1. 问题描述

问题需求

1.飞机的移动

2.敌机的生成与移动

3.子弹的生成与移动

4.碰撞检测

5.碰撞效果

6.暂停与开始

7.音效

应用场景

1.娱乐缓解压力

主要功能描述

1.通过 WSAD 控制飞机移动，空格发生子弹打击敌机

2.暂停/继续按钮控制游戏

3.点击交互效果

2. 数据结构或 UML 描述

全局变量

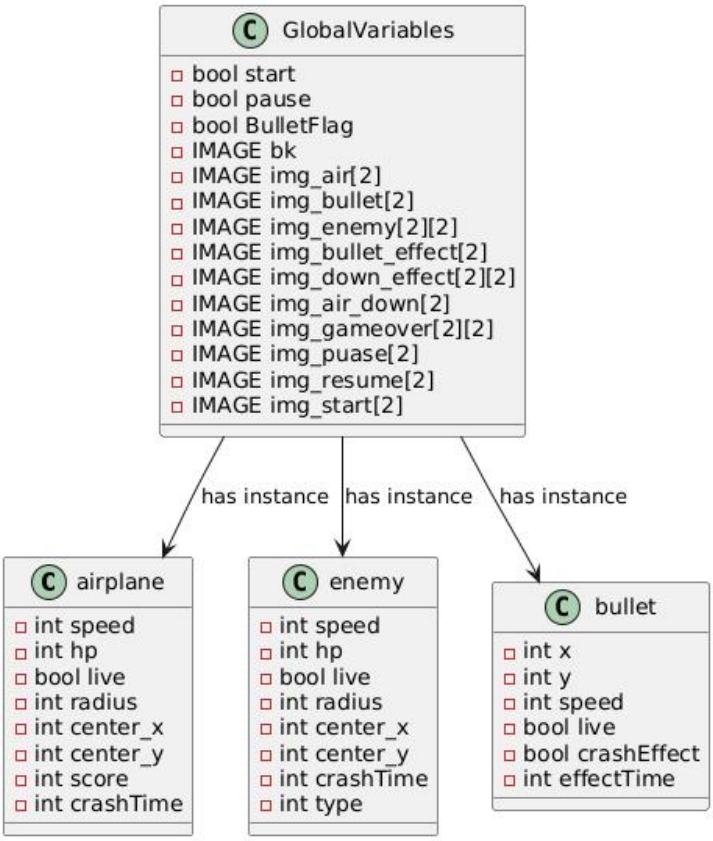


图 1-1 全局变量 uml

玩家飞机、敌机、子弹都可使用全局变量

初始化

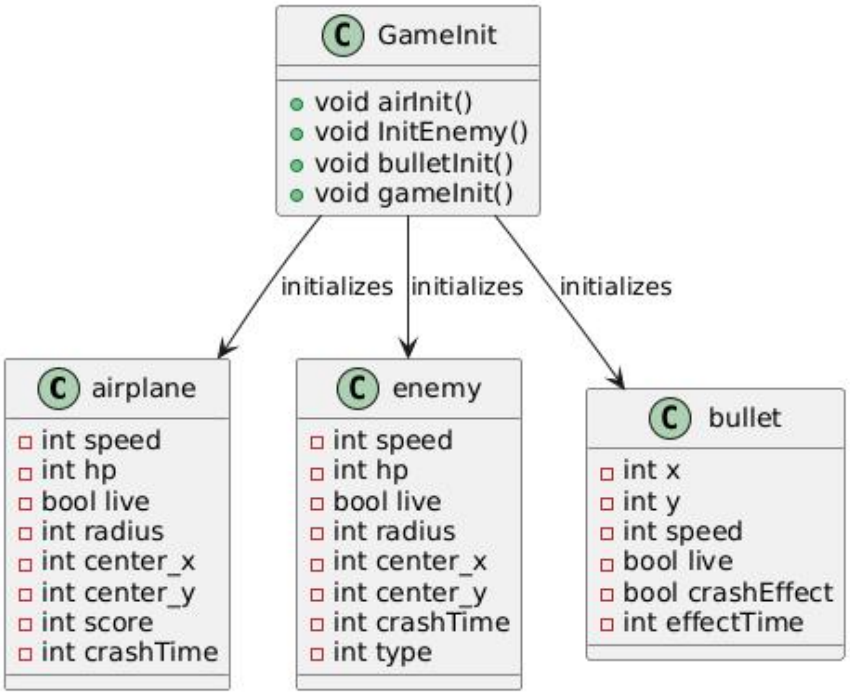


图 1-2 初始化 uml

gameinit 对玩家、敌机、子弹初始化

绘制

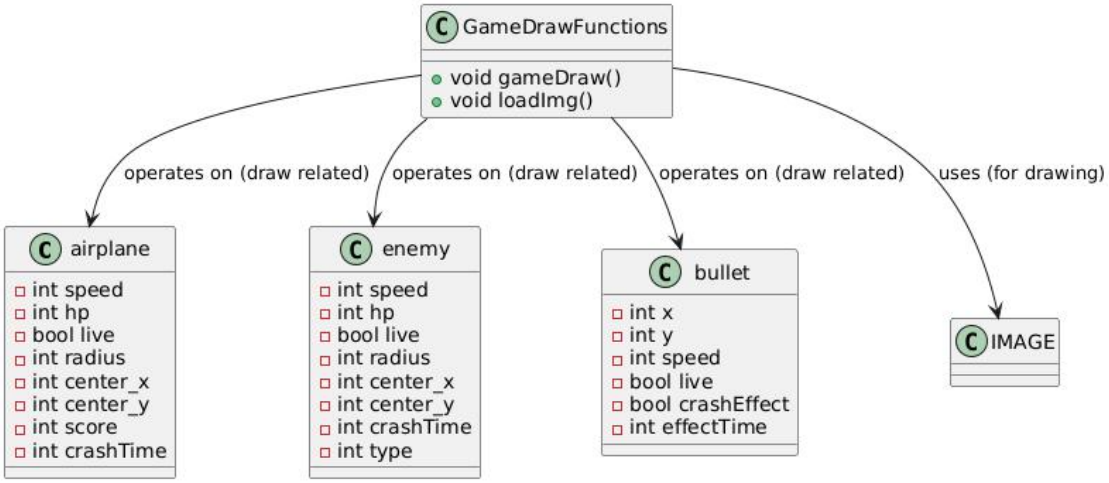


图 1-3 绘制 uml

gameDraw 绘制界面、玩家、敌机、子弹交互效果和加载图片

点击事件

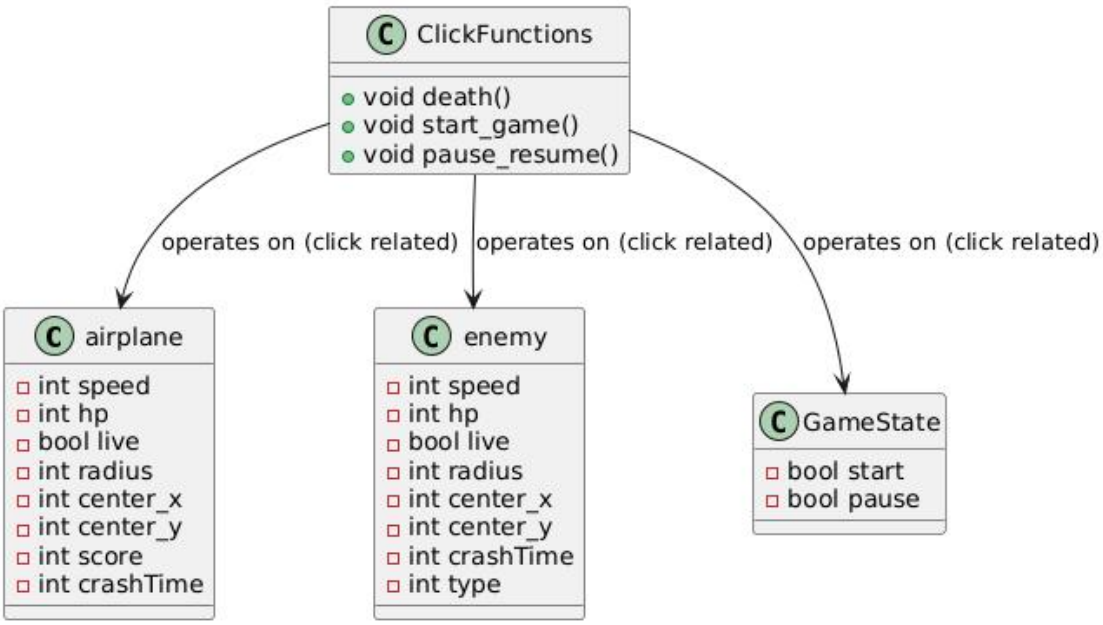


图 1-4 点击事件 uml

click 点击事件处理界面点击交互效果

BGM

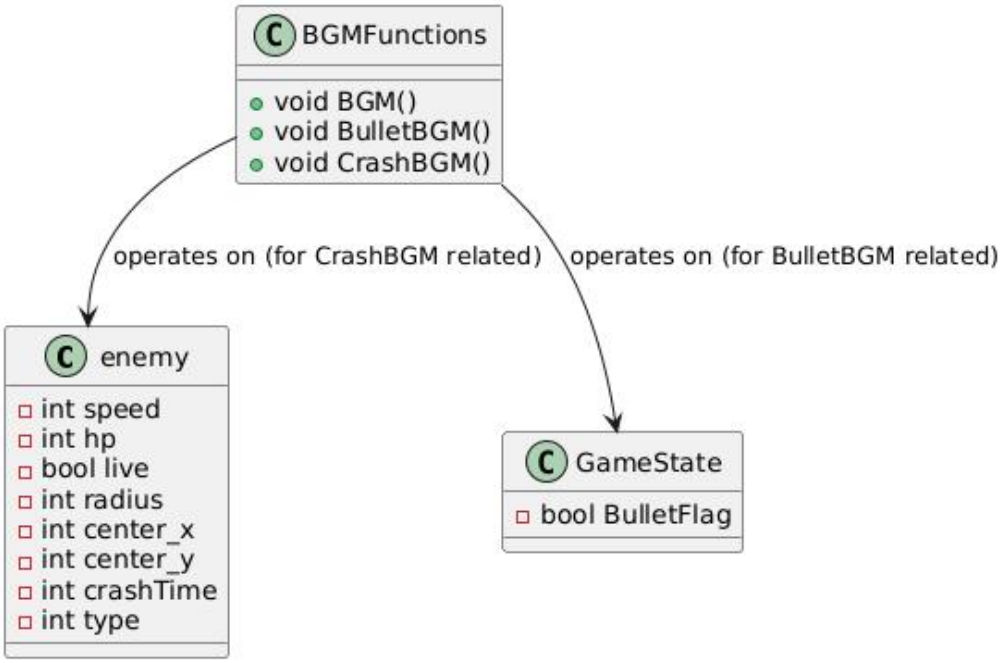


图 1-5 BGMuml

BGM 实现背景音乐和爆炸、射击音效

文本

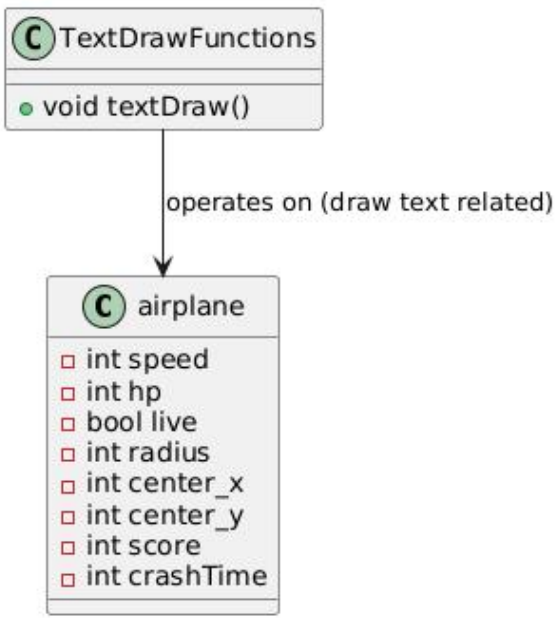


图 1-6 文本 uml

textDraw 显示玩家得分和血量

键盘控制

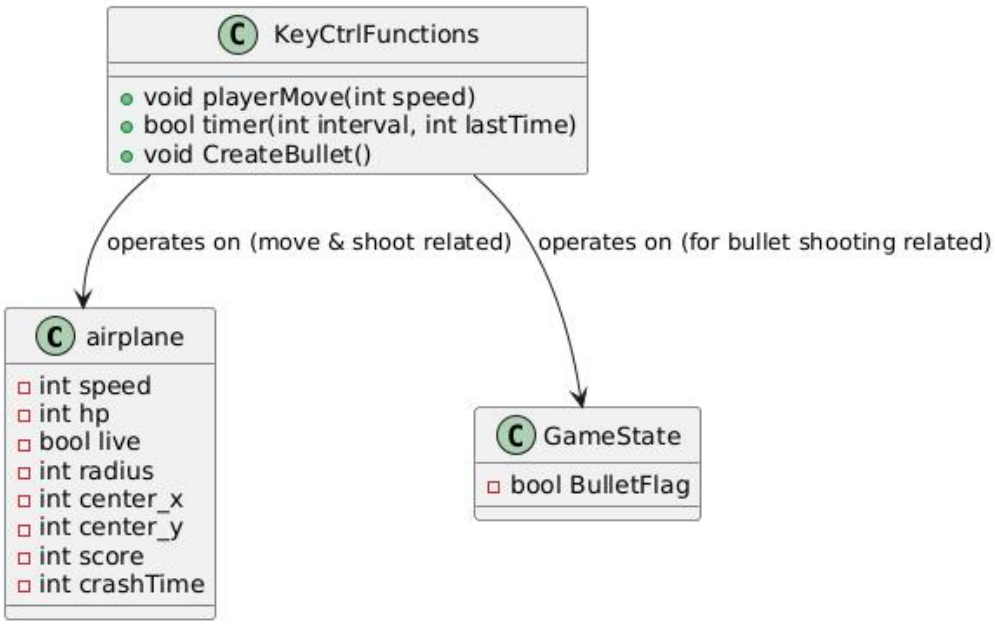


图 1-7 键盘控制 uml

keyCtrl 实现玩家按键控制飞机，发射子弹

子弹

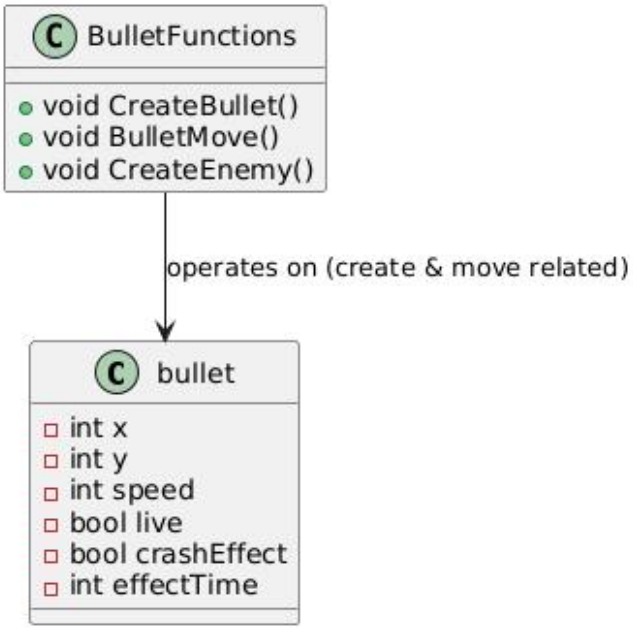


图 1-8 子弹 uml

BulletFunctions 实现子弹创建、移动、并避免在发射子弹时无法生成敌机的情况

敌机

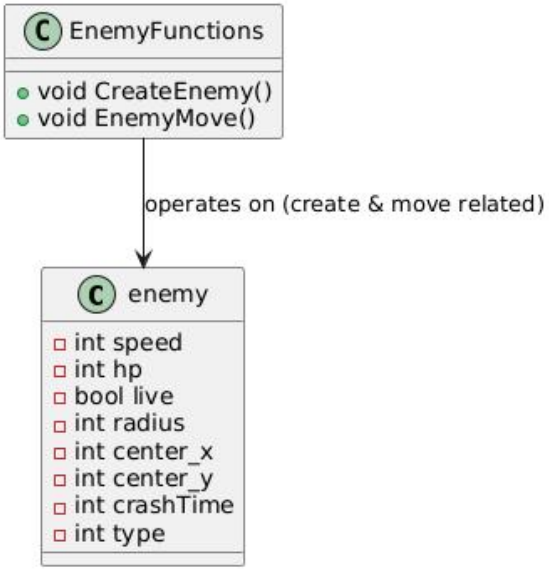


图 1-10 敌机 uml

EnemyFunctions 实现敌机创建、移动

碰撞检测

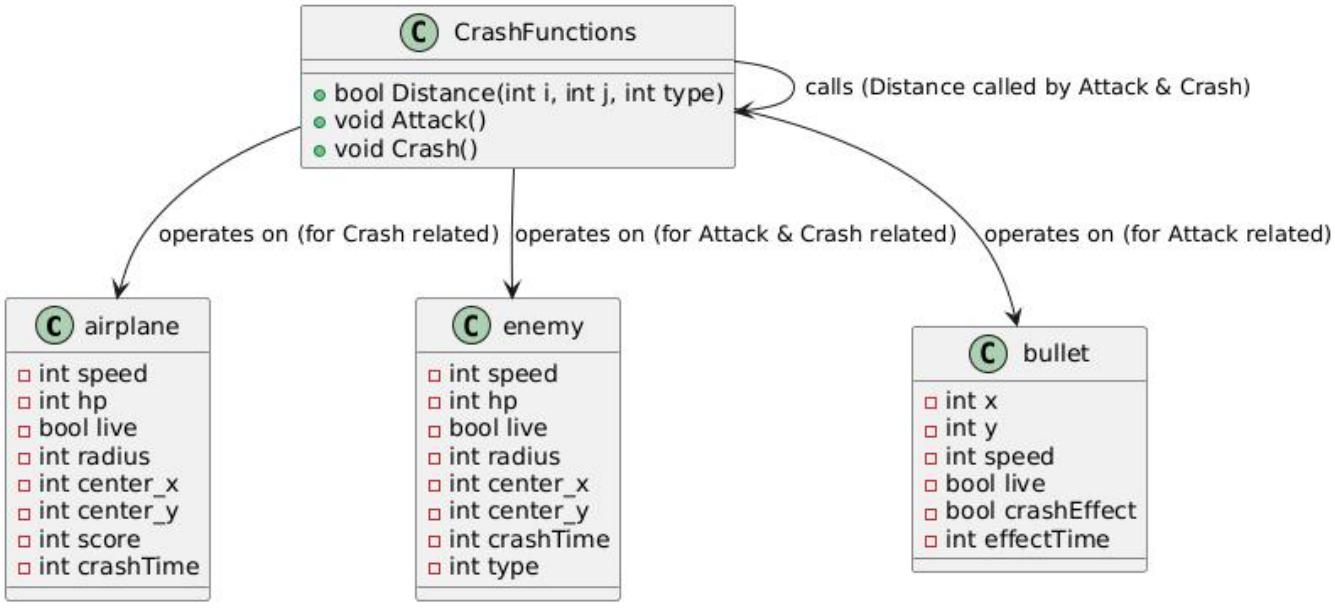
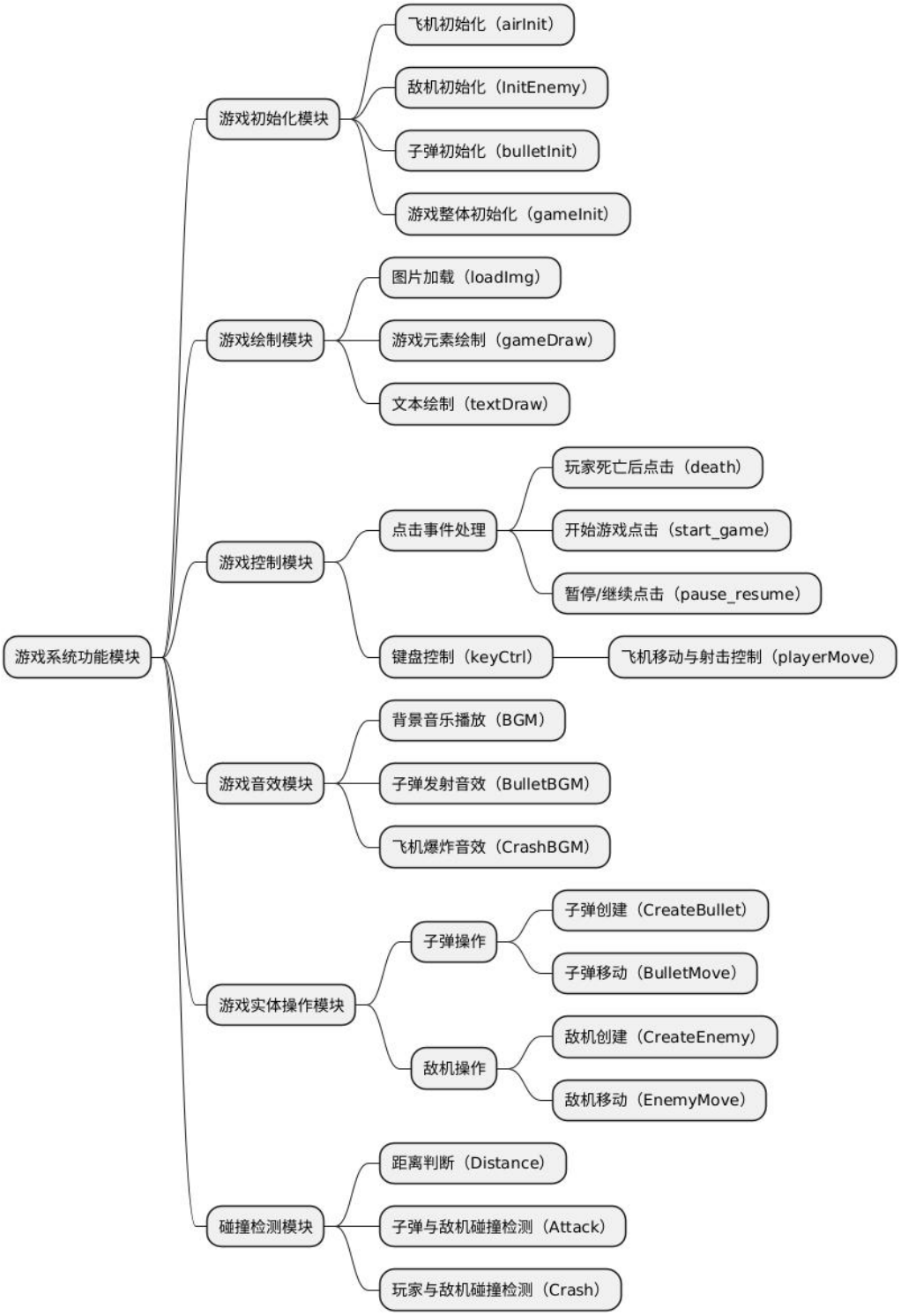


图 1-11 碰撞检测 uml

CrashFuntions 实现玩家飞机与敌机、子弹与敌机的碰撞检测

3. 算法描述

3.1 系统功能模块结构



系统总体流程图

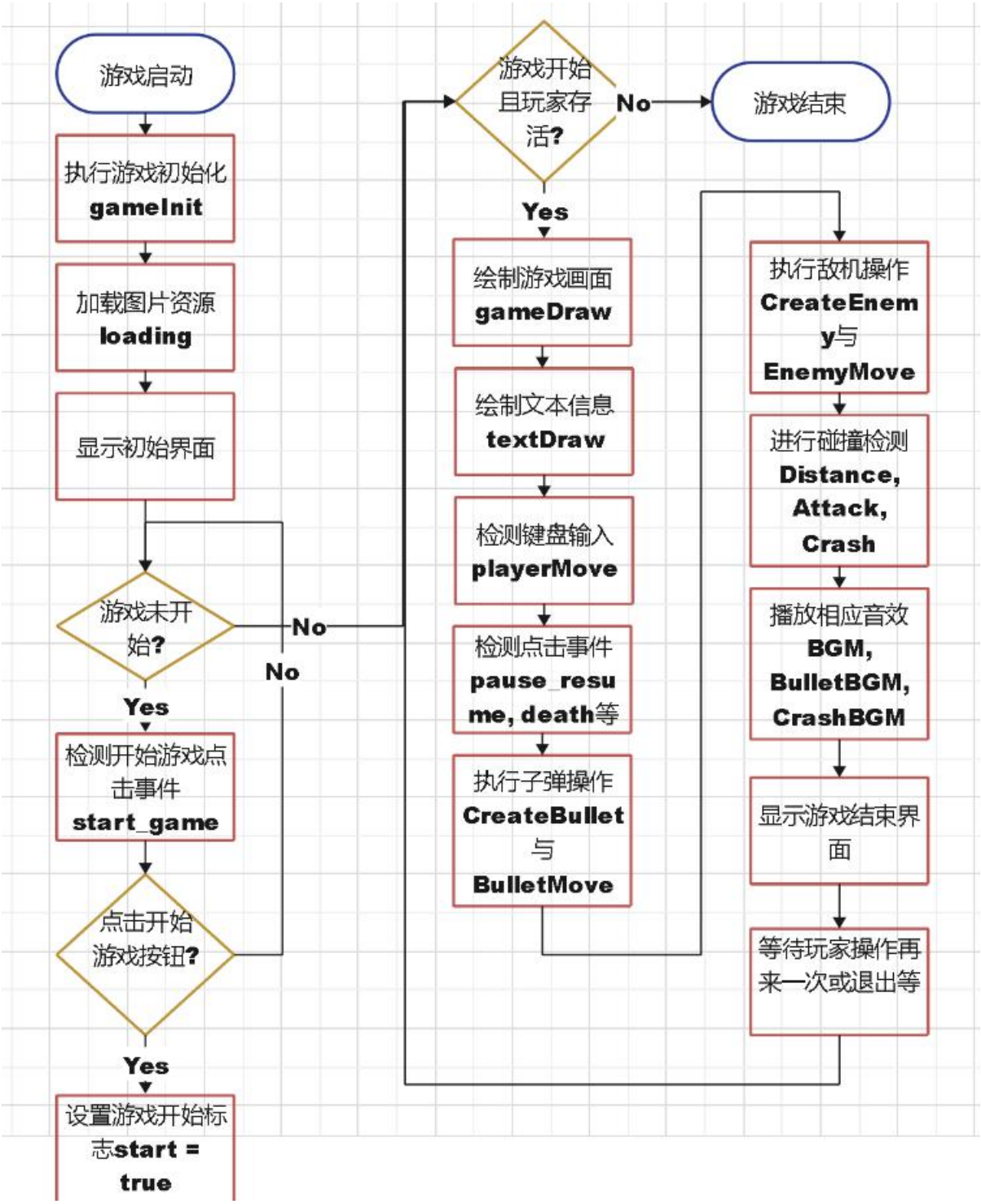


图 2-1 系统总体流程

3.2 游戏控制模块

1. 功能

“玩家死亡后点击（death）”主要处理玩家飞机死亡后，玩家点击鼠标时对应的操作，比如判断是否点击重新开始、返回主界面或者对点击到的敌机进行相关操作等；

“开始游戏点击（start_game）”针对游戏未开始时玩家点击特定区域来启动游戏的操作进行响应；

“暂停 / 继续点击（pause_resume）”则负责在游戏过程中玩家点击暂停或继续按钮时，实现游戏的暂停和恢复运行状态的功能。

“键盘控制（keyCtrl）”下的“飞机移动与射击控制（playerMove）”根据玩家按下的键盘按键来控制玩家飞机的移动方向以及子弹的发射操作，实现玩家对飞机的操控。

2. 算法描述

death()

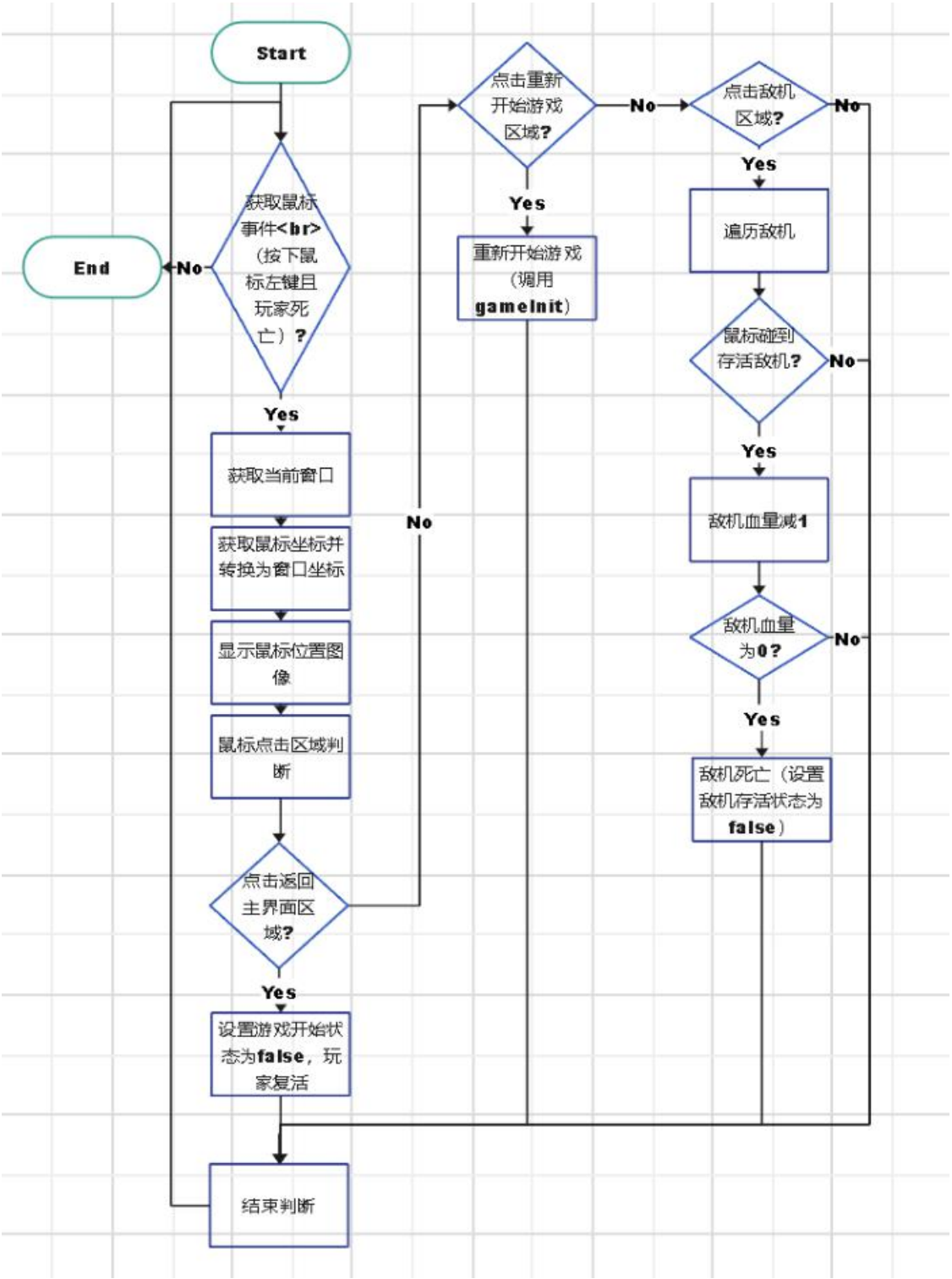


图 2-2 death()

start_game()

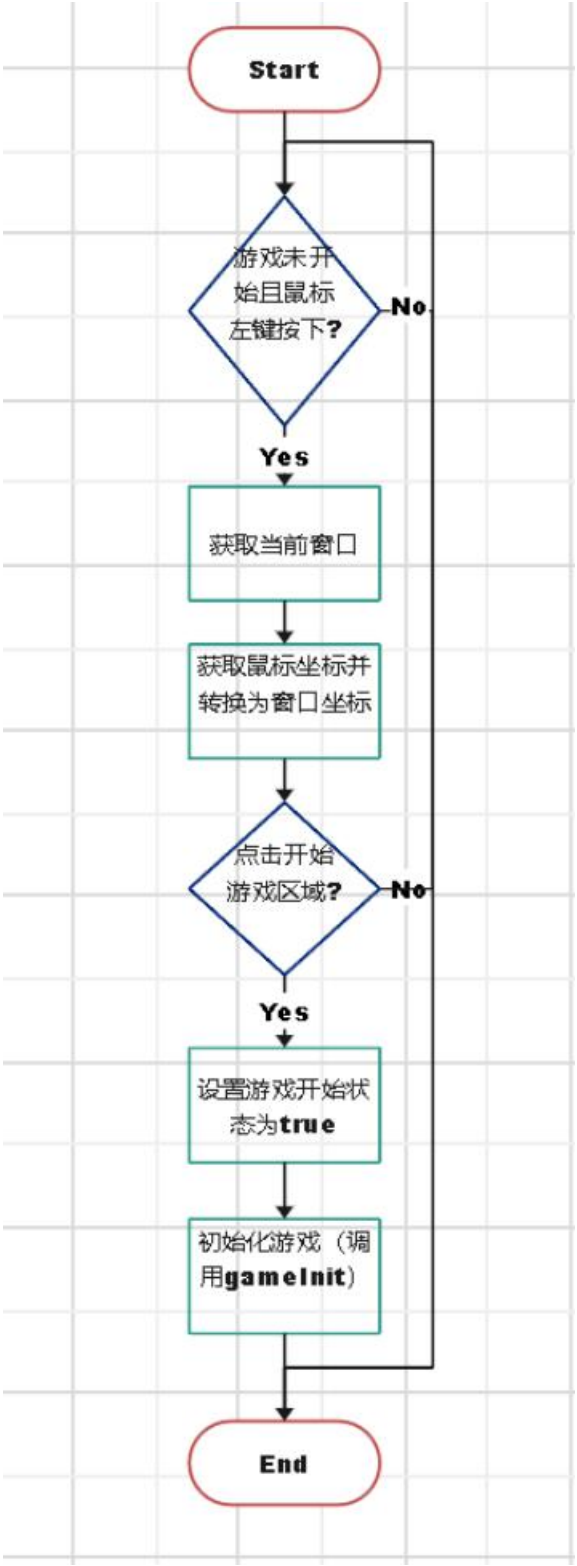


图 2-3 start_game()

pause_resume()

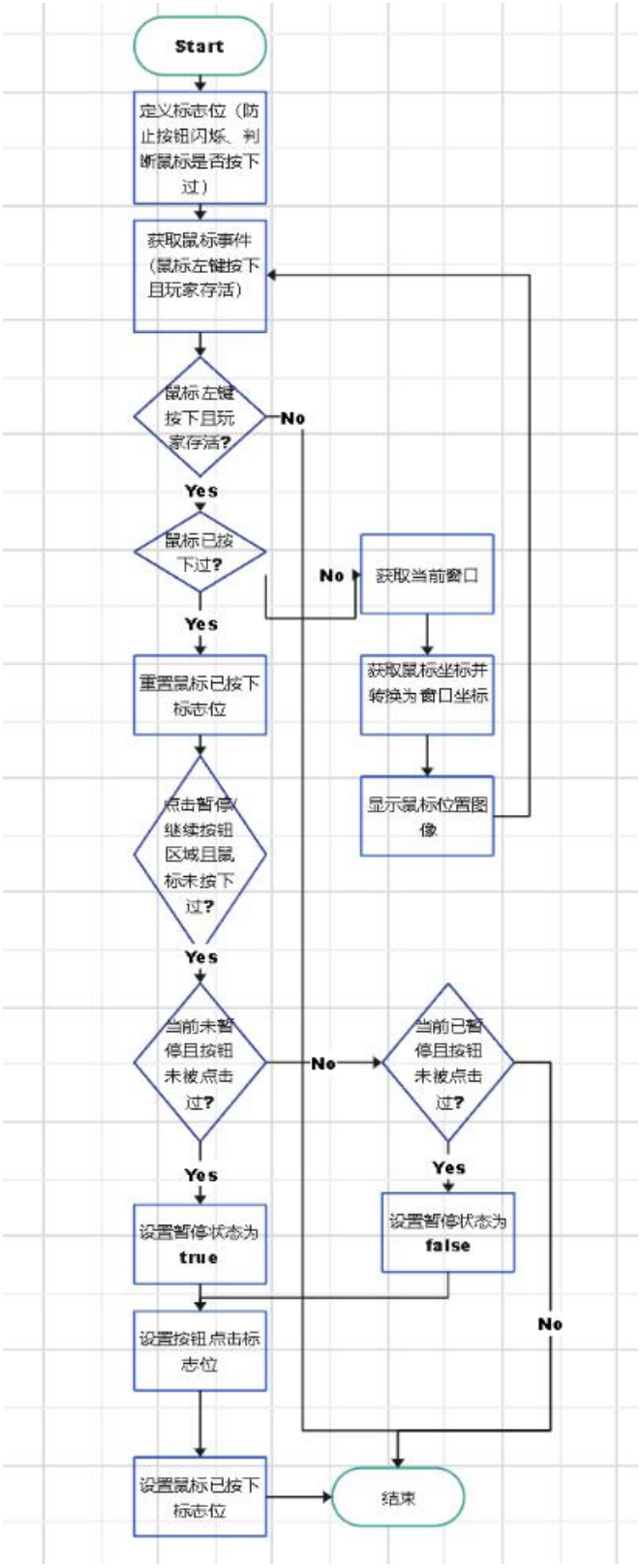


图 2-4 pause_resume()

keyCtrl()

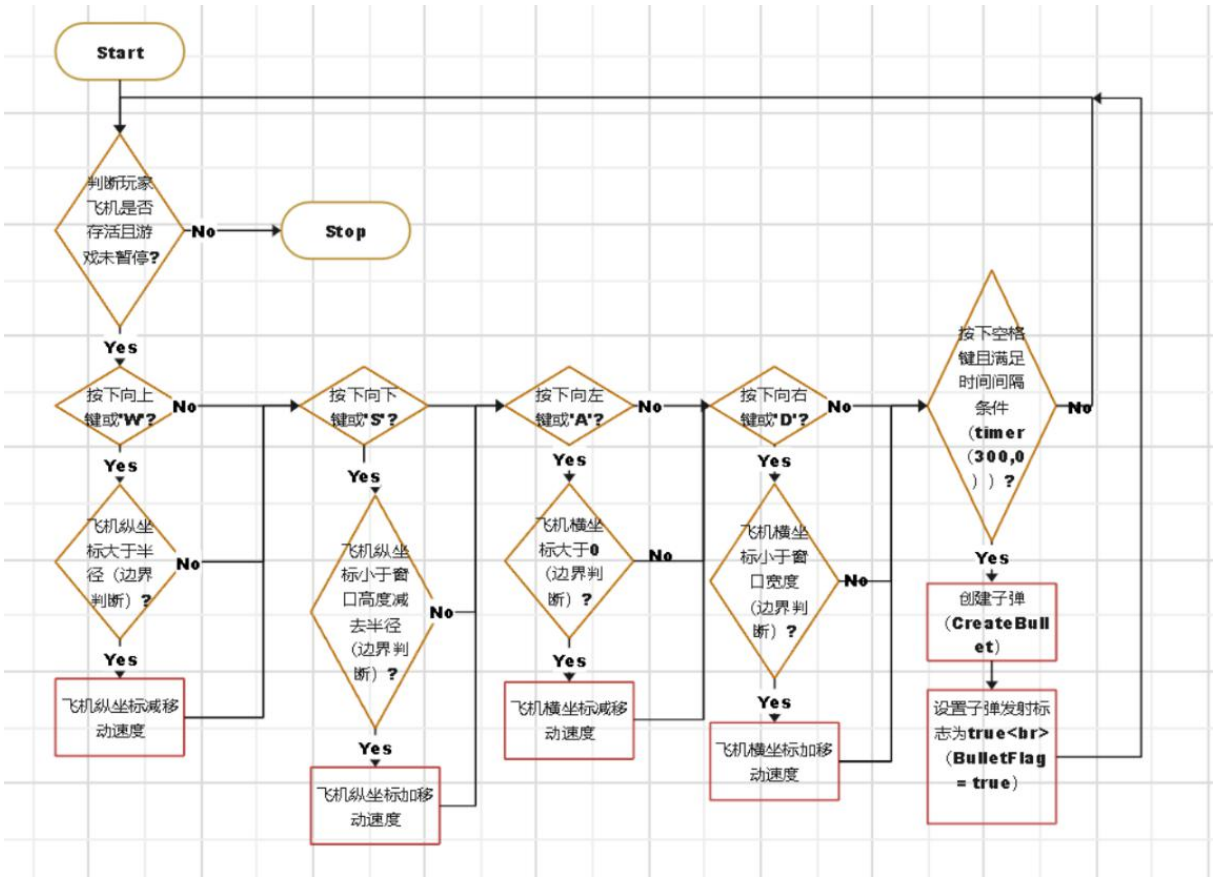


图 2-5 keyCtrl()

3. 具体语言实现

```
#include "click.h"
//玩家死亡后点击事件
void death()
{
    // 获取鼠标事件
    if (GetAsyncKeyState(VK_LBUTTON) && !player.live ) {//按下鼠标左键

        POINT p;//鼠标坐标
        HWND h = GetForegroundWindow();//获取当前窗口
        GetCursorPos(&p);//获取鼠标坐标
        ScreenToClient(h, &p);//将屏幕坐标转换为窗口坐标

        // 显示鼠标位置的图像
        putimage(p.x - 10, p.y - 10, &img_bullet_effect[0], NOTSRCERASE);
        putimage(p.x - 10, p.y - 10, &img_bullet_effect[1], SRCINVERT);

        // 判断鼠标点击的区域
        if (p.x > 325 && p.x < 385 && p.y > 250 && p.y < 310) {
            gameInit();  // 重新开始游戏
        }
        else if (p.x > 605 && p.x < 660 && p.y > 250 && p.y < 310) {//返回主界面
            start =false;
            player.live = true;
        }

        // 鼠标碰到敌机
        for (int i = 0; i < ENEMY_NUM; i++) {
            if (enemy[i].live) {//敌机存在才判断
                if (sqrt(pow(p.x - enemy[i].center_x, 2) + pow(p.y - enemy[i].center_y, 2)) < ene
my[i].radius * 2) {
                    enemy[i].hp--;//敌机血量减1
                    if (enemy[i].hp == 0) {//敌机血量为0，敌机爆炸
                        enemy[i].live = false;//敌机死亡
                    }
                }
            }
        }
    }
}



//开始游戏点击事件
void start_game()
```

```
{
    // 获取鼠标事件
    if (GetAsyncKeyState(VK_LBUTTON) && start == false ) { // 游戏未开始，且鼠标左键按下才开始游戏
        POINT p; // 鼠标坐标
        HWND h = GetForegroundWindow(); // 获取当前窗口
        GetCursorPos(&p); // 获取鼠标坐标
        ScreenToClient(h, &p); // 转换为窗口坐标

        if (p.x > 465 && p.x < 550 && p.y > 280 && p.y < 345) {
            start = true;//开始游戏
            gameInit();//初始化游戏
        }

    }
}

//暂停、继续点击事件
void pause_resume()
{
    static int pause_resume_flag = 0; // 防止按钮闪烁的标志位
    static bool was_mouse_down = false; // 判断鼠标是否按下过
    // 获取鼠标事件
    if (GetAsyncKeyState(VK_LBUTTON) && player.live) { // 鼠标左键按下
        POINT p; // 鼠标坐标
        HWND h = GetForegroundWindow(); // 获取当前窗口
        GetCursorPos(&p); // 获取鼠标坐标
        ScreenToClient(h, &p); // 转换为窗口坐标

        // 显示鼠标位置的图像
        putimage(p.x - 10, p.y - 10, &img_bullet_effect[0], NOTSRCERASE);
        putimage(p.x - 10, p.y - 10, &img_bullet_effect[1], SRCINVERT);

        if (p.x > 950 && p.x < 1000 && p.y > 0 && p.y < 40 && !was_mouse_down) {
            // 按钮点击逻辑
            if (!pause && pause_resume_flag == 0) { // 点击暂停按钮
                pause = true; // 暂停
                pause_resume_flag = 1; // 设置按钮状态已被点击
            }
            else if (pause && pause_resume_flag == 0) { // 点击继续按钮
                pause = false; // 继续
                pause_resume_flag = 1; // 设置按钮状态已被点击
            }
            was_mouse_down = true; // 设置鼠标已经按下
        }
```

```
        }
        else {
            // 鼠标松开后，复位状态
            if (was_mouse_down) {
                was_mouse_down = false; // 重置标志
                pause_resume_flag = 0; // 重置按钮状态标志，允许下一次点击
            }
        }
}
#include "keyCtrl.h"
//飞机的移动和子弹射击控制
void playerMove(int speed)
{
    if (player.live) {//飞机存活且未暂停才移动飞机和射击子弹
        //GetAsyncKeyState 函数流畅的读取键盘状态
    //检测字母方向键用大写，这样大小写都可以检测到
        if (GetAsyncKeyState(VK_UP) || GetAsyncKeyState('W')) {
            if (player.center_y > player.radius) {//边界判断
                player.center_y -= speed;
            }
        }
        if (GetAsyncKeyState(VK_DOWN) || GetAsyncKeyState('S')) {
            if (player.center_y < HEIGHT - player.radius) {//边界判断
                player.center_y += speed;
            }
        }
        if (GetAsyncKeyState(VK_LEFT) || GetAsyncKeyState('A')) {
            if (player.center_x > 0) {//边界判断，让飞机可以打击边界
                player.center_x -= speed;
            }
        }
        if (GetAsyncKeyState(VK_RIGHT) || GetAsyncKeyState('D')) {
            if (player.center_x < WIDTH) {//边界判断，让飞机可以打击边界
                player.center_x += speed;
            }
        }
        if (GetAsyncKeyState(VK_SPACE) && timer(300, 0)) {//按空格键，并且两次按键时间间隔大
于300ms，才发射子弹,也就是子弹发射频率
            CreateBullet();
            BulletFlag = true;
        }
    }
}
```

3.3 游戏实体操作模块模块

1. 功能

创建子弹，实现子弹的移动

创建敌机，实现敌机的移动

2. 算法描述

CreateBullet()

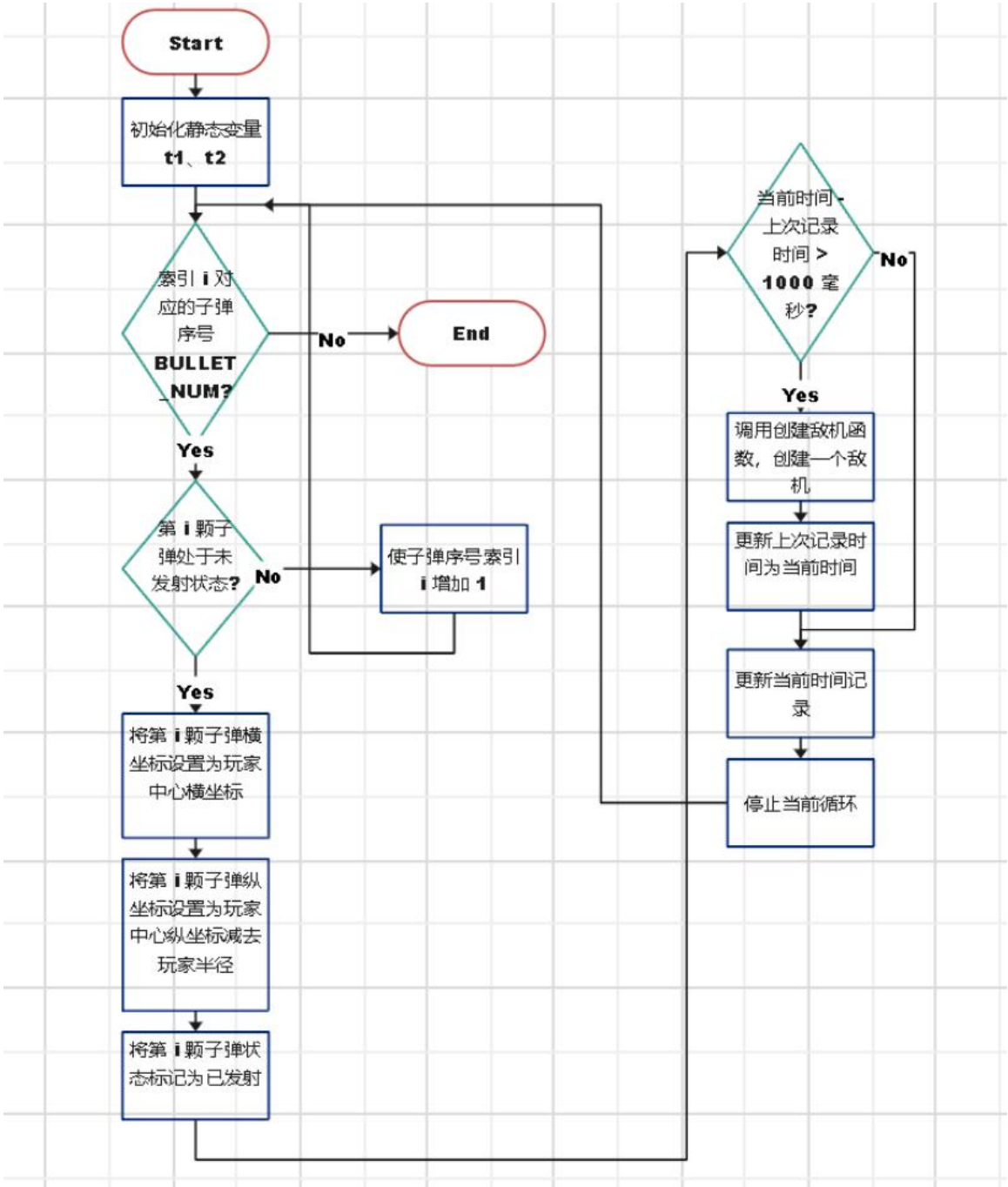


图 2-6 CreateBullet()

BulletMove()

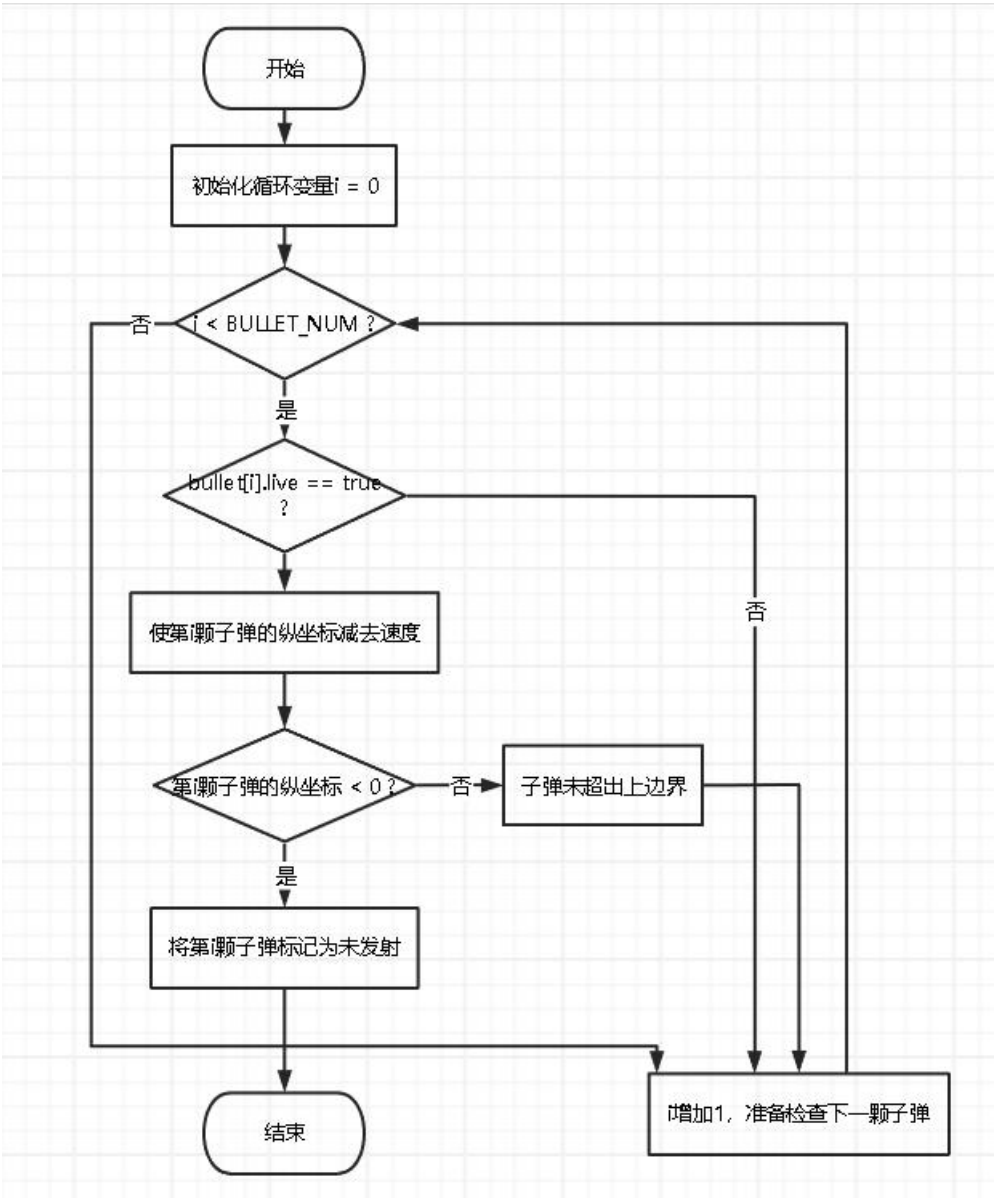


图 2-7 BulletMove()

CreateEnemy()

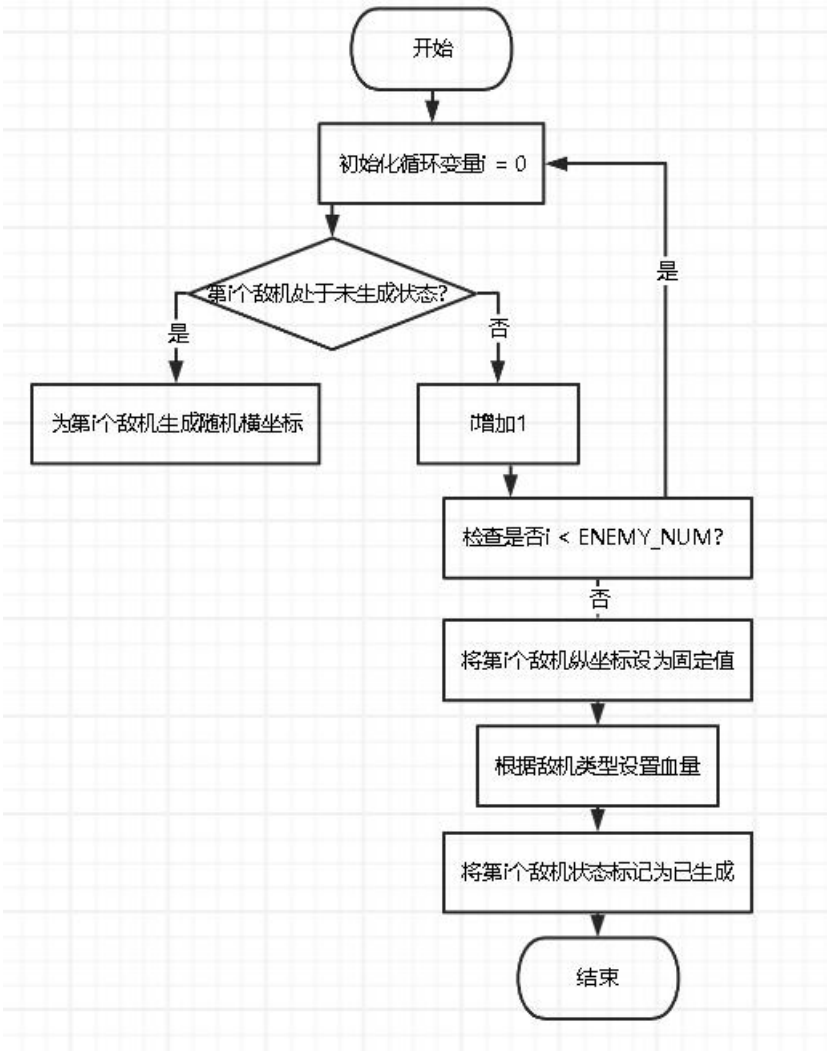


图 2-8 CreateEnemy()

EnemyMove()

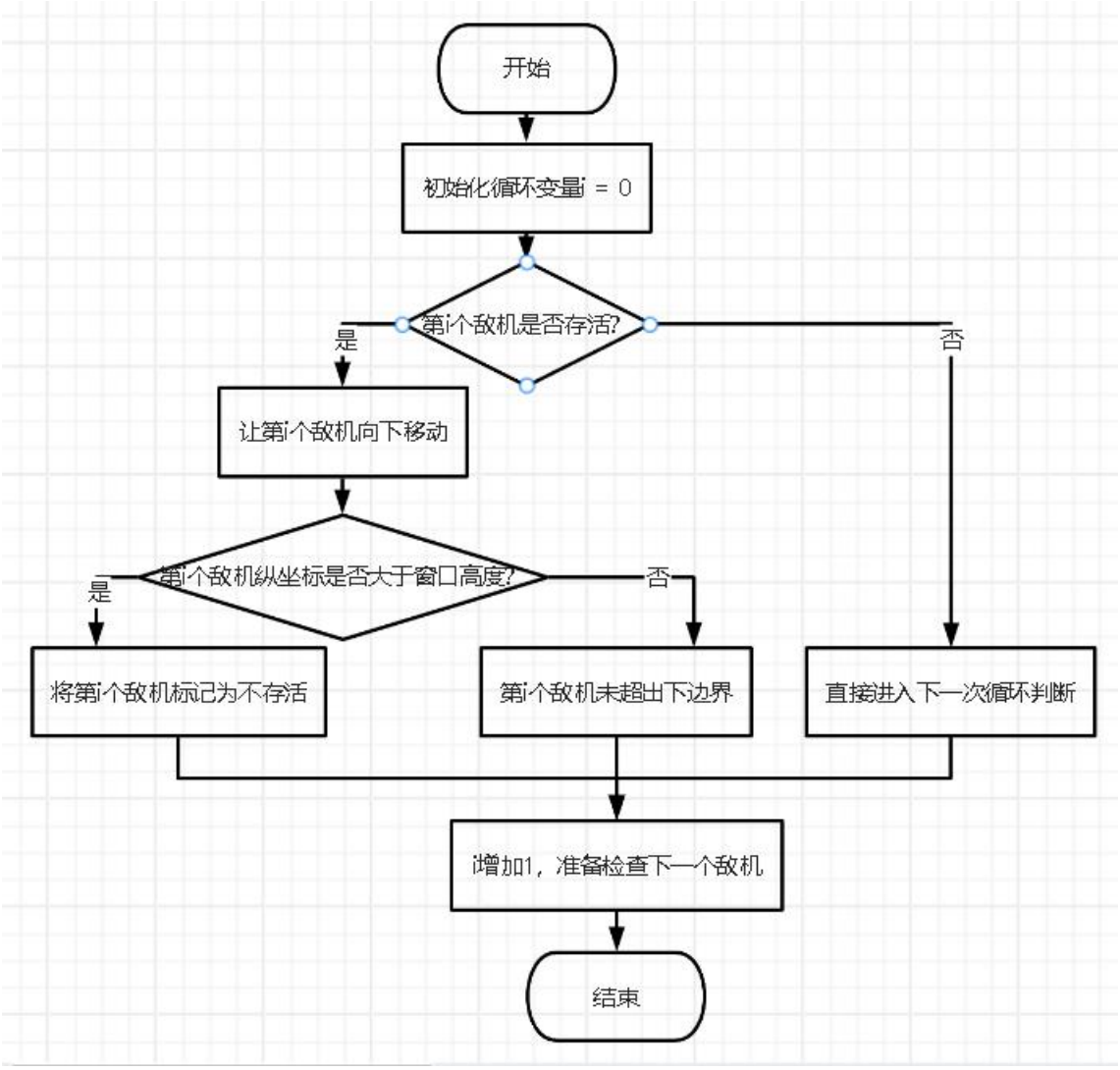


图 2-9 EnemyMove()

3. 具体语言实现

```
#include "bullet.h"
//子弹的创建
void CreateBullet()
{
    static DWORD t1 = 0, t2 = 0;

    for (int i = 0; i < BULLET_NUM; i++) {
        if (!bullet[i].live) {  // 子弹未发射
            bullet[i].x = player.center_x;  // 子弹的横坐标
            bullet[i].y = player.center_y - player.radius;  // 子弹的纵坐标
            bullet[i].live = true;  // 射出子弹后，子弹状态为已发射

            // 创建敌机的逻辑,防止进入按键检测不能创建敌机
            if (t2 - t1 > 1000) {  // 每 1 秒创建一次敌机
                CreateEnemy();
                t1 = t2;
            }
            t2 = clock();

            break;  // 产生一个子弹就退出
        }
    }
}
//子弹的移动
void BulletMove()
{
    for (int i = 0; i < BULLET_NUM; i++)
    {
        if (bullet[i].live)//子弹发射才移动子弹
        {
            bullet[i].y -= bullet[i].speed;
            if (bullet[i].y < 0) {//子弹射出边界，子弹消失
                bullet[i].live = false;
            }
        }
    }

}
#include "enemy.h"
//敌机的创建
void CreateEnemy()
{
    for (int i = 0; i < ENEMY_NUM; i++) {
```

```
            if (!enemy[i].live)//没有敌机时生成敌机
            {
                //x 坐标随机，y 坐标固定
                enemy[i].center_x = rand() % (WIDTH - enemy[i].radius);
                enemy[i].center_y = enemy[i].radius;
                //血量重置
                enemy[i].hp = enemy[i].type == BIG_ENEMY ? 3 : 1;
                enemy[i].live = true;
                break;//生产完一架敌机就退出循环
            }
        }
    }
    //敌机的移动
    void EnemyMove()
    {
        for (int i = 0; i < ENEMY_NUM; i++) {
            if (enemy[i].live)//敌机存在才移动敌机
            {
                enemy[i].center_y += enemy[i].speed;//敌机上下移动
                if (enemy[i].center_y > HEIGHT)//敌机出边界，敌机消失
                {
                    enemy[i].live = false;
                }
            }
        }

    }
```

3.4 碰撞检测模块

1. 功能

检测玩家飞机与敌机是否碰撞

检测子弹是否击中敌机

2. 算法描述

Distance(int i, int j, int type)

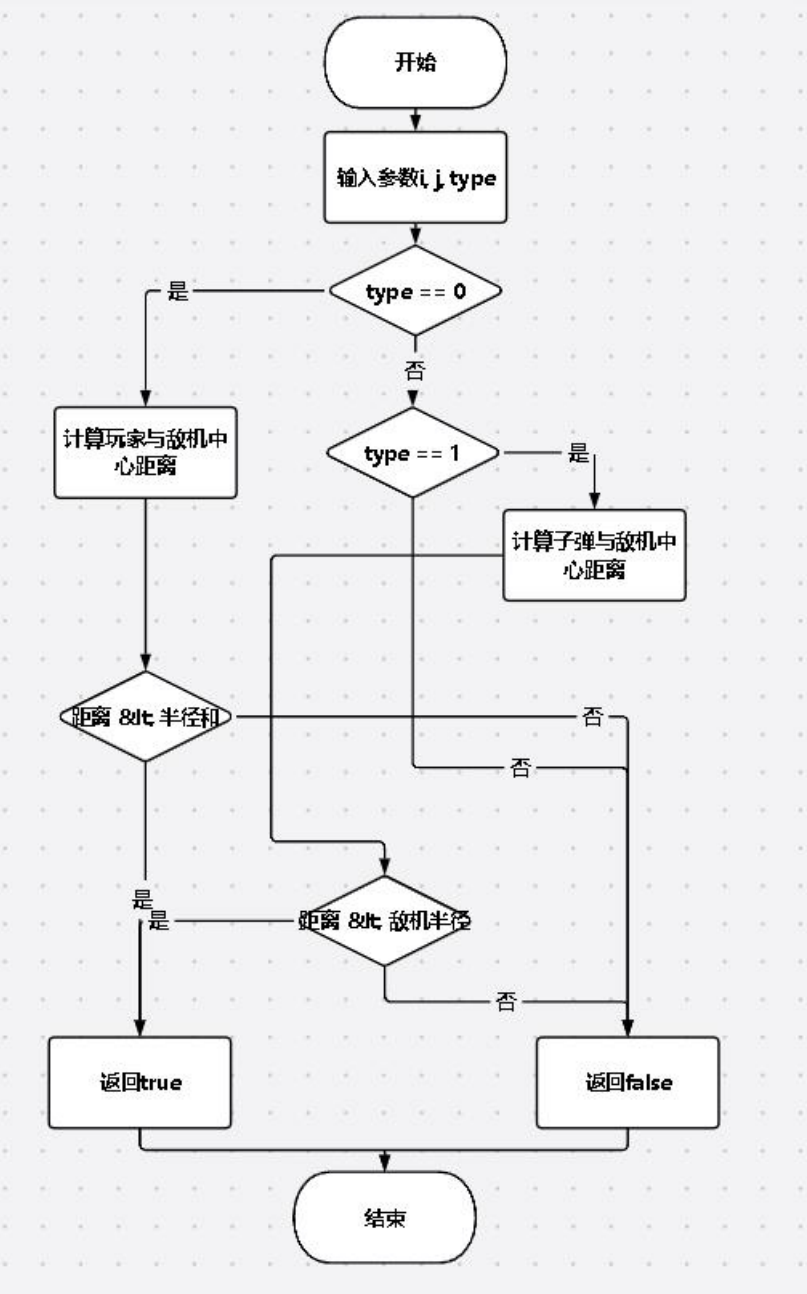


图 2-11 Distance

使用圆形碰撞检测

Attack()

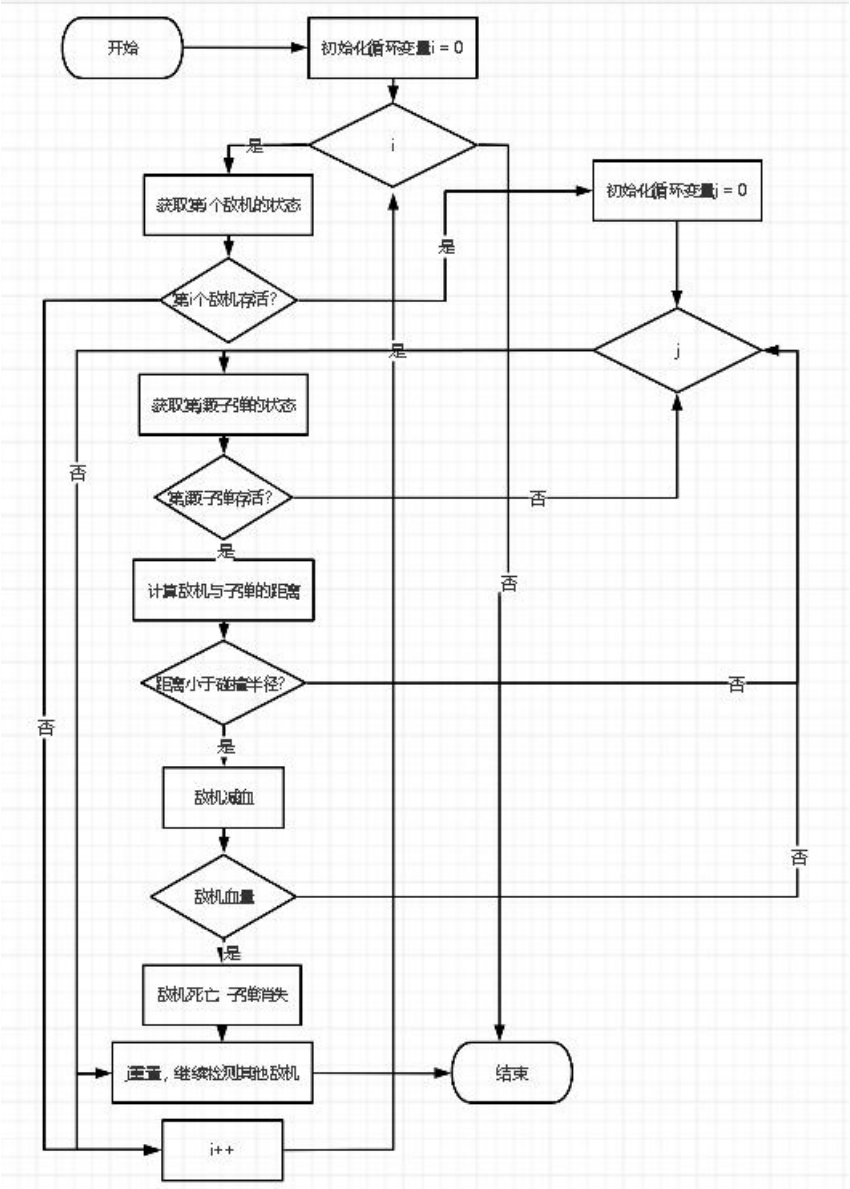


图 2-12 Attack()

Crash()

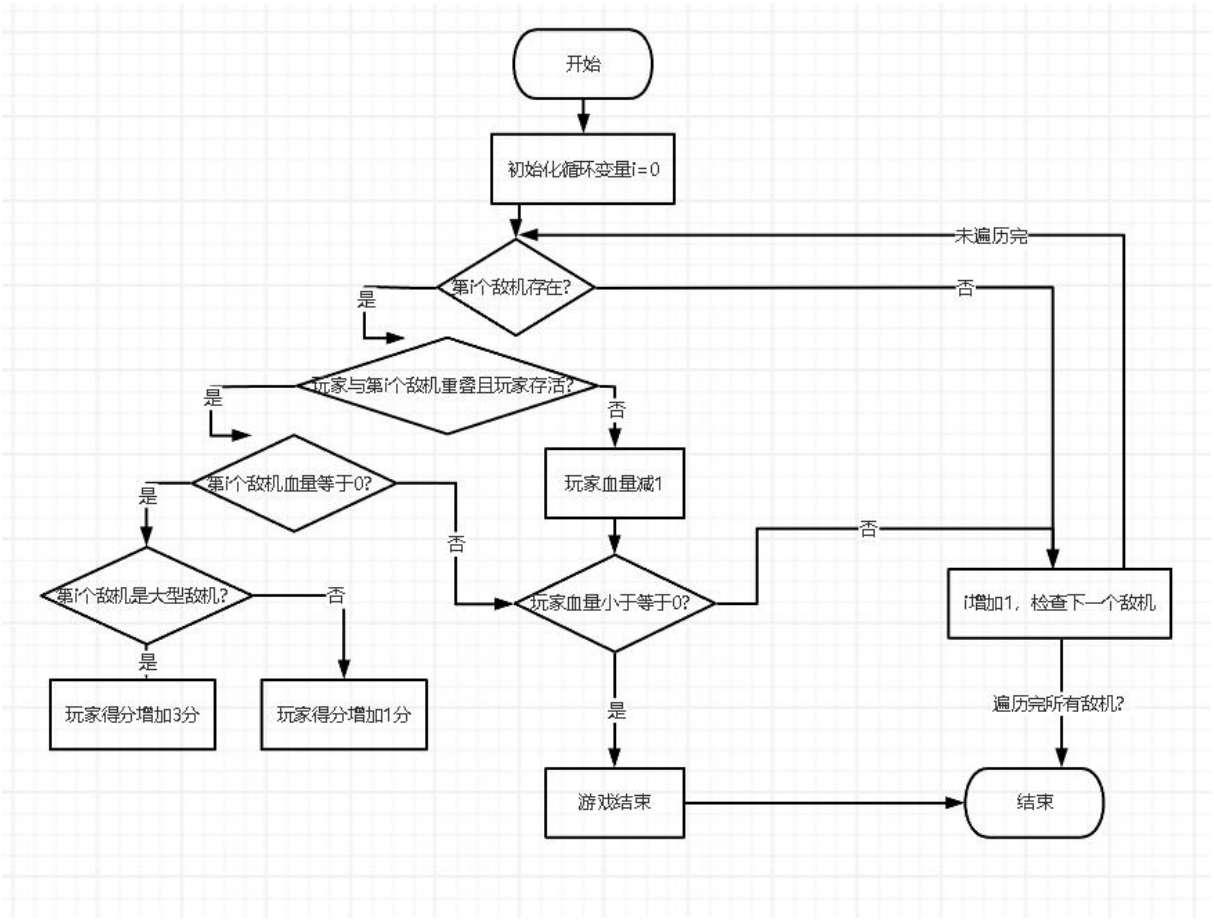


图 2-13 Crash()

3. 具体语言实现

```
#include "crash.h"
//距离判断
bool Distance(int i, int j, int type)
{
    if (type == 0)//判断飞机是否碰撞敌机
    {
        //中心距离判断
        return sqrt(pow(player.center_x - enemy[i].center_x, 2) + pow(player.center_y - enemy[i].c
enter_y, 2)) < player.radius + enemy[i].radius;
    }
    if (type == 1)//判断子弹是否击中敌机
    {
        return sqrt(pow(bullet[j].x - enemy[i].center_x, 2) + pow(bullet[j].y - enemy[i].center_y, 2))
 < enemy[j].radius;

    }
    return false;
}

// 子弹与敌机碰撞检测
void Attack()
{
    for (int i = 0; i < ENEMY_NUM; i++) {  // 遍历所有敌机
        if (!enemy[i].live) {  // 如果敌机已死亡，跳过当前敌机
            continue;
        }
        for (int j = 0; j < BULLET_NUM; j++) {  // 遍历所有子弹
            if (!bullet[j].live) {  // 如果子弹已死亡，跳过当前子弹
                continue;
            }
            // 检测子弹是否在敌机的半径范围内
            if (Distance(i, j, 1)) {//子弹在敌机的半径范围内，表示碰撞
                enemy[i].hp--;  // 敌机的血量减1
                bullet[j].crashEffect = true;
                bullet[j].live = false;  // 子弹消失
                if (enemy[i].hp == 0)//敌机的血量等于0，敌机消失
                {
                    if (enemy[i].type == BIG_ENEMY)
                    {
                        player.score += 3;//大敌机的血量等于0，得分加3
                    }
                    else
                    {
                        player.score += 1;//小敌机的血量等于0，得分加1
```

```
                    }
                    enemy[i].live = false;
                }
                break;  // 一旦碰撞，跳出内层循环，检测下一个子弹
            }
        }
    }
}
//玩家与敌机的碰撞检测
void Crash()
{
    for (int i = 0; i < ENEMY_NUM; i++)//遍历所有敌机
    {
        if (!enemy[i].live)//敌机不存在，跳过当前敌机
        {
            continue;
        }
        //检测玩家飞机是否与敌机重叠
        if (Distance(i, 0, 0) && player.live)//玩家飞机和敌机的碰撞
        {
            enemy[i].hp--;//敌机的血量减1
            player.hp--;//玩家飞机的血量减1
            if (enemy[i].hp == 0)//敌机的血量等于0，敌机消失
            {
                if (enemy[i].type == BIG_ENEMY)
                {
                    player.score += 3;//大敌机的血量等于0，得分加3
                }
                else
                {
                    player.score += 1;//小敌机的血量等于0，得分加1
                }
                enemy[i].live = false;
            }
            if (player.hp <= 0)//玩家血量为0，游戏结束
            {
                player.live = false;
            }
        }
    }
}
```

4. 复杂度分析

4.1 时间复杂度分析

1. 整体结构分析

main 函数中的无限循环（while (true)）表明程序会持续运行，除非手动终止或遇到某些控制条件。循环中调用了一系列的函数，我们需要分析每个函数调用的时间复杂度，并根据它们的执行方式来推算整体的时间复杂度。

2. 各主要函数调用的时间复杂度分析

1. **初始化函数**
   - initgraph(WIDTH, HEIGHT)、GetConsoleWindow()、SetWindowPos(hwnd, HWND_TOP, 800, 0, 0, 0, SWP_NOSIZE)：
     - 这些函数通常为图形或窗口初始化相关操作，通常为常数时间操作，时间复杂度是 **O(1)**。
2. **游戏初始化和背景音乐**
   - gameInit() 和 BGM()：
     - 这些函数执行的是一些初始化任务（例如设置玩家、敌机、子弹等），通常与游戏元素数量无关，因此时间复杂度为 **O(1)**。
3. **绘图函数**
   - BeginBatchDraw() 和 FlushBatchDraw()：
     - 这两个函数用于图形缓冲操作，通常是与图形绘制相关的操作，且开销较小，时间复杂度为 **O(1)**。
4. **游戏与文本绘制**
   - gameDraw() 和 textDraw()：
     - gameDraw() 的时间复杂度依赖于游戏元素数量，例如敌机、子弹数量，所以假设它的时间复杂度为 **O(ENEMY_NUM + BULLET_NUM)**。textDraw() 通常与绘制文本相关，其时间复杂度为 **O(1)**。
5. **游戏控制与状态切换**
   - start_game()、pause_resume() 和 death()：
     - 这些函数主要用于游戏状态的切换，不涉及大规模的数据操作，时间复杂度是 **O(1)**。
6. **计时函数**
   - timer(1000, 0)：
     - 这个函数用于计时操作，通常是基于系统时间进行比较，时间复杂度为 **O(1)**。
7. **敌机和子弹相关操作**
   - CreateEnemy()：
     - 该函数遍历所有敌机位置，最坏情况下需要遍历 **ENEMY_NUM** 个位置，因此时间复杂度为 **O(ENEMY_NUM)**。
   - EnemyMove()：
     - EnemyMove() 遍历所有敌机，并对每个敌机进行移动操作，时间复杂度为 **O(ENEMY_NUM)**。

- BulletMove()：
  - 该函数遍历所有子弹并进行移动操作，因此时间复杂度为 **O(BULLET_NUM)**。
- Attack()：
  - 该函数包含两层嵌套的循环，外层循环遍历所有敌机（**ENEMY_NUM**），内层循环遍历所有子弹（**BULLET_NUM**），因此时间复杂度为 **O(ENEMY_NUM × BULLET_NUM)**。

8. **碰撞检测**
   - Crash()：
     - 该函数遍历所有敌机进行碰撞检测，时间复杂度为 **O(ENEMY_NUM)**。

3. 综合时间复杂度分析

在每次循环内，执行的操作主要由以下几个部分的时间复杂度决定：

1. **与 ENEMY_NUM 相关的操作：**
   - CreateEnemy(), EnemyMove(), Crash() 等函数的时间复杂度为 **O(ENEMY_NUM)**。
2. **与 BULLET_NUM 相关的操作：**
   - BulletMove() 的时间复杂度为 **O(BULLET_NUM)**。
3. **与 ENEMY_NUM 和 BULLET_NUM 的乘积相关的操作：**
   - Attack() 函数的时间复杂度为 **O(ENEMY_NUM × BULLET_NUM)**。
4. **其他操作：**
   - 其他函数（如初始化、绘图、状态切换等）的时间复杂度均为 **O(1)**。

综合起来，每次循环的时间复杂度大致为：
$O(ENEMYNUM \times BULLETNUM + ENEMYNUM + BULLETNUM + 1)$

由于在渐进分析中，低阶项和常数项可以忽略不计，因此整体时间复杂度可以简化为：

$O(ENEMYNUM \times BULLETNUM)$

4. 结论

- 程序的时间复杂度在每次循环内是 **O(ENEMY_NUM × BULLET_NUM)**，这表示随着游戏中敌机数量（**ENEMY_NUM**）和子弹数量（**BULLET_NUM**）的增加，程序每次循环的执行时间将增加。
- 因为程序处于一个无限循环中，整体的时间复杂度与循环次数无关，仍然是 **O(ENEMY_NUM × BULLET_NUM)**，且游戏元素数量是影响程序性能的主要因素。

4.2 空间复杂度分析

1. 全局变量部分

- **图像相关变量：**

  - 代码中定义了多个 IMAGE 类型的变量，例如 bk（背景图片）、img_air[2]（飞机图片）、img_bullet[2]（子弹图片）、img_enemy[2][2]（大/小敌机的图片）、img_bullet_effect[2]（子弹击打特效图片）、img_down_effect[2][2]（敌机爆炸图片）、img_air_down[2]（玩家飞机爆炸图片）、img_start[2]（开始游戏图片）、img_puase[2]（暂停图片）、img_resume[2]（继续游戏图片）、img_gameover[2][2]（游戏结束图片）。
  - 这些变量用于存储游戏中各种图片资源，它们所占用的空间大小取决于图片本身的尺寸、格式以及图形库对图片存储的具体实现方式，但总体来说是固定的常量空间，与游戏运行过程中的动态元素数量（如敌机数量、子弹数量等）无关。可以将这部分空间复杂度看作 O(1)，因为它们的内存占用在程序启动后基本就固定下来了，不会随着游戏进程中游戏元素数量的变化而变化。
- **游戏元素结构体数组：**
  - 定义了 struct airplane player（玩家飞机结构体）、struct enemy enemy[ENEMY_NUM]（敌机数组结构体）、struct bullet bullet[BULLET_NUM]（子弹数组结构体）。
  - 对于 player 结构体，它只表示单个玩家飞机，其占用空间是固定的，可视为常数空间，复杂度为 O(1)。
  - 而 enemy 数组的空间大小取决于 ENEMY_NUM 的值，每个 enemy 结构体元素都有自己的属性（如坐标、血量、是否存活等），所以 enemy 数组总的空间复杂度与 ENEMY_NUM 成正比，为 O(ENEMY_NUM) 。
  - 同理，bullet 数组的空间复杂度与 BULLET_NUM 成正比，为 O(BULLET_NUM)。

2. 局部变量部分

在 main 函数内部的各个函数调用过程中，虽然会产生一些局部变量，但这些局部变量通常是在函数执行期间临时占用栈空间，函数执行结束后就会释放，并且它们的大小往往不依赖于游戏中主要的元素数量规模（敌机数量、子弹数量等），或者即使依赖也是常数级别的依赖（比如某个函数内部固定大小的临时数组等情况）。所以综合来看，这些局部变量对整体空间复杂度的影响相对较小，可以忽略不计，从渐近空间复杂度角度可将其视为 O(1)。

3. 综合空间复杂度分析

综合考虑全局变量和局部变量的空间占用情况，整体的空间复杂度主要由那些依赖游戏元素数量的结构体数组决定。由于存在 enemy 数组（空间复杂度为 O(ENEMY_NUM)和 bullet 数组（空间复杂度为 O(BULLET_NUM)，在渐近空间复杂度分析中，忽略常数项后，总的空间复杂度为 O(ENEMY_NUM+BULLET_NUM)，表示程序运行时所占用的空间主要由敌机数量和子弹数量决定，随着这两个数量的增加，程序占用的内存空间也会相应线性增加。

5. 系统测试

对主要模块给出预期结果，进行运行，是否与预期一致，如果产生错误如何解决。

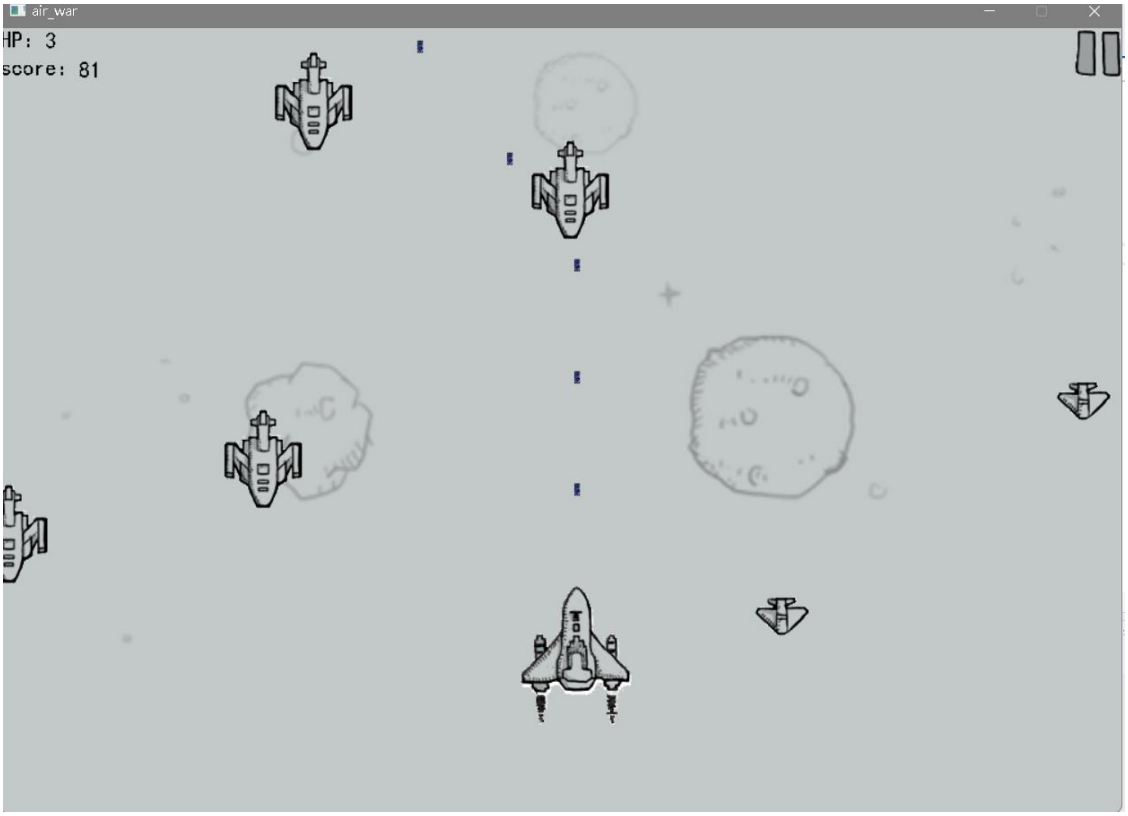


图 3-1 运行图

1.按住空格子弹发射时，不产生敌机的问题。

需要在创建子弹方法里面加上创建敌机的逻辑。

```
//子弹的创建
void CreateBullet()
{
    static DWORD t1 = 0, t2 = 0;

    for (int i = 0; i < BULLET_NUM; i++) {
        if (!bullet[i].live) {  // 子弹未发射
            bullet[i].x = player.center_x;  // 子弹的横坐标
            bullet[i].y = player.center_y - player.radius;  // 子弹的纵坐标
            bullet[i].live = true;  // 射出子弹后，子弹状态为已发射

            // 创建敌机的逻辑,防止进入按键检测不能创建敌机
            if (t2 - t1 > 1000) {  // 每 1 秒创建一次敌机
                CreateEnemy();
                t1 = t2;
            }
            t2 = clock();

            break;  // 产生一个子弹就退出
        }
    }
}
```

图 3-2 子弹创建

2.实现碰撞检测时需要确定碰撞面积

```
    // 获取鼠标事件
    if (GetAsyncKeyState(VK_LBUTTON) && !player.live ) {//按下鼠标左键

        POINT p;//鼠标坐标
        HWND h = GetForegroundWindow();//获取当前窗口
        GetCursorPos(&p);//获取鼠标坐标
        ScreenToClient(h, &p);//将屏幕坐标转换为窗口坐标

        // 打印调试信息
        /*char infox[20];
        char infoy[20];
        sprintf_s(infox, "%d", p.x);
        sprintf_s(infoy, "%d", p.y);
        outtextxy(80, 1, infox);
        outtextxy(300, 1, infoy);*/

        // 显示鼠标位置的图像
        putimage(p.x - 10, p.y - 10, &img_bullet_effect[0], NOTSRCERASE);
        putimage(p.x - 10, p.y - 10, &img_bullet_effect[1], SRCINVERT);
```

图 3-3 鼠标获取

用鼠标点击的位置来获取坐标，确定飞机大小(碰撞面积)

6. 总结

一、课程设计目的

本次课程设计旨在综合运用所学的编程知识与技能，深入理解游戏开发的完整流程，从游戏的初始化、元素创建、运动控制、碰撞检测到界面绘制与音效处理等各个环节进行实践。通过实际动手操作，培养解决复杂问题的能力，学会如何将不同的功能模块有机整合，以实现一个具有一定可玩性和交互性的游戏程序。同时，提升对代码结构、算法效率以及资源管理等方面的关注度，为今后从事相关软件开发工作或进一步深入学习游戏开发技术奠定坚实基础。

二、用到的知识与技术

在本次课程设计中，运用了多方面的知识与技术。首先，在图形绘制方面，使用了图形库相关函数来加载和显示各种图片资源，包括背景、飞机、子弹、敌机等图像元素，以构建游戏的可视化界面。其次，数据结构方面，定义了结构体来表示玩家飞机、敌机以及子弹等游戏对象，这些结构体包含了诸如坐标、速度、血量、存活状态等属性信息，方便对游戏元素进行管理和操作。再者，通过计时函数来控制游戏中一些事件的触发频率，例如定时创建敌机、控制子弹发射间隔等，实现游戏的节奏把握。另外，还涉及到了简单的音效播放函数调用，为游戏添加背景音乐、子弹射击音效以及碰撞爆炸音效等，增强游戏的沉浸感和趣味性。在算法层面，运用了碰撞检测算法，通过计算不同游戏元素之间的距离来判断是否发生碰撞，并依据碰撞结果进行相应的游戏逻辑处理，如减少血量、增加得分、触发爆炸效果等。

三、学习收获

通过本次课程设计，我学到了许多宝贵的知识和技能。在编程实践能力上有了显著提升，对于复杂程序的架构设计和代码组织有了更深入的理解。学会简单使用 PS 制作掩码图，学会了如何合理地划分功能模块，使得代码结构更加清晰、易于维护和扩展。例如，将游戏初始化、绘制、控制、

碰撞检测等功能分别封装在不同的函数和模块文件中，提高了代码的可读性和可复用性。在算法方面，对时间复杂度和空间复杂度的分析有了更直观的认识，明白了在编写代码时如何优化算法以提高程序的性能。在处理游戏元素的运动和交互过程中，进一步掌握了坐标系统和数学计算在编程中的应用，能够更加熟练地运用逻辑判断和循环结构来实现复杂的游戏逻辑。此外，还学会了如何利用图形库和音效库来丰富游戏的表现形式，增强用户体验，了解到多媒体元素在软件开发中的重要性以及如何将它们与程序逻辑紧密结合。

四、课程设计心得

在整个课程设计过程中，我深刻体会到了理论知识与实践相结合的重要性。尽管在之前的学习中已经掌握了许多编程概念和技术，但在实际应用到游戏开发项目中时，仍然遇到了诸多挑战。从最初的游戏规划和设计，到代码的编写、调试以及不断优化，每一个环节都需要耐心和细心。在解决各种问题的过程中，我学会了如何利用调试工具快速定位错误，如何查阅相关文档和资料获取技术支持，以及如何与同学和老师进行交流讨论，拓宽解决问题的思路。通过不断地尝试和改进，最终看到自己开发的游戏能够顺利运行并具有一定的可玩性，心中充满了成就感。同时，也认识到自己在编程能力和软件开发经验方面还有很大的提升空间，例如在代码优化、资源管理和用户界面设计等方面还可以做得更好。这次课程设计不仅是一次技术上的锻炼，更是一次对自己学习能力和解决问题能力的考验，为我今后的学习和工作积累了宝贵的经验，激励我在软件开发领域不断探索和前进。