinate2ScreenCoordinate(std::pair<int, int> mapPoint) {
    return mapCoordinate2ScreenCoordinate(mapPoint.first, mapPoint.second);
}

// 地图坐标转屏幕坐标 (tile 的中心)
Point Level1Scene::mapCoordinate2ScreenCoordinate(int mapX, int mapY) {
    float screenX = mapX * MAP_UNIT + 0.5 * MAP_UNIT;
    float screenY = visibleSize.height - mapY * MAP_UNIT - 0.5 * MAP_UNIT;
    return Point(screenX, screenY);
}
```

1.3 A*算法类

示例代码给的 A*算法类写得真是太烂了，下面重构一下这个类。

A*算法类 AStar 的主要成员变量和成员函数如下。

```
class AStar {
private:
    const std::vector<int> dx = { -1, -1, 0, 1, 1, 1, 0, -1 };
    const std::vector<int> dy = { 0, 1, 1, 1, 0, -1, -1, -1 };
```

```
    int width;
    int height;
    std::vector<std::vector<bool>> accessibleMatrix;    // 0-indexed

    float minDistance;    // st 到 ed 的最短路长度
    std::vector<std::pair<int, int>> path;    // 最短路

public:
    …

    int getManhattanDistance(std::pair<int, int> st, std::pair<int, int> ed);

    // 检查点 (x, y) 是否能走
    bool checkAccessible(int x, int y);
    bool checkAccessible(std::pair<int, int> p);

    // A* 算法寻路
    void work(std::pair<int, int> st, std::pair<int, int> ed);
};
```

其中 getManhattanDistance()函数用于求迷宫中两点的 Manhattan 距离。

```
int AStar::getManhattanDistance(std::pair<int, int> st, std::pair<int, int> ed) {
    return std::abs(st.first - ed.first) + std::abs(st.second - ed.second);
}
```

checkAccessible()函数用于检查点 (x, y) 是否能走。

```
// 检查点 (x, y) 是否能走
bool AStar::checkAccessible(int x, int y) {
    return 0 <= x && x < width && 0 <= y && y < height && accessibleMatrix[x][y];
}

bool AStar::checkAccessible(std::pair<int, int> p) {
    return checkAccessible(p.first, p.second);
}
```

work()函数实现 A*算法寻路，求得最短路径长度 minDistance 和最短路径 path。若起点与终点不连通，则 minDistance = INF，path 为空。实现与 **1.1.2** 类似，不再赘述。

```
void AStar::work(std::pair<int, int> st, std::pair<int, int> ed) {
    path.clear();
    std::vector<std::vector<float>> dis(width, std::vector<float>(height, INF));
    std::vector<std::vector<std::pair<int, int>>> pre(width, std::vector<std::pair<int, int>>(height));    // 前驱
    dis[st.first][st.second] = 0;
    pre[st.first][st.second] = std::pair<int, int>(0, 0);

    using T = std::tuple<float, float, std::pair<int, int>>;    // { 到终点的估计距离，到起点的实
```

```
际距离, 节点坐标 }
    std::priority_queue<T, std::vector<T>, std::greater<T>> heap;
    heap.push({ getManhattanDistance(st, ed), 0, st });

    // A* 算法
    while (heap.size()) {
        auto [_, real, cur] = heap.top();
        heap.pop();

        auto [curX, curY] = cur;

        // 到达终点
        if (curX == ed.first && curY == ed.second) {
            break;
        }

        for (int i = 0; i < 8; i++) {
            int tmpX = curX + dx[i], tmpY = curY + dy[i];
            if (checkAccessible(tmpX, tmpY)) {
                float distance = (i & 1) ? std::sqrt(2) : 1;   // 斜着走长度规定为 √2
                if (dis[tmpX][tmpY] > real + distance) {
                    dis[tmpX][tmpY] = real + distance;
                    heap.push({ dis[tmpX][tmpY] + getManhattanDistance(std::pair<int,
int>(tmpX, tmpY), ed),
                                dis[tmpX][tmpY], std::pair<int, int>(tmpX, tmpY) });
                    pre[tmpX][tmpY] = cur;
                }
            }
        }
    }

    minDistance = dis[ed.first][ed.second];

    // 不连通
    if (!doubleCompare(minDistance, INF)) {
        return;
    }

    // 求路径
    for (std::pair<int, int> cur = ed; cur != std::pair<int, int>(0, 0); cur =
pre[cur.first][cur.second]) {
        path.push_back(cur);
    }
    std::reverse(path.begin(), path.end());
```

```
}
```

1.4 第一层场景类

1.4.1 Level1Scene 类概览

Level1Scene 类的主要成员变量和成员函数如下。

```
class Level1Scene : public BasicScene {
private:
    // 迷宫地图
    TiledMap* tiledMaze;   // 迷宫地图
    std::vector<std::vector<bool>> accessibleMatrix;   // 可通过矩阵
    std::set<std::pair<int, int>> availablePoints;   // 未占用的点
    std::set<std::pair<int, int>> usedAvailablePoints;   // 已占用的未占用的点

    // 可视化路径
    DrawNode* pathDrawer;   // 路径的绘图节点
    float moveSpeed;   // 移动到(八)相邻节点的时间间隔

    // 角色
    Sprite* cattail;   // 猫尾草
    Sprite* walnut;   // 坚果墙

    // item
    Sprite* forbidden;
    std::vector<Sprite*> portals;
    std::vector<Sprite*> skulls;
    …

    bool canPlace;   // 能否放置坚果墙

    Label* placeTip;
    Label* findTip;
    Label* noWayTip;
    Label* dangerousTip;

    …
public:
    …

    bool init();

    …

    void update(float dt);
```

```
    // 预加载音乐
    void preloadSounds();

    // 键盘监听的回调函数
    void onKeyPressed(EventKeyboard::KeyCode keyCode, Event* event);

    // 触屏开始的回调函数
    bool onTouchBegan(Touch* touch, Event* event);

    // 触屏结束的回调函数
    void onTouchEnded(Touch* touch, Event* event);

    // 加载绘图节点
    void loadPathDrawer(int z);

    // 加载提示信息
    void loadPlaceTip();
    void loadFindTip();
    void loadNoWayTip();
    void loadDangerousTip();

    // 加载纯色背景
    void loadPureBackground();

    // 加载迷宫地图
    void loadTiledMaze(int mapWidth, int mapHeight, Point position, int z, std::string mapPath,
int mapUnit = 32, int tag = -1);

    // 屏幕坐标转地图坐标
    std::pair<int, int> screenCoordinate2MapCoordinate(Point screenPoint);
    std::pair<int, int> screenCoordinate2MapCoordinate(float screenX, float screenY);

    // 地图坐标转屏幕坐标 (tile 的中心)
    Point mapCoordinate2ScreenCoordinate(std::pair<int, int> mapPoint);
    Point mapCoordinate2ScreenCoordinate(int mapX, int mapY);

    …

    // 加载猫尾草
    void loadCattail(Point position, int z, bool visible = true, int tag = -1);

    // 加载坚果墙
    void loadWalnut(Point position, int z, bool visible = true, int tag = -1);
```

```
    // 加载禁止
    void loadForbidden(Point position, int z, bool visible = false, int tag = -1);

    // 加载传送门
    void loadPortal(Point position, int z, bool visible = false, int tag = -1);

    // 加载头颅
    void loadSkull(Point position, int z, bool visible = false, int tag = -1);

    …

    // 随机取一个可用的点
    std::pair<int, int> getRandomAvailablePoint();

    // 占用一个可用的点
    void occupyAvailablePoint(std::pair<int, int> point);

    // 归还一个可用的点
    void returnAvailablePoint(std::pair<int, int> point);

    // 恢复可用的点
    void recoverAvailablePoints();

    // 碰撞检测
    void collisionDetection();

    …
};
```

1.4.2 加载场景

Level1Scene 类的 init()函数加载场景，主要包括：

（1）加载纯色背景和迷宫地图。

（2）加载路径可视化的绘图节点。

（3）在随机一个空位置加载猫尾草，初始时可见。

（4）在可视区域之外加载坚果墙，初始时不可见。

（5）加载禁止放置的符号，初始时不可见。

（6）加载提示信息，初始时不可见。

（7）预加载音乐与音效，并播放 bgm。

（8）添加键盘监听和鼠标监听。

```
bool Level1Scene::init() {
    …

// ---------=  初始化变量  =---------
    …
    pause = false;
```

```
// ---------=  右下角关闭按钮  =---------
    …

// ---------=  加载场景  =---------
    // 加载纯色背景
    loadPureBackground();

    // 加载迷宫地图
    loadTiledMaze(MAP_WIDTH, MAP_HEIGHT, Point(visibleSize.width / 2, visibleSize.height - MAP_HEIGHT * MAP_UNIT / 2), 0, TILED_MAZE_1_PATH);  // 画面中上方

    // 加载绘图节点
    loadPathDrawer(5);

    // 加载猫尾草
    auto cattailPoint = getRandomAvailablePoint();
    Point cattailPosition = mapCoordinate2ScreenCoordinate(cattailPoint);
    loadCattail(cattailPosition, 3);
    occupyAvailablePoint(cattailPoint);

    // 加载坚果墙
    loadWalnut(Point(0, 2 * visibleSize.height), 3, false);

    // 加载禁止
    loadForbidden(Point(0, 0), 5, false);

    // 加载提示信息
    loadPlaceTip();
    loadFindTip();
    loadNoWayTip();
    loadDangerousTip();

    …

    // 初始化变量
    …
    canPlace = true;

// ---------=  音乐与音效  =---------
    preloadSounds();
    playBackgroundMusic(BGM_PATH);
```

```
// ---------=  添加监听  =---------
    addKeyboardListener();   // 添加键盘监听
    addTouchListener();   // 添加触屏监听

// ---------= Update =---------
    this->scheduleUpdate();

    CCLOG("Level1Scene initialized successfully.");

    return true;
}
```

1.4.3 加载绘图节点

loadPathDrawer()函数用于加载绘图节点。

```
// 加载绘图节点
void Level1Scene::loadPathDrawer(int z) {
    pathDrawer = DrawNode::create();
    this->addChild(pathDrawer, z);

    moveSpeed = 0.2;
}
```

1.4.4 加载猫尾草、坚果墙、禁止标志、传送门、头颅

loadCattail()函数用于加载猫尾草。

loadWalnut()函数用于加载坚果墙。

loadForbidden()函数用于加载禁止标志。

loadPortal()函数用于加载传送门。

loadSkull()函数用于加载头颅。

```
// 加载猫尾草
void Level1Scene::loadCattail(Point position, int z, bool visible, int tag) {
    cattail = Sprite::create(CATTAIL_TILE_PATH);
    cattail->setPosition(position);
    cattail->setVisible(visible);
    if (~tag) {
        cattail->setTag(tag);
    }
    this->addChild(cattail, z);
}

// 加载坚果墙
void Level1Scene::loadWalnut(Point position, int z, bool visible, int tag) {
    walnut = Sprite::create(WALNUT_TILE_PATH);
    walnut->setPosition(position);
    walnut->setVisible(visible);
    if (~tag) {
        walnut->setTag(tag);
```

```
        }
        this->addChild(walnut, z);
}

// 加载禁止
void Level1Scene::loadForbidden(Point position, int z, bool visible, int tag) {
        forbidden = Sprite::create(FORBIDDEN_TILE_PATH);
        forbidden->setPosition(position);
        forbidden->setVisible(visible);
        if (~tag) {
                forbidden->setTag(tag);
        }
        this->addChild(forbidden, z);
}

// 加载传送门
void Level1Scene::loadPortal(Point position, int z, bool visible, int tag) {
        auto portal = Sprite::create(PORTAL_TILE_PATH);
        portal->setPosition(position);
        portal->setVisible(visible);
        if (~tag) {
                portal->setTag(tag);
        }
        this->addChild(portal, z);

        portals.push_back(portal);
}

// 加载头颅
void Level1Scene::loadSkull(Point position, int z, bool visible, int tag) {
        auto skull = Sprite::create(SKULL_TILE_PATH);
        skull->setPosition(position);
        skull->setVisible(visible);
        if (~tag) {
                skull->setTag(tag);
        }
        this->addChild(skull, z);

        skulls.push_back(skull);
}
```

1.4.5 加载提示信息

Config.h 指定了提示信息的文本。

```
// 提示信息
static const std::string PLACE_TIP = "Click to place the walnut!";
```

```
static const std::string FIND_TIP = "Found the walnut!";
static const std::string NO_WAY_TIP = "No way!";
static const std::string DANGEROUS_TIP = "Dangerous! Press R to restart!";
```

Level1Scene 类实现了几个函数用于加载提示信息，包括：

（1）loadPlaceTip()函数加载 PLACE_TIP，颜色为黄色。

（2）loadFindTip()函数加载 FIND_TIP，颜色为绿色。

（3）loadNoWayTip()函数加载 NO_WAY_TIP，颜色为栏色。

（4）loadDangerousTip()函数加载 DANGEROUS_TIP，颜色为红色。

```
void Level1Scene::loadPlaceTip() {
    placeTip = Label::createWithBMFont(TIPS_FNT_PATH, PLACE_TIP);
    placeTip->setPosition(Point(
        placeTip->getContentSize().width / 2 + 20,
        (visibleSize.height - MAP_HEIGHT * MAP_UNIT) / 2
    ));
    placeTip->setColor(Color3B::YELLOW);

    auto actionSequence = Sequence::create(
        FadeOut::create(1),
        FadeIn::create(1),
        nullptr
    );
    auto action = RepeatForever::create(actionSequence);
    placeTip->runAction(action);

    this->addChild(placeTip, 5);
}

void Level1Scene::loadFindTip() {
    findTip = Label::createWithBMFont(TIPS_FNT_PATH, FIND_TIP);
    findTip->setPosition(Point(
        findTip->getContentSize().width / 2 + 20,
        (visibleSize.height - MAP_HEIGHT * MAP_UNIT) / 2
    ));
    findTip->setColor(Color3B::GREEN);
    findTip->setVisible(false);

    this->addChild(findTip, 5);
}

void Level1Scene::loadNoWayTip() {
    noWayTip = Label::createWithBMFont(TIPS_FNT_PATH, NO_WAY_TIP);
    noWayTip->setPosition(Point(
        noWayTip->getContentSize().width / 2 + 20,
        (visibleSize.height - MAP_HEIGHT * MAP_UNIT) / 2
```

```
    ));
    noWayTip->setColor(Color3B::BLUE);
    noWayTip->setVisible(false);

    this->addChild(noWayTip, 5);
}

void Level1Scene::loadDangerousTip() {
    dangerousTip = Label::createWithBMFont(TIPS_FNT_PATH, DANGEROUS_TIP);
    dangerousTip->setPosition(Point(
        dangerousTip->getContentSize().width / 2 + 20,
        (visibleSize.height - MAP_HEIGHT * MAP_UNIT) / 2
    ));
    dangerousTip->setColor(Color3B::RED);
    dangerousTip->setVisible(false);

    this->addChild(dangerousTip, 5);
}
```

1.4.6 添加触屏监听

Level1Scene 类中，触屏事件的回调函数：

（1）onTouchBegan()函数将触屏点坐标转化为地图坐标，并检查是否越界。若越界，则在对应位置显示禁止图标；否则隐藏提示信息，放置坚果墙，并用 A*算法寻路。

（2）onTouchEnded()函数在触屏结束后重置变量 canPlace。

```
// 触屏开始的回调函数
bool Level1Scene::onTouchBegan(Touch* touch, Event* event) {
    // 屏幕坐标转地图坐标
    auto [touchX, touchY] = screenCoordinate2MapCoordinate(touch->getLocation());
    // touchingPoint = Point(touchX, touchY);

    if (!canPlace || touchX < 0 || touchX >= MAP_WIDTH || touchY < 0 || touchY >= MAP_HEIGHT || !accessibleMatrix[touchX][touchY]) {  // 不可放置或越界或不可到达
        forbidden->setPosition(mapCoordinate2ScreenCoordinate(touchX, touchY));
        forbidden->setVisible(true);
    }
    else if (screenCoordinate2MapCoordinate(cattail->getPosition()) == std::pair<int, int>(touchX, touchY)) {  // 坚果墙与猫尾草重合
        forbidden->setPosition(mapCoordinate2ScreenCoordinate(touchX, touchY));
        forbidden->setVisible(true);
    }
    else {  // 成功放置
        // 隐藏提示信息
        placeTip->setVisible(false);
        findTip->setVisible(false);
        noWayTip->setVisible(false);
```

```
            dangerousTip->setVisible(false);

            // 放置坚果墙
            canPlace = false;
            walnutInDangerous = false;
            walnut->setPosition(mapCoordinate2ScreenCoordinate(touchX, touchY));
            walnut->setVisible(true);

            // 寻路
            std::pair<int, int> st = screenCoordinate2MapCoordinate(cattail->getPosition());
            std::pair<int, int> ed = screenCoordinate2MapCoordinate(walnut->getPosition());
            AStar solver(MAP_WIDTH, MAP_HEIGHT, accessibleMatrix, st, ed);

            if (doubleCompare(solver.getMinDistance(), INF / 2) >= 0) {   // 最短距离 >= INF / 2, 不连通
                placeTip->setVisible(false);
                noWayTip->setVisible(true);

                canPlace = true;
            }
            else {   // 连通, 显示路径
                Vector<FiniteTimeAction*> actionVector;   // 动作序列

                auto path = solver.getPath();
                auto color = COLORS[generateRandomInteger(0, (int)COLORS.size() - 1)];
                for (int i = 1; i < path.size(); i++) {
                    auto currentPoint = mapCoordinate2ScreenCoordinate(path[i]);
                    pathDrawer->drawLine(mapCoordinate2ScreenCoordinate(path[i - 1]), currentPoint, color);

                    auto moveAction = MoveTo::create(moveSpeed, currentPoint);
                    actionVector.pushBack(moveAction);
                }

                if (actionVector.empty()) {
                    return true;
                }

                auto actionSequence = Sequence::create(actionVector);
                cattail->runAction(actionSequence);
            }
        }

        return true;
```

```
}

// 触屏结束的回调函数
void Level1Scene::onTouchEnded(Touch* touch, Event* event) {
    forbidden->setVisible(false);

    // canPlace = true;
}
```

1.4.7 猫尾草与坚果墙的碰撞检测

Level1Scene 类的 update()函数调用碰撞检测函数 collisionDetection()进行碰撞检测。若猫尾草和坚果墙所在的屏幕坐标转化为地图坐标后在同一位置，则认为碰撞，显示对应提示信息并播放对应音效。

```
// 碰撞检测
void Level1Scene::collisionDetection() {
    …

    // 猫尾草与坚果墙
    if (!walnutInDangerous && screenCoordinate2MapCoordinate(cattail->getPosition()) == screenCoordinate2MapCoordinate(walnut->getPosition())) {
        findTip->setVisible(true);

        walnut->setPosition(Point(0, 2 * visibleSize.height));
        walnut->setVisible(false);

        canPlace = true;

        playSoundEffect(FIND_SOUND_PATH);
    }

    …
}
```

1.5 编译运行

成功运行游戏，如**图 1.5.1** 所示。

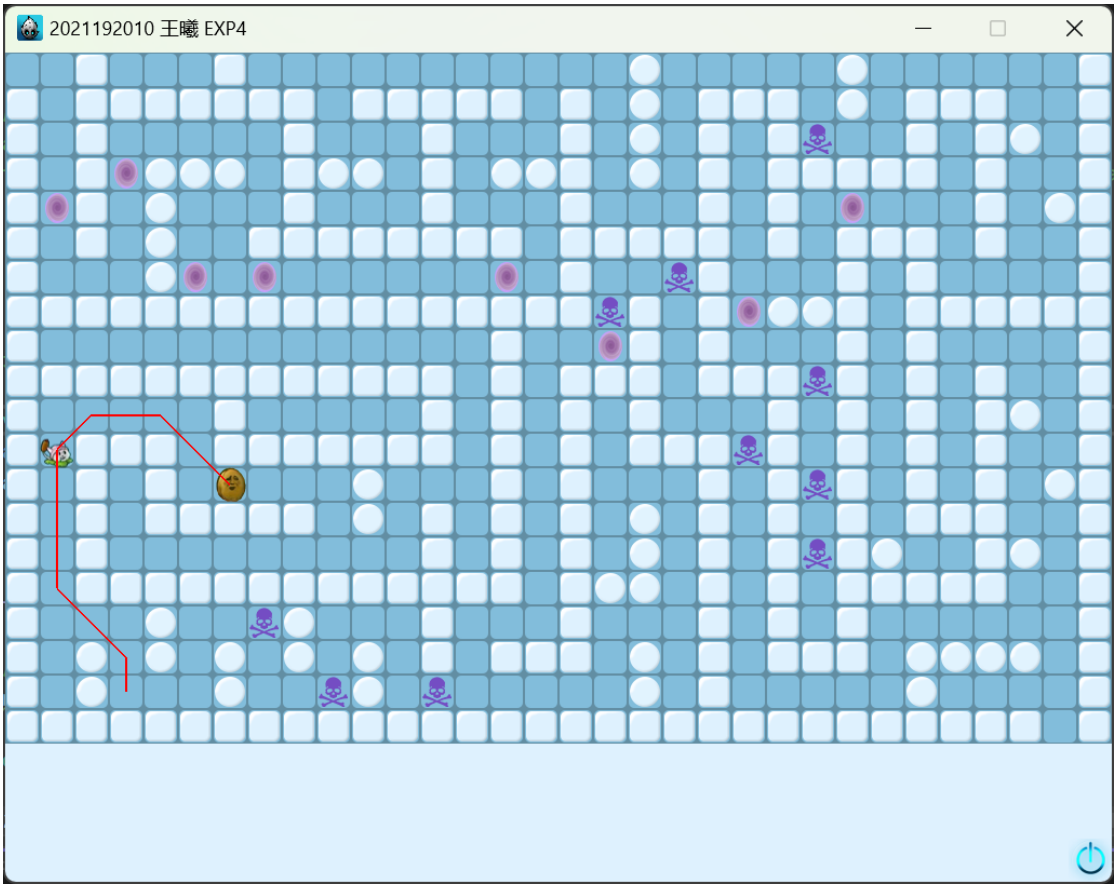


图 1.5.1：成功运行游戏

2. 修改窗口大小

在 AppDelegate.cpp 中修改 designResolutionSize 为 1024 x 768。

```
// static cocos2d::Size designResolutionSize = cocos2d::Size(480, 320);
static cocos2d::Size designResolutionSize = cocos2d::Size(1024, 768);
```

其中 1024 = 32 * 32，768 > 20 * 32，其中 32 x 20 是地图的大小，32 x 32 是 Tile 的大小。

修改后，窗口可完整地显示地图。如**图 2.1** 所示。

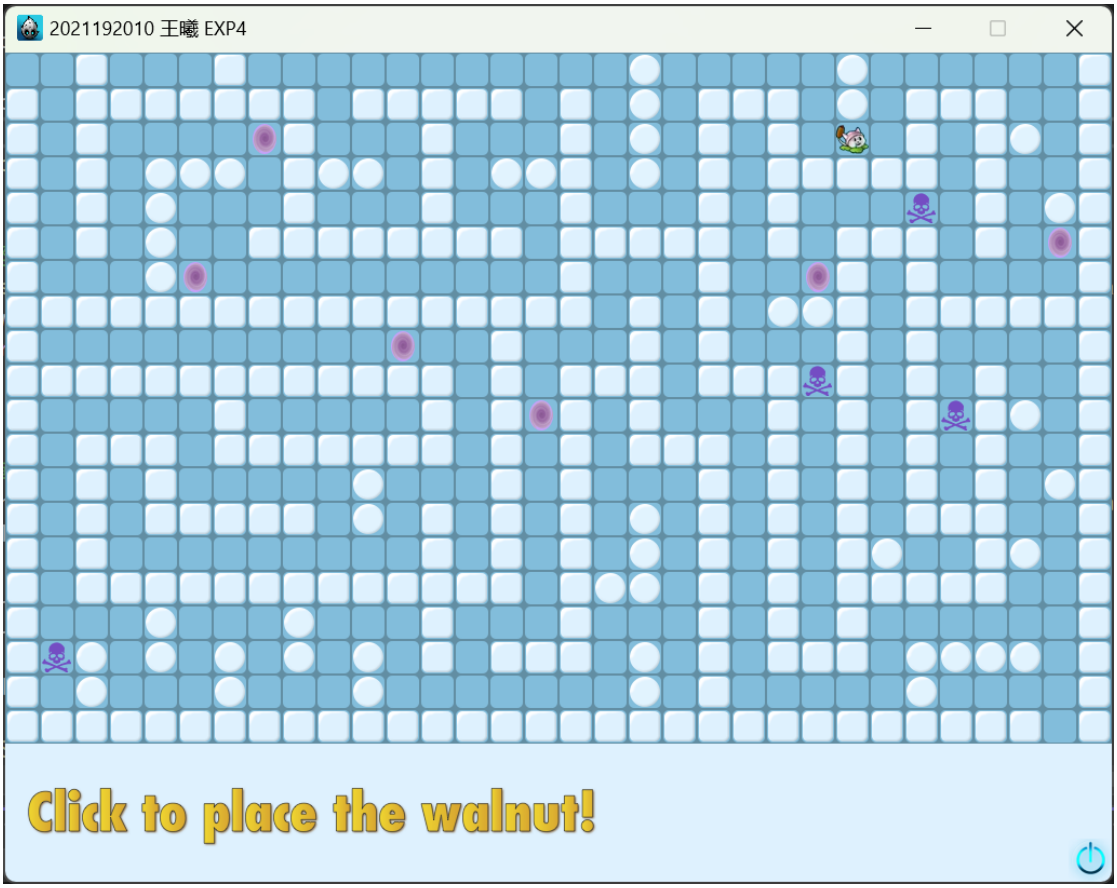


图 2.1：窗口可完整地显示地图

3. 增加 Dangerous 区域

本次实验中 Dangerous 区域用紫色的头颅表示。

1.4.4 中已实现加载头颅。

在 Level1Scene 类的 collisionDetection()函数中加入猫尾草与头颅的碰撞检测，若碰撞，则停止猫尾草的移动，并显示对应的提示信息和播放对应的音效。

因坚果墙可能位于头颅位置，本次实验中规定优先判定猫尾草与头颅的碰撞，即认为 dangerous。为此，猫尾草与头颅的碰撞检测应在猫尾草与坚果墙的碰撞检测之前。

```
// 碰撞检测
void Level1Scene::collisionDetection() {
    // 猫尾草与头颅 (先判断危险)
    for (auto skull : skulls) {
        if (screenCoordinate2MapCoordinate(cattail->getPosition()) == screenCoordinate2MapCoordinate(skull->getPosition())) {
            dangerousTip->setVisible(true);

            cattail->stopAllActions();

            canPlace = false;

            playSoundEffect(DANGEROUS_SOUND_PATH);
        }

        …
    }

    // 猫尾草与坚果墙
    …
}
```

运行，猫尾草碰到头颅时如**图 3.1** 所示。

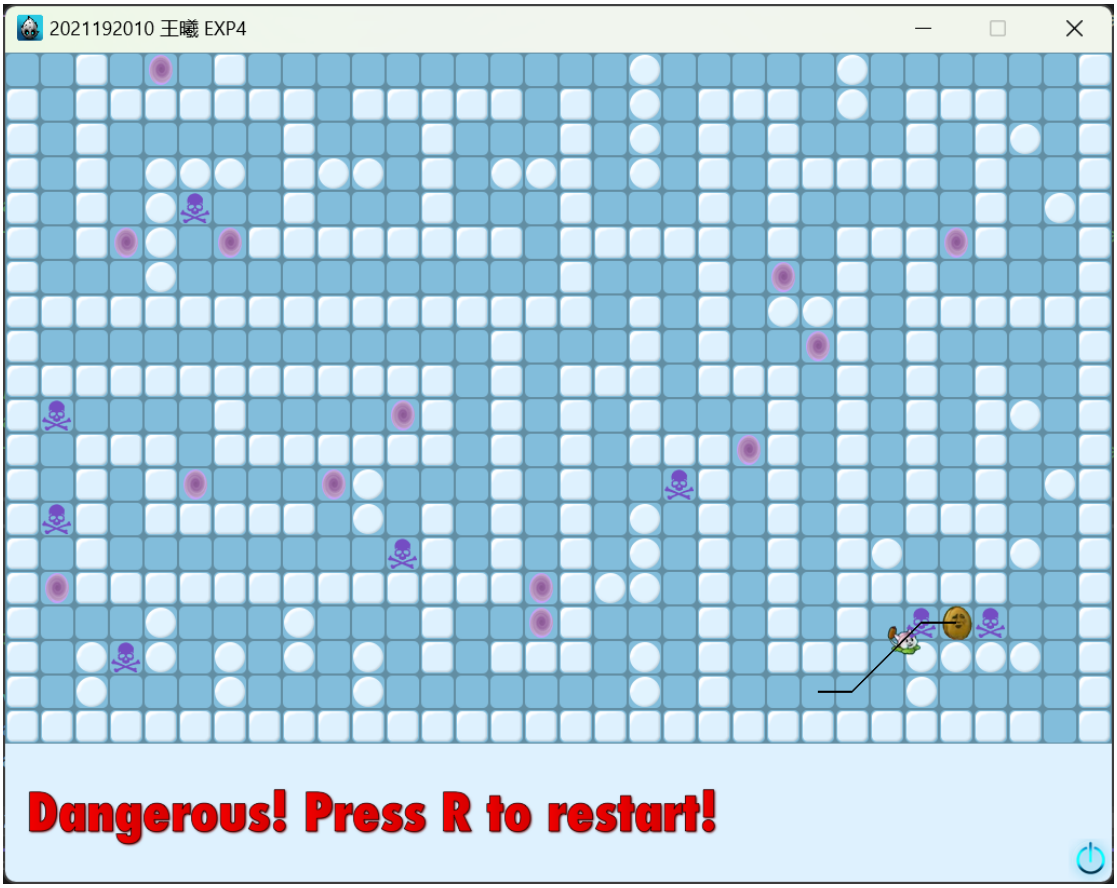


图 3.1：猫尾草碰到头颅

4. 增加按键监听

Level1Scene 类的父类 BasicScene 类已实现添加键盘监听的函数 addKeyboardListener()。

Level1Scene 类重写父类的键盘监听的回调函数 onKeyPressed()，有如下功能：

（1）ESC 键：关闭游戏。

（2）R 键：重玩本层。

```
// 键盘监听的回调函数
void Level1Scene::onKeyPressed(EventKeyboard::KeyCode keyCode, Event* event) {
    switch (keyCode) {
        case EventKeyboard::KeyCode::KEY_ESCAPE: {    // ESC: 关闭
            Director::getInstance()->end();
            break;
        }
        case EventKeyboard::KeyCode::KEY_R: {    // R: 重玩
            Director::getInstance()->replaceScene(Level1Scene::createScene());
            break;
        }
        …
    }
}
```

5. 增加地图层次

5.1 第二层地图绘制

在 Tiled 中用**实验 1** 的方式绘制第二层的地图。如**图 5.1.1** 所示。

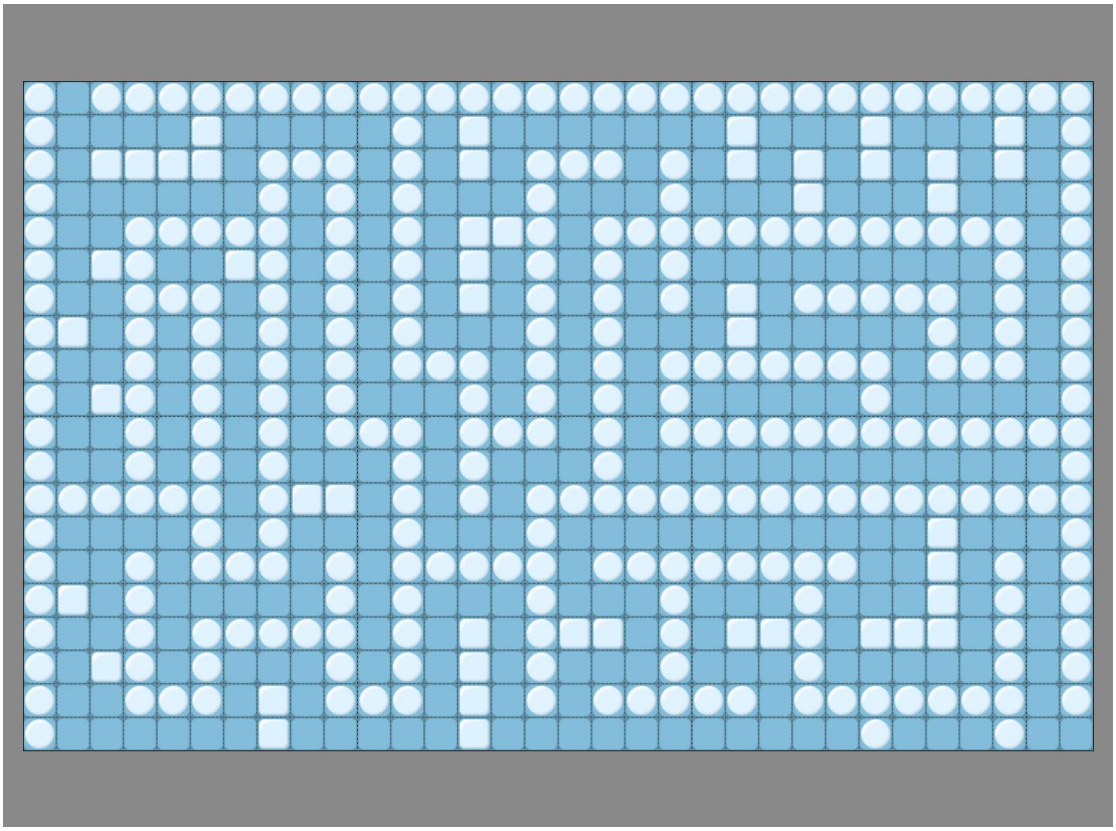


图 5.1.1：绘制第二层的地图

将地图导出为 tiledMaze2.tmx，并将地图文件复制到 Resources 文件夹中。注意要带上图块集的.tsx 文件。如**图 5.1.2** 所示。

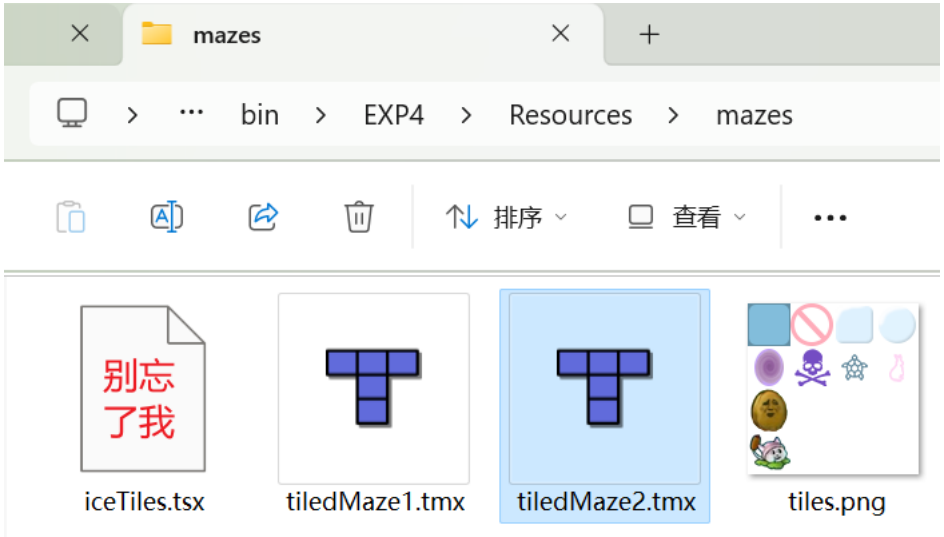


图 5.1.2：将地图文件复制到 Resources 文件夹中

5.2 第二层场景加载

将 Level1Scene.h 和 Level1Scene.cpp 分别复制一份，分别命名为 Level2Scene.h 和

Level2Scene.cpp，并将其加入 Visual Studio 中。如**图 5.2.1** 所示。

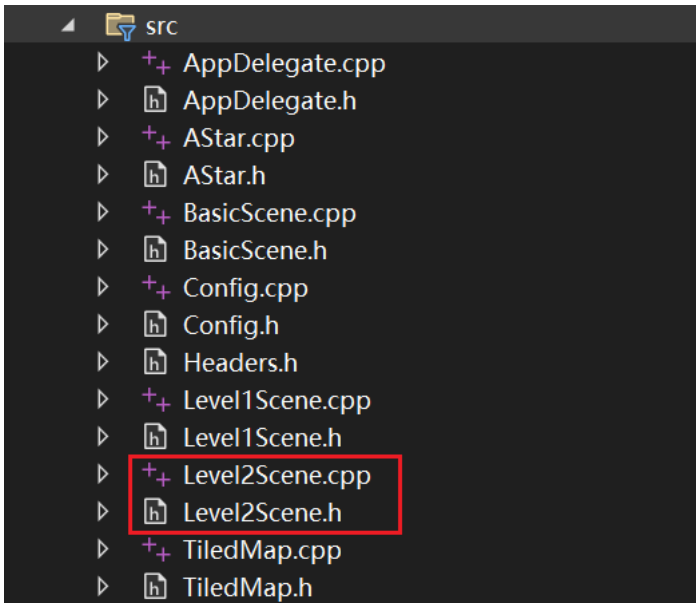


图 5.2.1：第二层场景的源文件

在 Level2Scene 类的 init()函数中加载相应的背景。

运行，第二层的场景如**图 5.2.2** 所示。

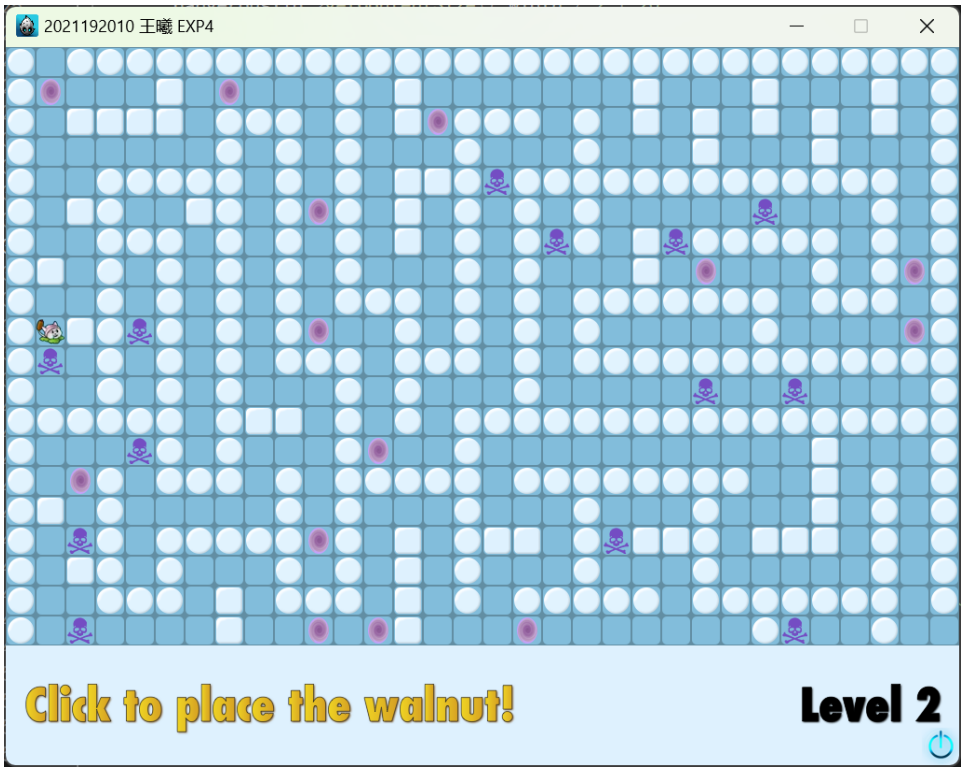


图 5.2.2：第二层的场景

在Level1Scene类的collisionDetection()函数中加入猫尾草与传送门的碰撞检测，若碰撞，则切换到第二层场景。

```
// 碰撞检测
void Level1Scene::collisionDetection() {
    …
```

```
    // 猫尾草与传送门
    for (auto portal : portals) {
        if (screenCoordinate2MapCoordinate(cattail->getPosition()) == screenCoordinate2MapCoordinate(portal->getPosition())) {
            Director::getInstance()->replaceScene(Level2Scene::createScene());
        }
    }

    …
}
```

同理，在第二层的场景中，猫尾草与传送门碰撞时切换到第一层场景。

6. 增加音乐

6.1 BasicScene 类的音乐函数

本次实验沿用**实验 3** 的 BasicScene 类，其中实现了预加载音效 bgm 和播放音效/bgm 的函数。

```
// 预加载音效
void BasicScene::preloadSoundEffect(std::string soundEffectPath) {
    SimpleAudioEngine::getInstance()->preloadEffect(soundEffectPath.c_str());
}

// 预加载背景音乐
void BasicScene::preloadBackgroundMusic(std::string backgroundMusicPath) {

SimpleAudioEngine::getInstance()->preloadBackgroundMusic(backgroundMusicPath.c_str());
}

// 播放音效
void BasicScene::playSoundEffect(std::string soundEffectPath) {
    SimpleAudioEngine::getInstance()->playEffect(soundEffectPath.c_str());
}

// 播放背景音乐
void BasicScene::playBackgroundMusic(std::string backgroundMusicPath) {
    SimpleAudioEngine::getInstance()->playBackgroundMusic(backgroundMusicPath.c_str());
}
```

6.2 音乐与音效设计

本次实验用到如下音效：

（1）bgm：《春雪》- 周深。

（2）猫尾草找到坚果墙的音效：阙英华（不知道是哪个游戏里的角色）的笑声。

（3）猫尾草碰到头颅的音效：阙英华死掉的声音。

6.3 加载音乐与音效

Level1Scene 类重写父类 BasicScene 的 proloadSounds()函数，用于预加载本场景的音乐与音效。

```
// 预加载音乐
void Level1Scene::preloadSounds() {
    preloadBackgroundMusic(BGM_PATH);
    preloadSoundEffect(FIND_SOUND_PATH);
    preloadSoundEffect(DANGEROUS_SOUND_PATH);
    …
}
```

Level1Scene 类的 init()函数中预加载本场景用到的音乐和音效，并播放 bgm。

```
bool Level2Scene::init() {
```

```
        …

// ----------=  音乐与音效  =----------
        preloadSounds();
        playBackgroundMusic(BGM_PATH);

        …
}
```

7. 修复游戏 Bug

7.1 猫尾草与坚果墙重合

猫尾草与坚果墙重合，即玩家将坚果墙放到猫尾草位置时，会同时出现“Found the walnut!”和“No way!”两个提示信息。如**图 7.1.1** 所示。

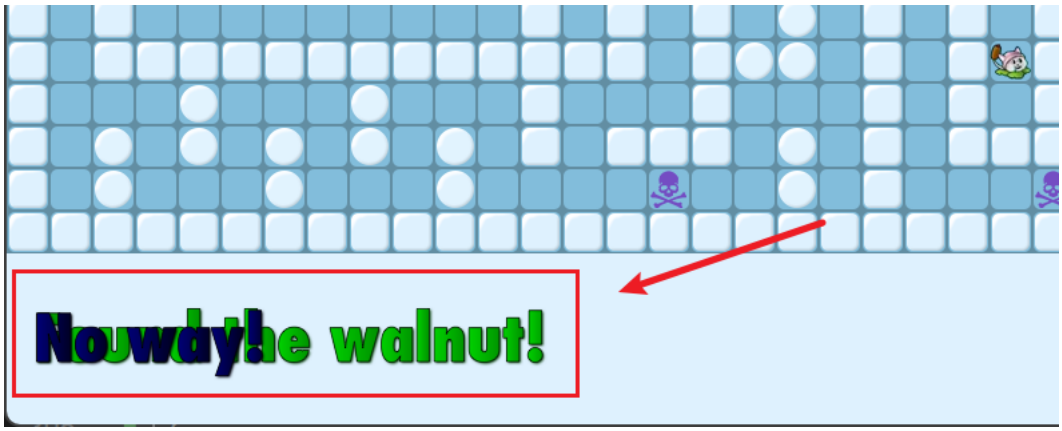


图 7.1.1：猫尾草与坚果墙重合时的双重提示信息

在 Level1Scene 类的 onTouchBegan()函数中特判放置位置与猫尾草重合的情况，规定此时禁止放置。此时玩家尝试将坚果墙放到猫尾草位置时会显示禁止放置，如**图 7.1.2** 所示。

图 7.1.2：玩家尝试将坚果墙放到猫尾草位置时会显示禁止放置

7.2 坚果墙与头颅重合

玩家将坚果墙放在头颅位置，猫尾草寻路碰到坚果墙时，会同时判定“Found the walnut!”和“Dangerous! Press R to restart!”。如**图 7.2.1** 所示。

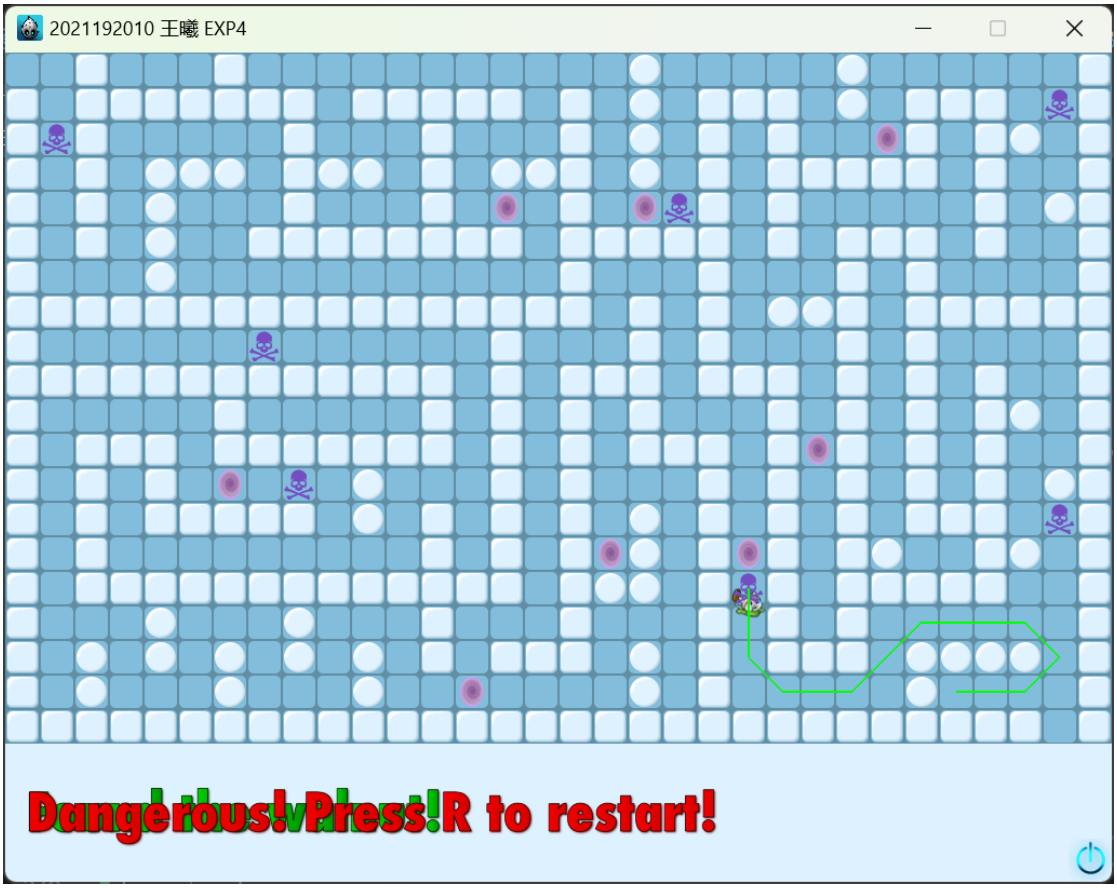


图 7.2.1：将坚果墙放在头颅位置，猫尾草寻路成功时的双重提示信息

在 Level1Scene 中加入一个成员变量 walnutInDangerous，表示坚果墙是否在危险位置，并在 onTouchBegan()函数中成功放置坚果墙时初始化。

```
// 触屏开始的回调函数
bool Level1Scene::onTouchBegan(Touch* touch, Event* event) {
    …

    if (!canPlace || touchX < 0 || touchX >= MAP_WIDTH || touchY < 0 || touchY >= MAP_HEIGHT || !accessibleMatrix[touchX][touchY]) {  // 不可放置或越界或不可到达
        …
    }
    else if (screenCoordinate2MapCoordinate(cattail->getPosition()) == std::pair<int, int>(touchX, touchY)) {  // 坚果墙与猫尾草重合
        …
    }
    else {  // 成功放置
        …

        // 放置坚果墙
        …
        walnutInDangerous = false;
```

```
            …
        }

        return true;
}
```

在 collisionDetection()函数的猫尾草与头颅的碰撞检测中，判定坚果墙是否在头颅位置，若是，则将 walnutInDangerous 置为 true 并 return，不再往下执行。

可能会有如下情况：恰好执行完猫尾草与头颅的碰撞检测后，猫尾草碰到坚果墙，此时仍会出现该 bug。为解决该问题，在猫尾草与坚果墙的碰撞检测的条件中加入!walnutInDangerous。

```
// 碰撞检测
void Level1Scene::collisionDetection() {
    // 猫尾草与头颅 (先判断危险)
    for (auto skull : skulls) {
        …

        // 坚果墙在头颅位置
        if (screenCoordinate2MapCoordinate(skull->getPosition()) == screenCoordinate2MapCoordinate(walnut->getPosition())) {
            walnutInDangerous = true;
            return;
        }
    }

    // 猫尾草与坚果墙
    if (!walnutInDangerous && screenCoordinate2MapCoordinate(cattail->getPosition()) == screenCoordinate2MapCoordinate(walnut->getPosition())) {
        …
    }

    …
}
```

8. 增加粒子效果

8.1 开始场景

开始场景类 StartScene 继承于场景基类 BasicScene，其主要的成员变量和成员函数如下。

```
class StartScene : public BasicScene {
private:
    ControlButton* playButton;
    CCParticleSystemQuad* particleSystem;   // 粒子系统

public:
    …

    bool init();

    …

    void update(float dt);

    void loadPlayButton(Point position, int z, int tag = -1);

    void playButtonOnClick(Ref* pSender, extension::Control::EventType event);

    void loadParticle();
};
```

StartScene 类的 init()函数调用父类的 loadImage()函数加载背景图片，并调用 loadPlayButton()函数加载按钮。

```
bool StartScene::init() {
    ...

    // ---------=  加载场景  =---------
    loadImage(START_BACKGROUND_PATH, Point(visibleSize.width / 2, visibleSize.height / 2), 0);

    loadPlayButton(Point(visibleSize.width / 2, 100 + 300 / 2), 5);

    …

    return true;
}
```

loadPlayButton()函数加载按钮，并为其添加闪烁效果，并添加监听。按钮的回调函数为 playButtonOnClick()。

```
void StartScene::loadPlayButton(Point position, int z, int tag) {
    ui::Scale9Sprite* buttonImage = ui::Scale9Sprite::create(PLAY_BUTTON_PATH);
    playButton = ControlButton::create(buttonImage);
```

```
    playButton->setBackgroundSpriteForState(buttonImage, Control::State::NORMAL);
    playButton->setAdjustBackgroundImage(false);
    playButton->setPosition(position);
    if (~tag) {
        playButton->setTag(tag);
    }
    this->addChild(playButton, z);

    // 闪烁
    auto actionSequence = Sequence::create(
        FadeOut::create(1),
        FadeIn::create(1),
        nullptr
    );
    auto action = RepeatForever::create(actionSequence);
    playButton->runAction(action);

    // 添加监听
    playButton->addTargetWithActionForControlEvents(
        this,
        cccontrol_selector(StartScene::playButtonOnClick),
        extension::Control::EventType::TOUCH_DOWN);
}

void StartScene::playButtonOnClick(Ref* pSender, extension::Control::EventType event) {
    loadParticle();

    pause = false;
}
```

运行，开始场景如**图 8.1.1** 所示。

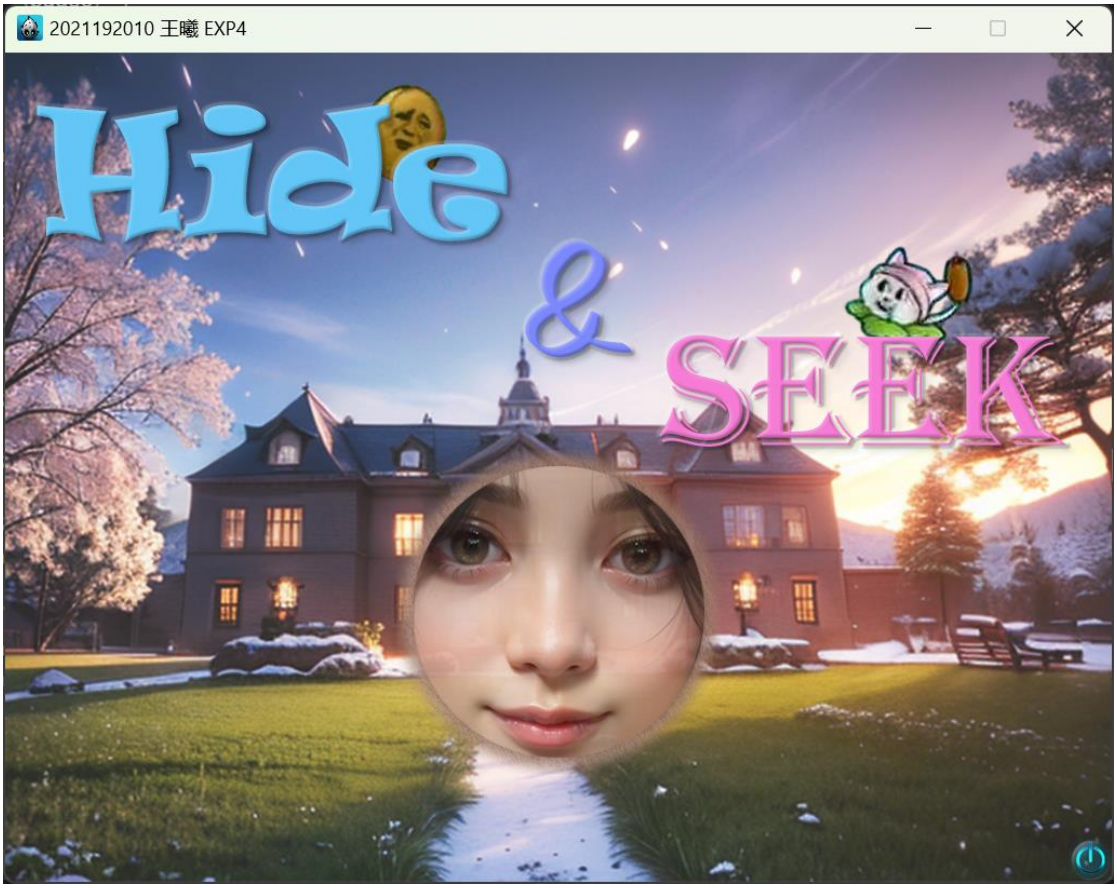


图 8.1.1：开始场景

8.2 粒子效果

8.2.1 加载粒子效果

StartScene 类的 loadParticle()函数加载粒子系统。

```
void StartScene::loadParticle() {
    particleSystem = CCParticleSystemQuad::create(PARTICLE_PATH);

    // particleSystem->setDuration(2);    // 持续 2 s
    particleSystem->setAutoRemoveOnFinish(true);    // 完成后自动移除
    this->addChild(particleSystem, 2);
}
```

8.2.2 按钮发射效果

实现按钮点击后，在按钮的中偏下处播放向下喷射的粒子效果，并让按钮像火箭一样向上发射。待按钮飞出可视区域后，切换到第一层的场景开始游戏。

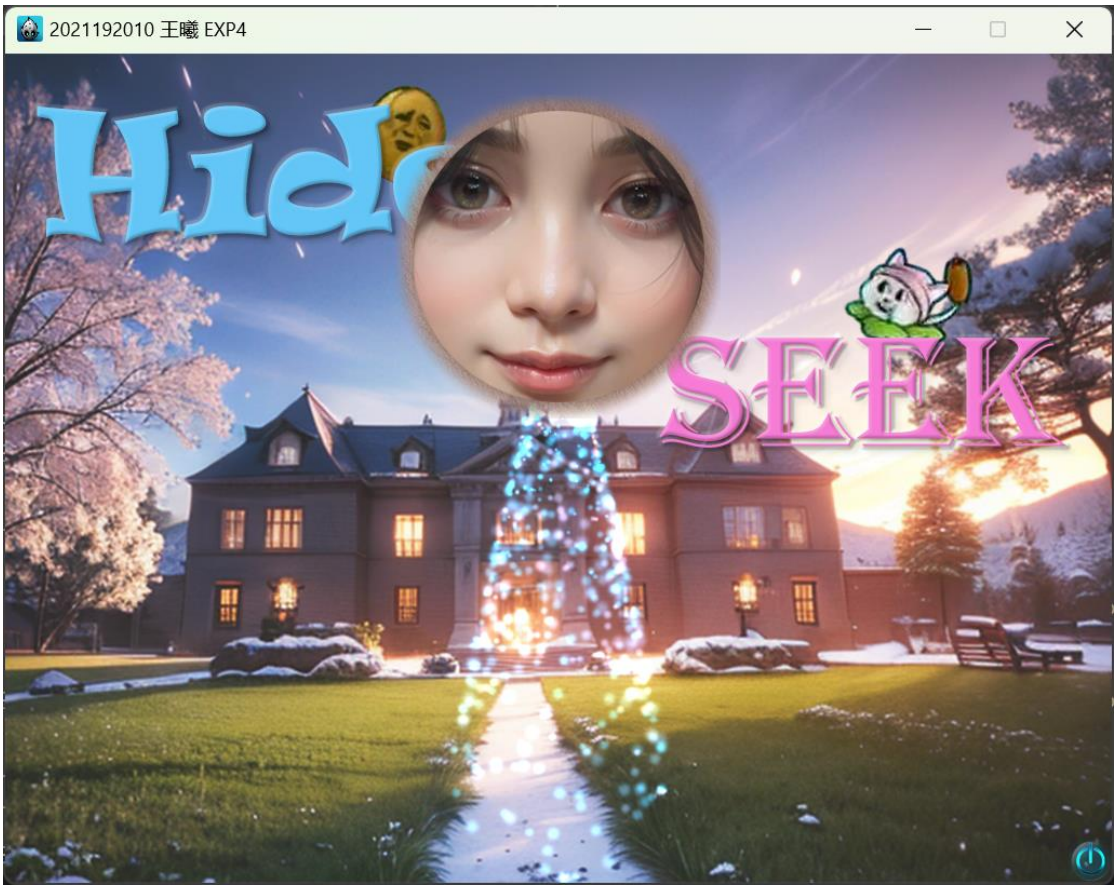
在 StartScene 类的 update()函数中实现上述功能。

```
void StartScene::update(float dt) {
    if (pause) {
        return;
    }

    playButton->setPositionY(playButton->getPositionY() + PLAY_BUTTON_FLY_SPEED);
```

```
    particleSystem->setPosition(Point(playButton->getPositionX(), playButton->getPositionY() - 50));

    // 开始按钮飞出屏幕
    if (doubleCompare(playButton->getPositionY(), visibleSize.height + playButton->getContentSize().height) >= 0) {
        Director::getInstance()->replaceScene(Level1Scene::createScene());
    }
}
```

运行，玩家点击按钮后，按钮起飞。

录屏看起来像按钮是用鼻涕喷上去的，有一种武器 A 的感觉，总之就是有点掉 SAN。正常人类，按理来说，不应该能想到按钮是这么用的。（这一段划掉）



9. 游戏优化

9.1 策划

9.1.1 概述

游戏《Hide&Seek》的主要内容是猫尾草与坚果墙的捉迷藏。

玩家在迷宫中合法的位置放置坚果墙。有如下两种情况：

（1） 若猫尾草与坚果墙连通，则猫尾草会用 A*算法寻路，找到坚果墙，并可视化路径。找到坚果墙后，显示提示信息“Found the walnut!”，并播放笑声音效。

（2） 若猫尾草与坚果墙不连通，则显示提示信息“No way!”。

迷宫地图除可通过的 Tile 和不可通过的 Tile 外，还有如下两种 Tile：

（1） 头颅：危险区域，猫尾草进入该 Tile 会停止，并显示提示信息“Dangerous! Press R to restart!”。

（2） 传送门：切换一二层场景，猫尾草进入该 Tile 会切换场景。

9.1.2 音乐设计

《Hide&Seek》用到了如下音乐：

（1） bgm：《春雪》- 周深。

（2） 猫尾草找到坚果墙的音效：阚英华（不知道是哪个游戏里的角色）的笑声。

（3） 猫尾草碰到头颅的音效：阚英华死掉的声音。

9.1.3 场景设计

灵感来源于 bgm《春雪》中的歌词“我想在冬夜雪地与你追逐”，故将原游戏《迷宫寻宝》的场景改为冰雪风格，并将游戏主题改为 Hide&Seek。

主场景如**图 9.1.3.1** 所示。

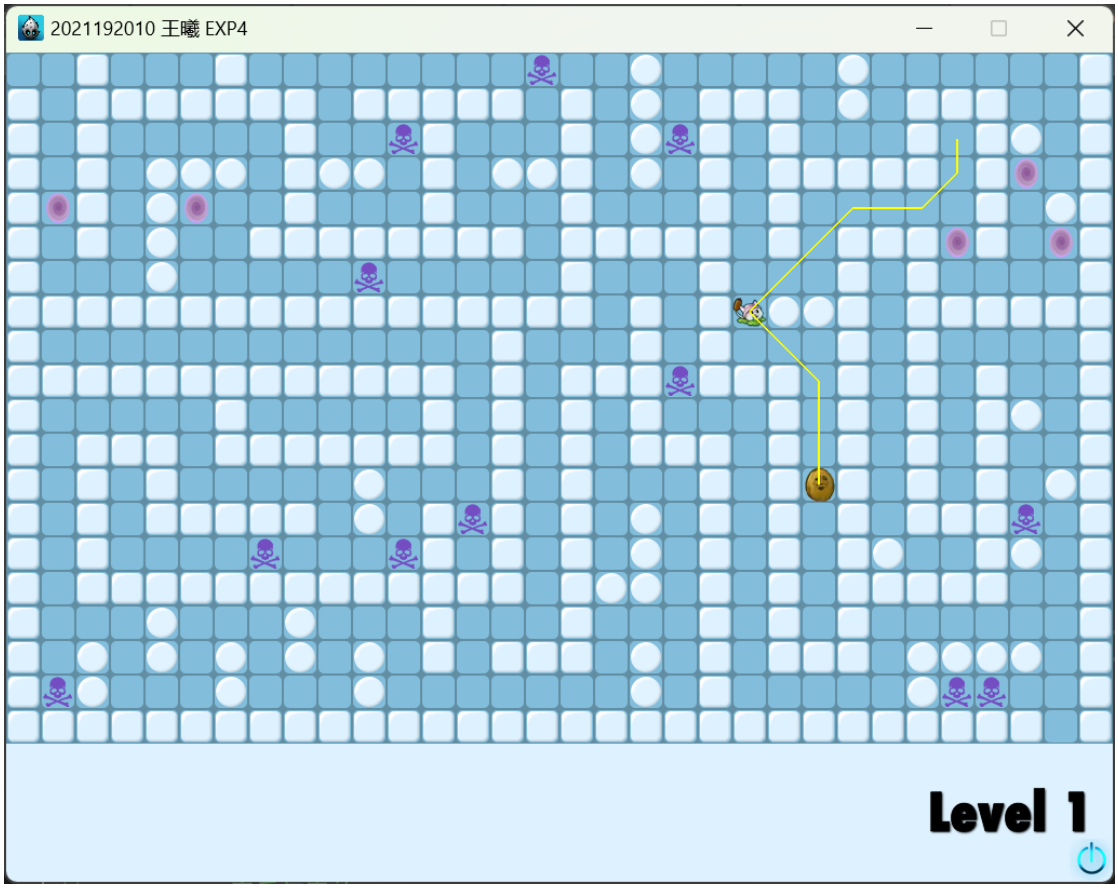


图 9.1.3.1：《Hide&Seek》的主场景

为《Hide&Seek》搭配一个雪地风格的欢迎界面。如**图 9.1.3.2** 所示。

图 9.1.3.2：《Hide&Seek》的欢迎界面（背景）

欢迎界面的底图用 Stable Diffusion 绘制，Prompt 和参数如下。

(masterpiece:1,2),best quality,highres,original,extremely detailed wallpaper,perfect lighting,(extremely detailed CG:1.2), lawn,in winter,(day),snowflakes,sky,no_humans,bright, Negative prompt: (NSFW:1.5),(worst quality:2),(low quality:2),(normal quality:2),lowres,normal quality,((monochrome)),((grayscale)),skin spots,acnes,skin blemishes,age spot,(ugly:1.331),(duplicate:1.331),(morbid:1.21),(mutilated:1.21),(tranny:1.331),mutated hands,(poorly drawn hands:1.5),blurry,(bad anatomy:1.21),(bad proportions:1.331),extra limbs,(disfigured:1.331),(missing arms:1.331),(extra legs:1.331),(fused fingers:1.61051),(too many fingers:1.61051),(unclear eyes:1.331),lowers,bad hands,missing fingers,extra digit,bad hands,missing fingers,(((extra arms and legs))), Steps: 20, Sampler: DPM++ 2M Karras, CFG scale: 7, Seed: 337774986, Size: 768x512, Model hash: 0bb33c7041, Model: 2.5D - Guofeng3 V32, Clip skip: 2, Version: v1.7.0

9.1.4 胜负判定

《Hide&Seek》无胜负机制。

9.2 显示层数

《Hide&Seek》的第一层场景与第二层场景风格统一、观感相似，玩家难以区分。如**图 9.2.1** 和**图 9.2.2** 所示。

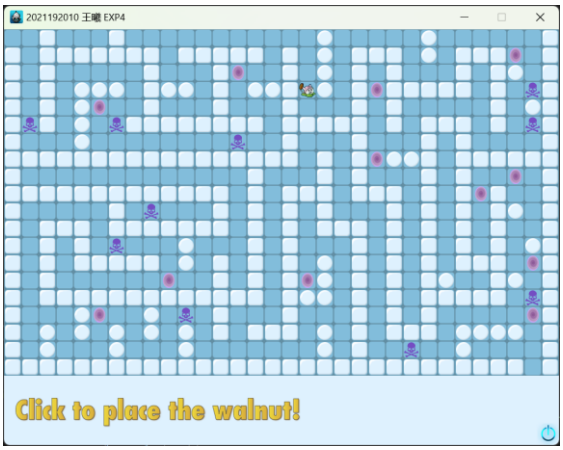


图 9.2.1：第一层场景

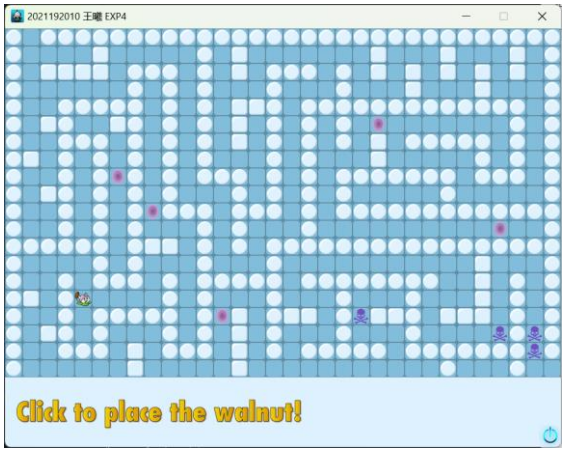


图 9.2.2：第二层场景

Level1Scene 类的 dispalyLevel()函数在第一层场景的右下角显示当前层数的提示信息“Level 1”，颜色为黑色。同理为 Level2Scene 添加。

```
// 加载层数信息
void Level1Scene::displayLevel() {
    auto levelTip = Label::createWithBMFont(TIPS_FNT_PATH, "Level 1");
    levelTip->setPosition(Point(
        visibleSize.width - levelTip->getContentSize().width / 2 - 20,
        (visibleSize.height - MAP_HEIGHT * MAP_UNIT) / 2
    ));
    levelTip->setColor(Color3B::BLACK);
    this->addChild(levelTip);
}
```

添加后，玩家可通过右下角的层数信息区分场景。如**图 9.2.3** 和**图 9.2.4** 所示。

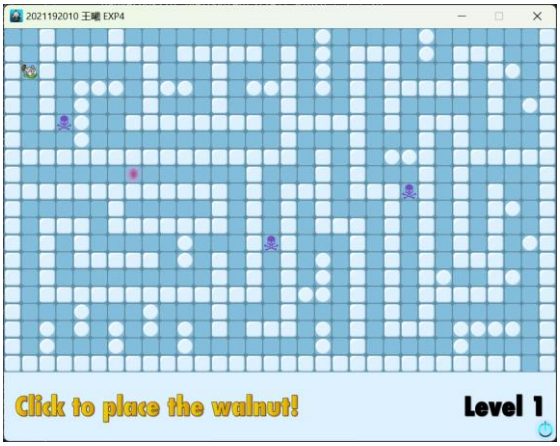


图 9.2.3：第一层场景

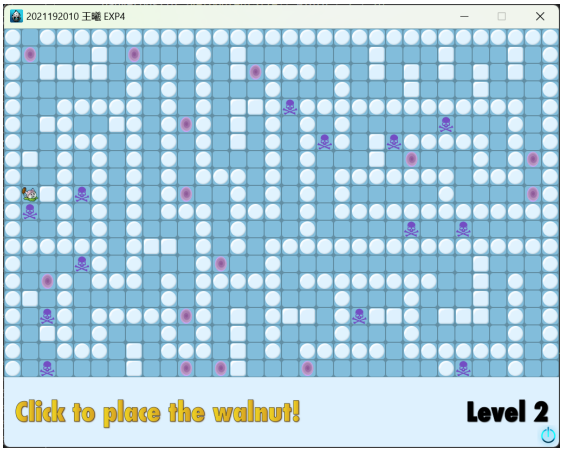


图 9.2.4：第二层场景

9.3 随机路径颜色

寻路的可视化路径采用随机颜色。

先在 Config.h 中指定可选的路径颜色，注意要用 Color4F 类型。

```
// 路径的颜色
static const std::vector<Color4F> COLORS = {
    Color4F::BLACK,
```

```
    Color4F::GREEN,
    Color4F::MAGENTA,
    Color4F::ORANGE,
    Color4F::RED,
    Color4F::WHITE,
    Color4F::YELLOW
};
```

在 Level1Scene 类的 onTouchBegan()函数中，猫尾草与坚果墙连通时，随机取一种颜色作为可视化路径的颜色。

```
// 触屏开始的回调函数
bool Level1Scene::onTouchBegan(Touch* touch, Event* event) {
    ...
    else {  // 成功放置
        ...

        if (doubleCompare(solver.getMinDistance(), INF / 2) >= 0) {  // 最短距离 >= INF / 2, 不连通
            ...
        }
        else {  // 连通, 显示路径
            ...

            auto color = COLORS[generateRandomInteger(0, (int)COLORS.size() - 1)];
            for (int i = 1; i < path.size(); i++) {
                auto currentPoint = mapCoordinate2ScreenCoordinate(path[i]);
                pathDrawer->drawLine(mapCoordinate2ScreenCoordinate(path[i - 1]), currentPoint, color);

                ...
            }

            ...
        }
    }

    return true;
}
```

运行，可视化路径以随机颜色呈现。如**图 9.3.1** 所示。

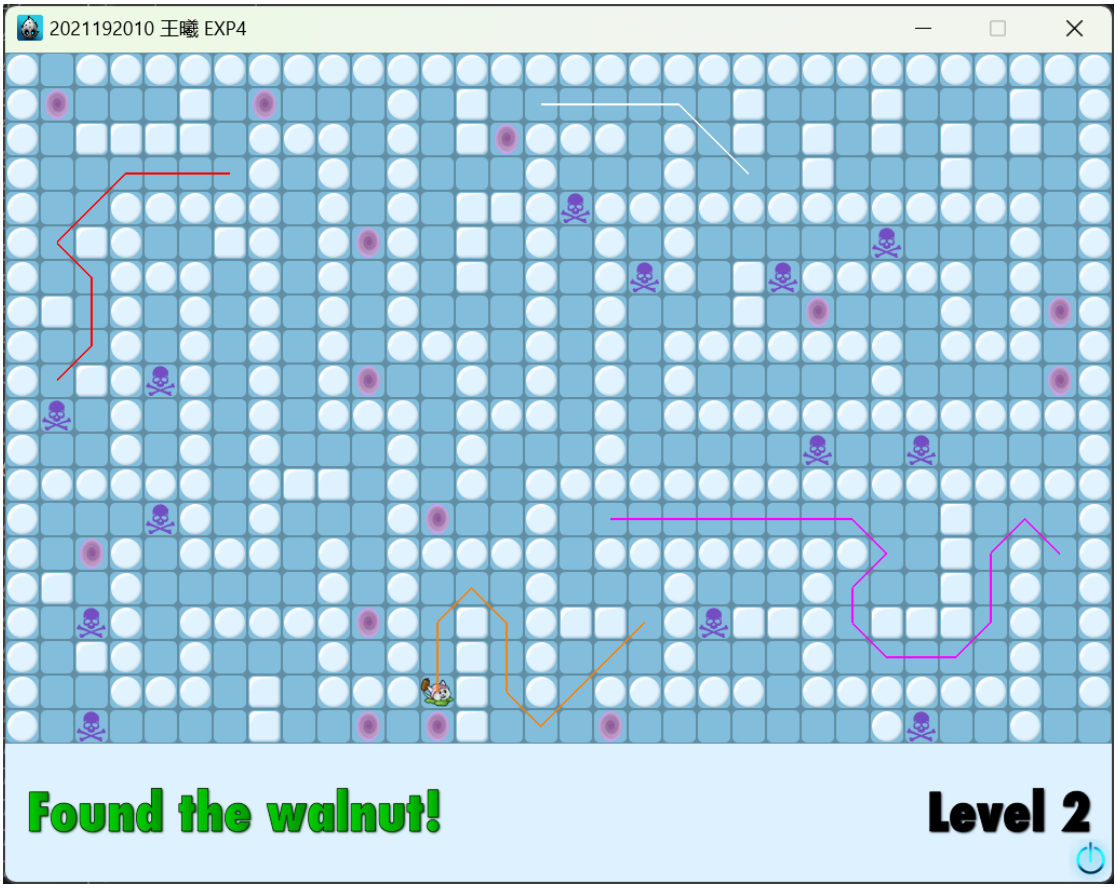


图 9.3.1：可视化路径以随机颜色呈现

9.4 随机传送门和头颅

因《Hide&Seek》的游戏体量较小，为增加可玩性，在迷宫地图的随机数量的空闲位置（未被 Tile 占用的位置）随机生成传送门或头颅。

Level1Scene 类加入成员变量：占用的点 availablePoints 和已占用的未占用的点 usedAvailablePoints。

```
class Level1Scene : public BasicScene {
private:
    // 迷宫地图
    ...
    std::set<std::pair<int, int>> availablePoints;   // 未占用的点
    std::set<std::pair<int, int>> usedAvailablePoints;   // 已占用的未占用的点


    ...
}
```

其中 availablePoints 在加载迷宫地图后初始化。

```
// 加载迷宫地图
void Level1Scene::loadTiledMaze(int mapWidth, int mapHeight, Point position, int z, std::string mapPath, int mapUnit, int tag) {
    ...
```

```
    // 预处理可用点
    accessibleMatrix = tiledMaze->getAccessibleMatrix();
    for (int i = 0; i < MAP_WIDTH; i++) {
        for (int j = 0; j < MAP_HEIGHT; j++) {
            if (accessibleMatrix[i][j]) {
                availablePoints.insert({ i, j });
            }
        }
    }
}
```

上述两个 set 通过占用一个可用的点的函数 occupyAvailablePoint()、归还一个可用的点的函数 returnAvailablePoint()和恢复可用的点的函数 recoverAvailablePoints()维护。

```
// 占用一个可用的点
void Level1Scene::occupyAvailablePoint(std::pair<int, int> point) {
    availablePoints.erase(point);
    usedAvailablePoints.insert(point);
}

// 归还一个可用的点
void Level1Scene::returnAvailablePoint(std::pair<int, int> point) {
    usedAvailablePoints.erase(point);
    availablePoints.insert(point);
}

// 恢复可用的点
void Level1Scene::recoverAvailablePoints() {
    for (auto availablePoint : usedAvailablePoints) {
        availablePoints.insert(availablePoint);
    }
    usedAvailablePoints.clear();
}
```

Level1Scene 类的 generateRandomItems()函数随机在未占用的点上随机生成传送门或头颅。

```
// 随机在未占用的点上生成 item
void Level1Scene::generateRandomItems() {
    std::vector<std::pair<int, int>> candidates;
    for (auto availablePoint : availablePoints) {
        candidates.push_back(availablePoint);
    }
    shuffle(candidates.begin(), candidates.end(), rnd);

    assert(candidates.size());

    // 生成随机数量的 item , 至少 1 个
```

```
    int cnt = generateRandomInteger(1, std::max((int)(candidates.size() * 0.1) - 1, 1));
    for (int i = 0; i < cnt; i++) {
        Point position = mapCoordinate2ScreenCoordinate(candidates[i]);

        // 随机生成一种 item
        // int randomItemIndex = generateRandomInteger(1, 4);
        int randomItemIndex = generateRandomInteger(1, 2);  // 目前只支持传送门和头颅
        if (randomItemIndex == 1) {
            loadPortal(position, 3, true);
        }
        else if (randomItemIndex == 2) {
            loadSkull(position, 3, true);
        }
        else if (randomItemIndex == 3) {
            loadSpeedDown(position, 3, true);
        }
        else {
            loadSpeedUp(position, 3, true);
        }

        occupyAvailablePoint(candidates[i]);
    }
}
```

其中 getRandomAvailablePoint()函数随机取一个空闲点。

```
// 随机取一个可用的点
std::pair<int, int> Level1Scene::getRandomAvailablePoint() {
    std::vector<std::pair<int, int>> candidates;
    for (auto availablePoint : availablePoints) {
        candidates.push_back(availablePoint);
    }

    return candidates[generateRandomInteger(0, (int)candidates.size() - 1)];
}
```

9.5 刷新分布

9.4 中随机生成的传送门和头颅的位置难免不尽人意，可能会出现猫尾草的活动范围很小的情况。为此，提供如下两个刷新分布的键盘交互：

（1）F 键：刷新猫尾草的位置，即将猫尾草重置到场景中的任一空闲点。

（2）G 键：刷新传送门和头颅的位置，即保持数量不变，将传送门和头颅重置到场景的空闲点。

```
// 键盘监听的回调函数
void Level1Scene::onKeyPressed(EventKeyboard::KeyCode keyCode, Event* event) {
    switch (keyCode) {
        ...
```

```
        case EventKeyboard::KeyCode::KEY_F: {    // F: 刷新猫尾草位置
            // 归还旧点
            auto cattailPoint = screenCoordinate2MapCoordinate(cattail->getPosition());
            returnAvailablePoint(cattailPoint);

            cattailPoint = getRandomAvailablePoint();
            Point cattailPosition = mapCoordinate2ScreenCoordinate(cattailPoint);
            cattail->setPosition(cattailPosition);
            occupyAvailablePoint(cattailPoint);

            break;
        }
        case EventKeyboard::KeyCode::KEY_G: {    // G: 刷新 item 位置
            // 传送门
            for (auto portal : portals) {
                // 归还旧点
                auto portalPoint = screenCoordinate2MapCoordinate(portal->getPosition());
                returnAvailablePoint(portalPoint);

                portalPoint = getRandomAvailablePoint();
                Point portalPosition = mapCoordinate2ScreenCoordinate(portalPoint);
                portal->setPosition(portalPosition);
                occupyAvailablePoint(portalPoint);
            }

            // 头颅
            for (auto skull : skulls) {
                // 归还旧点
                auto skullPoint = screenCoordinate2MapCoordinate(skull->getPosition());
                returnAvailablePoint(skullPoint);

                skullPoint = getRandomAvailablePoint();
                Point skullPosition = mapCoordinate2ScreenCoordinate(skullPoint);
                skull->setPosition(skullPosition);
                occupyAvailablePoint(skullPoint);
            }

            // todo: 减速、加速

            break;
        }
    }
}
```

运行，保持传送门和头颅的位置不变，刷新猫尾草的位置的效果如**图 9.5.1** 和**图 9.5.2**

所示。

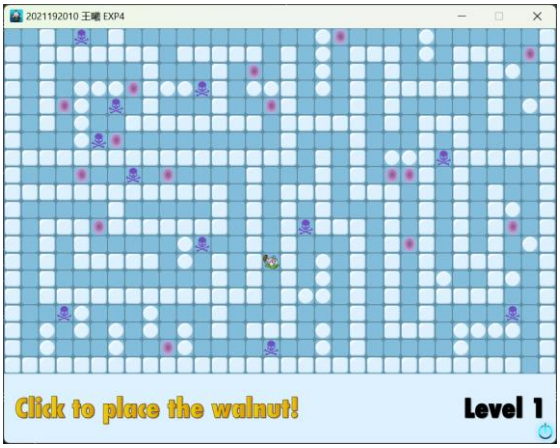


图 9.5.1：猫尾草位置刷新前

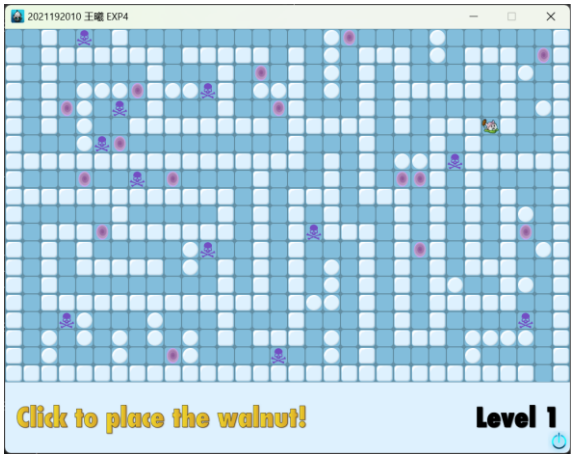


图 9.5.2：猫尾草位置刷新后

保持猫尾草的位置不变，刷新传送门和头颅的位置的效果如**图 9.5.3** 和**图 9.5.4** 所示。

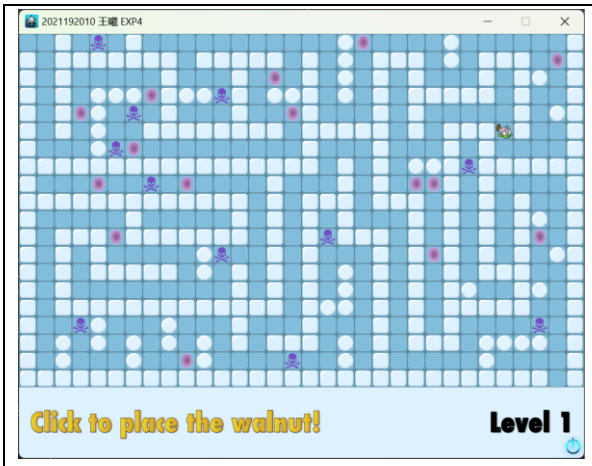


图 9.5.3：传送门和头颅位置刷新前

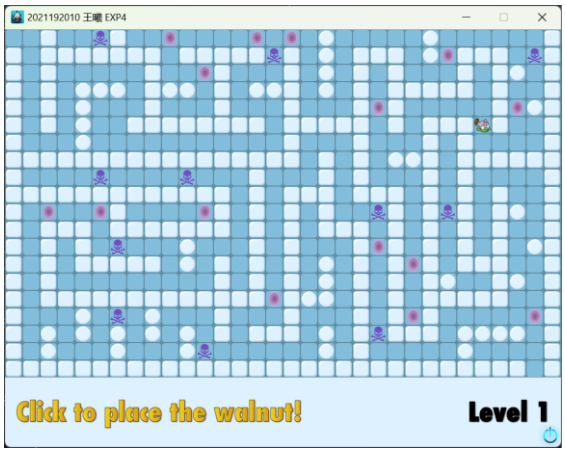


图 9.5.4：传送门和头颅位置刷新后

9.6 第二层场景加强

第一层场景中猫尾草移动到（八）相邻点的时间间隔为 0.1，随机生成传送门和头颅的密度不超过空闲点的 10%。

加强第二层场景：猫尾草移动到（八）相邻点的时间间隔为 0.05，随机生成传送门和头颅的密度不超过空闲点的 15%。

```
// 加载绘图节点
void Level2Scene::loadPathDrawer(int z) {
    ...

    moveSpeed = 0.05;
}

// 随机在未占用的点上生成 item
void Level2Scene::generateRandomItems() {
    ...

    // 生成随机数量的 item，至少 1 个
```

```
    int cnt = generateRandomInteger(1, std::max((int)(candidates.size() * 0.15) - 1, 1));
    ...
}
```

直观上，第二层场景中，猫尾草的移动速度更快，传送门和头颅更密集，猫尾草的活动范围更小。

四、实验结论或心得体会

10. 实验结论

运行录屏见【**附件**】**Hide&Seek.mp4**。

10.1 重构与编译

成功运行游戏，如**图 10.1.1** 所示。

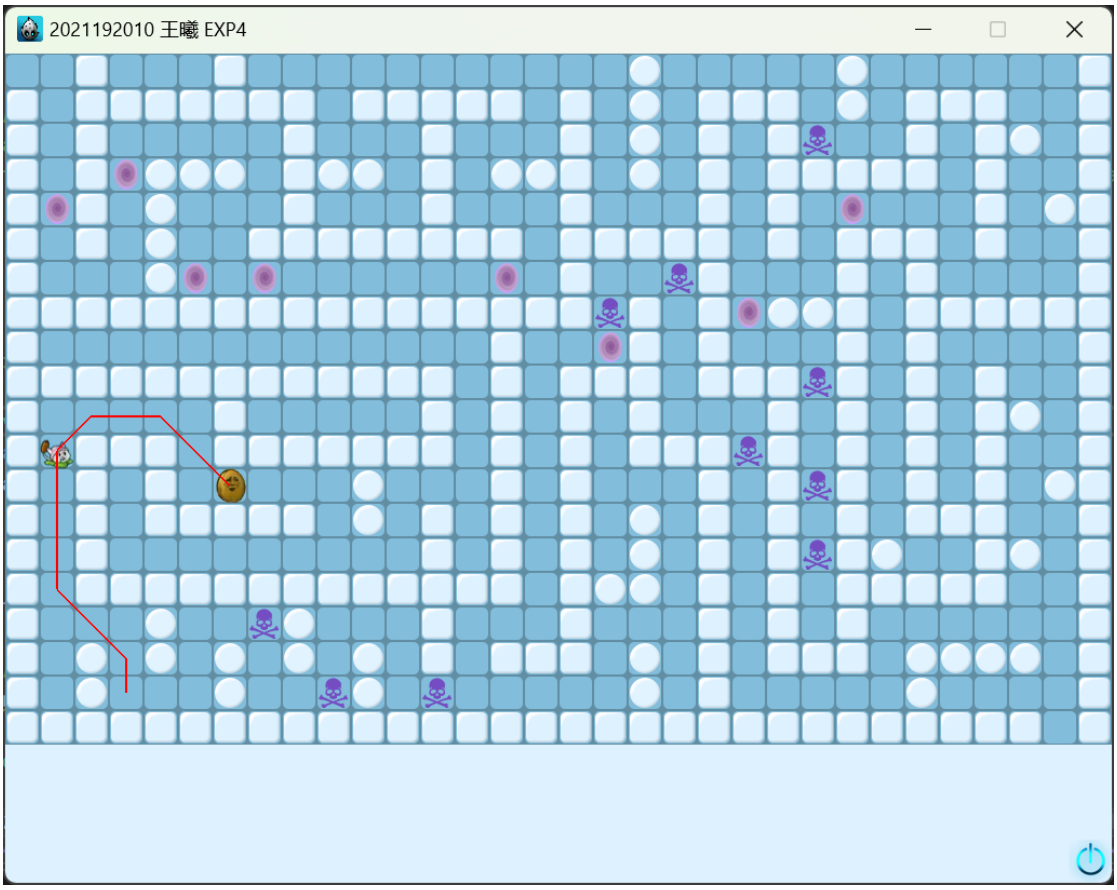


图 10.1.1：成功运行游戏

10.2 修改窗口大小

修改后，窗口可完整地显示地图。如**图 10.2.1** 所示。

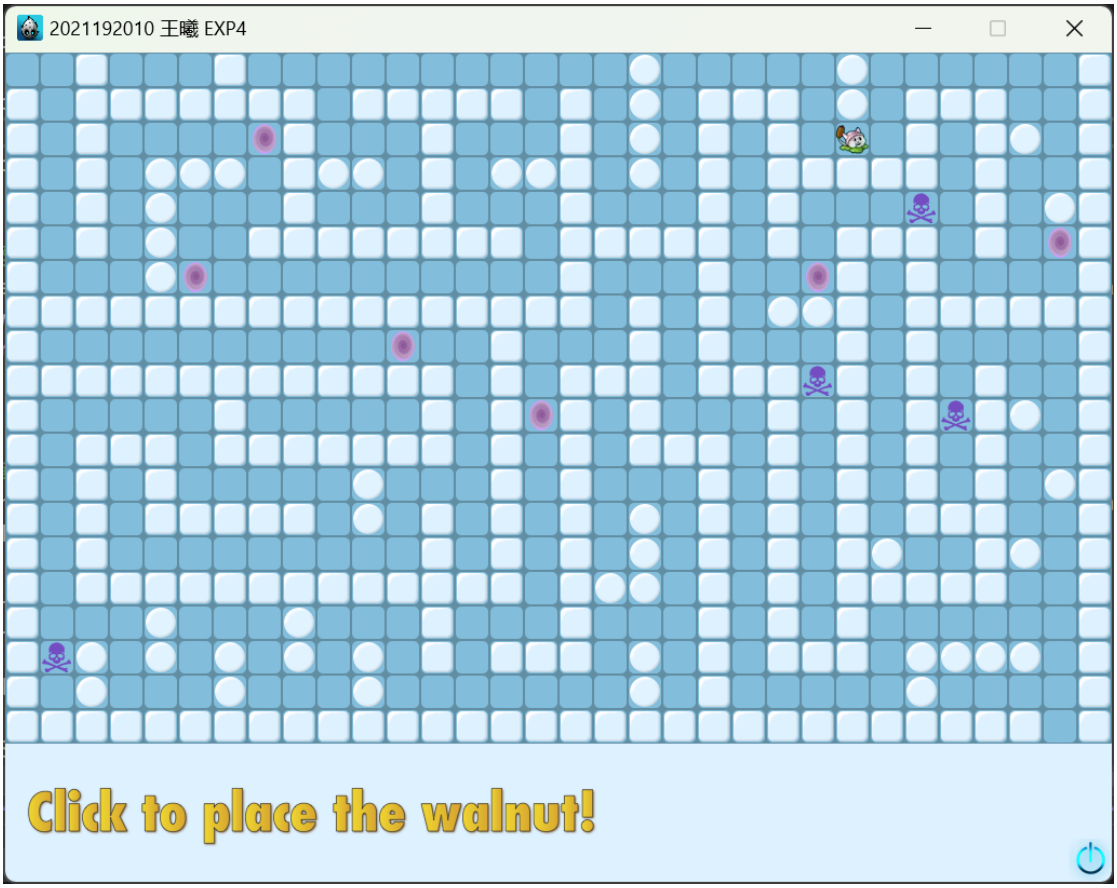


图 10.2.1：窗口可完整地显示地图

10.3 增加 Dangerous 区域

猫尾草碰到头颅（Dangerous 区域）时如**图 10.3.1** 所示。

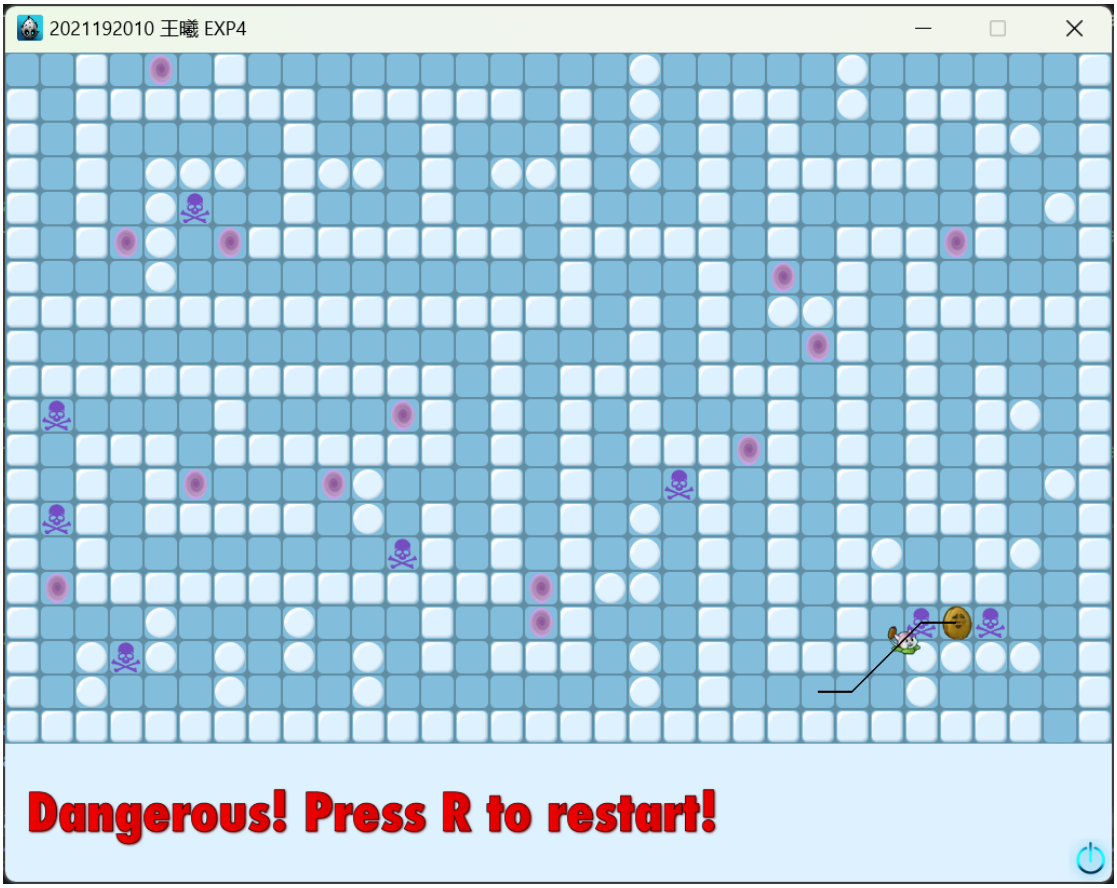


图 10.3.1：猫尾草碰到头颅

10.4 增加按键监听

《Hide&Seek》中的按键监听如下：

（1）ESC 键：关闭游戏。

（2）R 键：重玩本层。

（3）F 键：刷新猫尾草位置。

（4）G 键：刷新传送门和头颅位置。

10.5 增加地图层次

第二层的场景如**图 10.5.1** 所示。

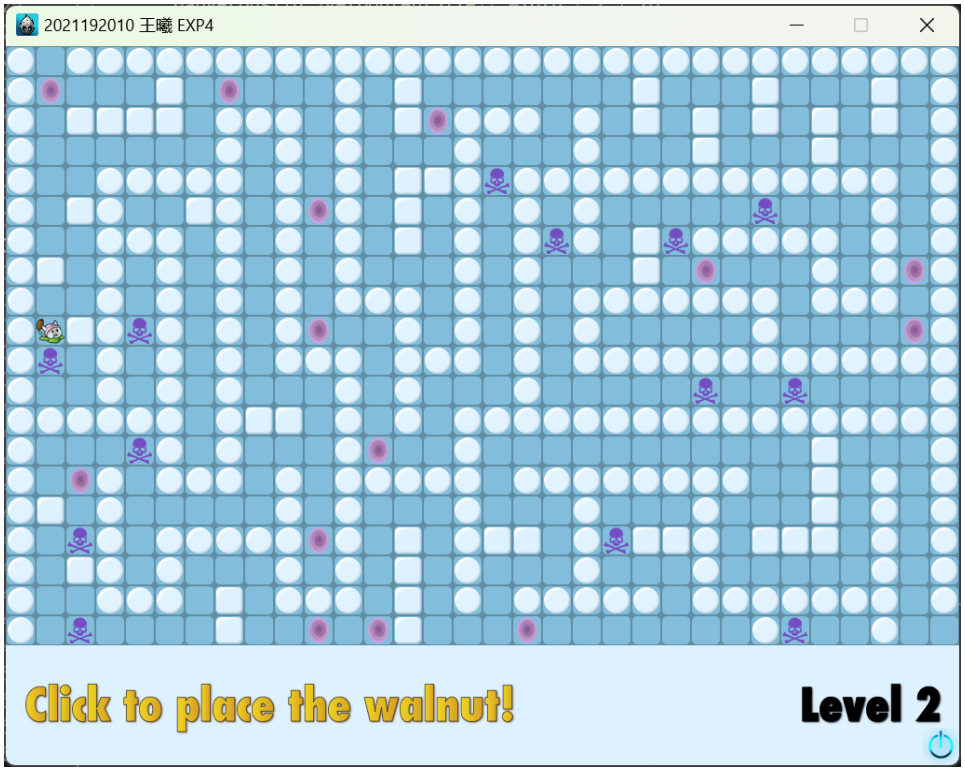


图 10.5.1：第二层的场景

10.6 增加音乐

《Hide&Seek》用到如下音效：

（1）bgm：《春雪》- 周深。

（2）猫尾草找到坚果墙的音效：阚英华（不知道是哪个游戏里的角色）的笑声。

（3）猫尾草碰到头颅的音效：阚英华死掉的声音。

10.7 修复游戏 Bug

修复了如下两个 Bug：

（1）猫尾草与坚果墙重合，即玩家将坚果墙放到猫尾草位置时，会同时出现“Found the walnut!”和“No way!”两个提示信息。

（2）玩家将坚果墙放在头颅位置，猫尾草寻路碰到坚果墙时，会同时判定“Found the walnut!”和“Dangerous! Press R to restart!”。

10.8 增加粒子效果

增加了按钮喷射上天的粒子效果。如**图 10.8.1** 所示。

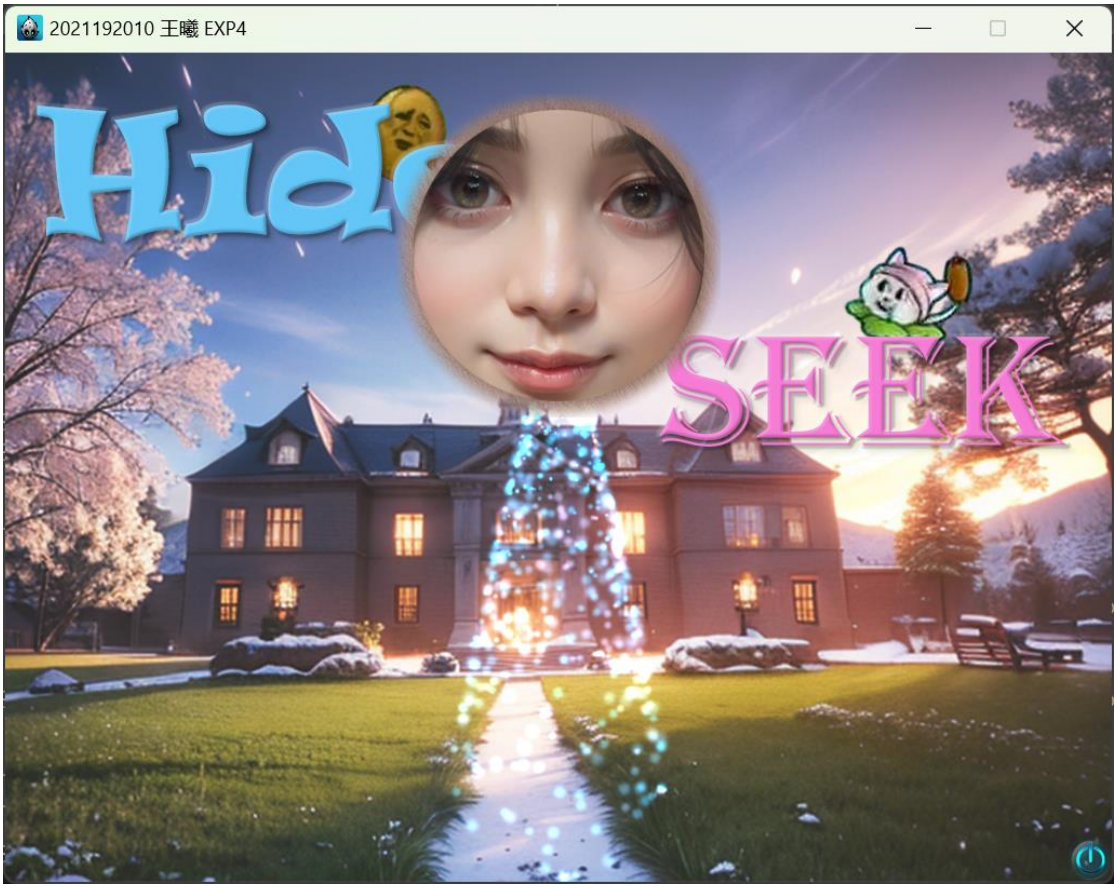


图 10.8.1：按钮喷射上天的粒子效果

10.9 游戏优化

《Hide&Seek》进行了如下优化：

（1）将场景改为与 bgm 匹配的冰雪风格。

（2）将游戏主题改为与 bgm 的歌词“我想在冬夜雪地与你追逐”撇皮的捉迷藏主题。

（3）显示当前所在层数。

（4）随机路径颜色。

（5）随机传送门和头颅。

（6）刷新猫尾草、传送门和头颅的位置。

（7）加强第二层场景。

11. 实验心得

（1）美工的重要性：玩法相同的游戏，美工好的给玩家的第一印象好，容易让玩家更有兴趣游玩。如**图 11.1** 和**图 11.2** 所示。

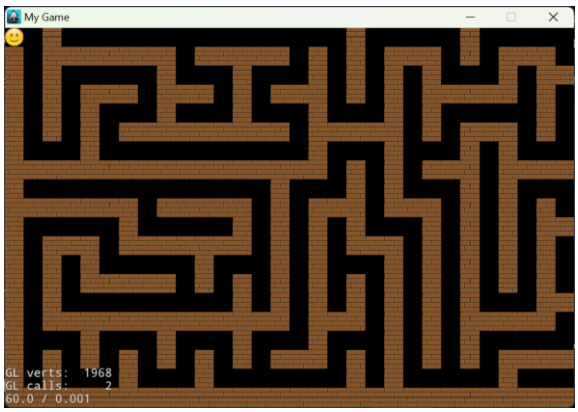


图 11.1：《迷宫寻宝》场景

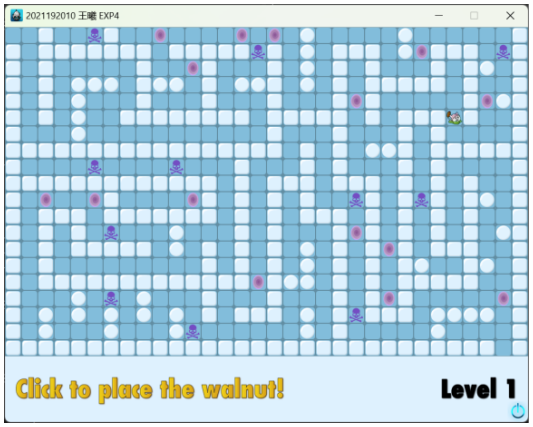


图 11.2：《Hide&Seek》场景

（2）基于瓦块地图的迷宫虽然观感整齐方正，但看久了难免有一种死板的感觉。可让玩家看到迷宫内的场景，并将本次实验的游戏作为迷宫的小地图。

（3）在游戏中加入少量掉 SAN 的、精神污染的东西可增强游戏的记忆点  。

（4）游戏中需要加入玩家操作的提示信息，否则玩家可能不知道下一步应该做什么。如我第一次看到**图 11.1** 的场景时就不知道该干什么，以为是要通过键盘操控笑脸走迷宫，但是又没看到终点在哪。

指导教师批阅意见：   成绩评定： 评语：   指导教师签字： 年　　月　　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。