摘　要

本坦克大战游戏是用 C++编写的基于 SFML 图形库的程序，这是一款单机单玩家的游戏，正常实现了绝大部分游戏功能，比如移动、开火、击毁敌人和自动生成敌人等。

关键词：C++、SFML 图形库、面向对象程序设计

目 录

第一章 绪 论

1.1 项目的基本要求

坦克大战游戏要求能实现己方坦克正确移动以及开火，敌方坦克的自动寻路以及自动开火，并实现地图的导入，要求地图上有不同类型的障碍物，同时坦克和子弹不能穿过障碍物，障碍物有一定的坚固程度，能被子弹击碎。最后还要求能击毁敌方坦克，或者敌方坦克能摧毁己方坦克。

1.2 本论文的结构安排

第二章将重点分析图形界面板块的相关问题，比如如何实现图片导入，地图导入，图形移动，图形旋转等等。

第三章将重点介绍逻辑功能板块的内容。比如怎样写坦克类，怎样通过继承的方法使用现有函数和简化代码量，怎样实现自动生成坦克功能、坦克开火功能、坦克移动功能、坦克寻路功能，怎样实现子弹飞行、子弹打击功能等等。

第二章 图形界面的实现

图形界面作为逻辑程序外化和可视化最重要的一步，在整个工程中占有举足轻重的地位，而现有的 SFML 图形库直接提供了大量丰富而强大的 API 接口，很大程度地为程序员提供了方便。本章将重点介绍如何利用 SFML 图形库实现自己的坦克大战图形界面。

2.1 游戏循环 game loop

首先，与普通的 C++程序不同，绝大部分游戏都是一个动态的程序，而不是在输出某些结果后就程序结束。对于游戏这类程序，理论上只要游戏未结束，程序就会一直不停地运行，实质是程序一直在一个称为 game loop 的 while()死循环里面不停地运行。

一个典型的 game loop 如图 2-1 所示：

```
    /*---------------- game loop ---------------*/
while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed)
            window.close();
    }

    window.clear();
    /*---Update game---*/
    window.display();

}
```

图 2-1

我们可以很轻易地发现，每次程序在循环里面执行一次（称为一帧），相应的变量就会进行一次改变（比如说坦克、子弹），它们或者移动或者旋转或者消失等。然后由于循环还在继续，所以变量还会继续改变，这样原本间断的改变，在保持 60FPS 即每秒刷新 60 次的频率下，坦克的移动、子弹的飞行都会变成连续的，让图形界面程序就变得像动画一样连续流畅了。

所以，我们可以得到启示：把最开始窗口的初始化、坦克的初始化、地图

的导入等等放在 game loop 外面，而把坦克的移动、开火，子弹的移动，碰撞判断，画界面等逻辑功能放在 game loop 里面，让它们保持不断更新，从而实现的更新。

程序的逻辑框图如图 2-2 所示：

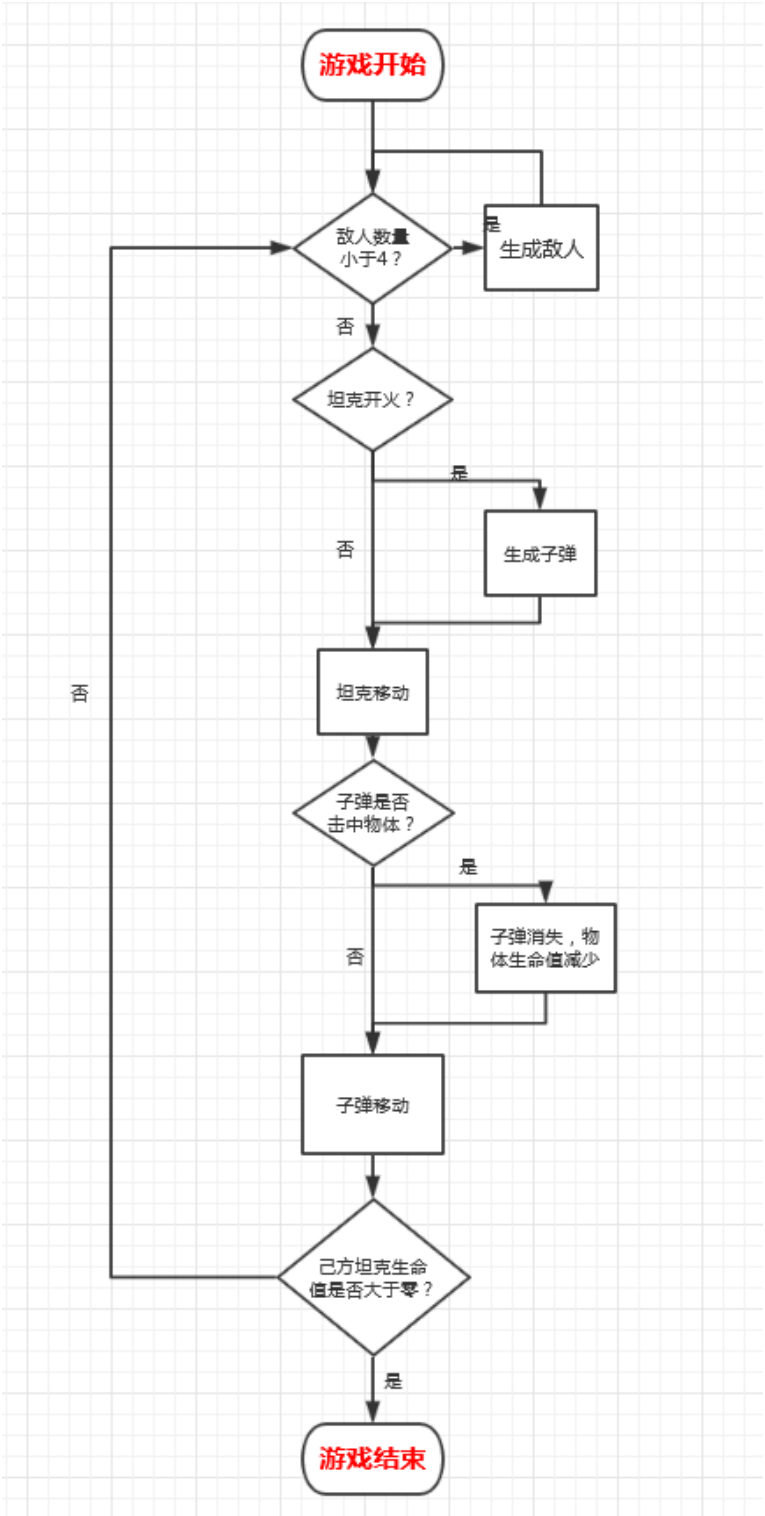


图 2-2

2.2 精灵类 Sprite 与纹理类 Texture

在 sfml 图形库里面，精灵类 Sprite 和纹理类 Texture 都与图形界面直接相关，sf::Texture 更底层，主要用于从磁盘导入图片等功能；而 sf::Sprite 更接近上层， 它包含了更多关于图形操作的函数，比如图片的旋转和移动。

2.2.1 纹理类 Texture

在 sfml 图形库里面，与图片直接相关的类是 sf::Texture，当我们想要从磁盘中导入一张现有的图片时，一般就声明一个 Texture 类的对象 texture，然后调用该对象的 loadFromFile 函数，写出图片在磁盘中的路径就可以了。

如图 2-3 所示：

```
sf::Texture texture_tank;
texture_tank.loadFromFile("E:\\VS Files\\SFML_Programming\\MyTank.jpg");
```

图 2- 3

上图就从 E 盘的 VS Files 目录下的 SFML_Programming 目录中，导入了一张坦克的图片给 texture_tank 对象，在这个例子中，我们就成功地导入己方坦克的图片，它是一个 50*50 像素的图片，实际效果如图 2-4 所示：

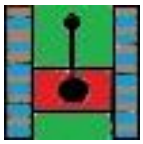


图 2-4

同理考虑到游戏的需要和图形界面的美观，我们可以用相同的方法为敌方坦克、己方子弹、敌方子弹都读取各自的图片，本程序使用的图片分别如图 2-5、图 2-6、图 2-7 所示：

图 2-5

图 2-6

图 2-7

2.2.2 精灵类 Sprite

在 sfml 图形库里面，与图片操作直接相关的类是 sf::Sprite，该类的提供了

大量功能强大的成员函数，比如读取纹理 texture 函数，设置和返回图片位置、图片缩放比例、图片边界范围、图片旋转中心、图片旋转和平移函数。

例如在本程序中对己方坦克 tank 进行初始化的时候如图 2-8 所示：

```
tank.setOrigin(25, 25);
tank.setPosition(125, 125);
tank.setTexture(texture_tank);
```

图 2-8

对于上图，由于坦克图片大小是 50*50 的，所以第一个函数 setOrigin(25,25)就将坦克的旋转平移中心设定在了实际图片的中心；而第二个函数 setPosition()就自然是在设置坦克在地图上的初始位置；第三个函数 setTexture()就将该 tank 对象导入之前已经读取的坦克图片，实现逻辑变量和图片文件的对接。

在这里，我才深深地体会到 C++面向对象设计和继承的强大优越性：当我们想让逻辑上的一个坦克变量在图形界面上移动时，只需要在定义坦克类的时候，让它继承 sf::Sprite。如图 2-9 所示：

```
class Tank :public sf::Sprite {

private:
    //......
public:
    //......

};
```

图 2-9

这样，该坦克对象就能调用所有与图片操作相关的函数了！我们不必关心这些函数的底层实现原理，只要知道它们的用法，就可以挪为己用了，继承这一特性，大大地降低了编程难度，让我们有更多的精力去操心逻辑功能的部分，从而更合理地分配自己的时间。

2.3 地图的导入

对于普通的游戏来说，地图或许就是一张背景图片，不需要考虑物体和地图的碰撞和交互，那么这种情况下导入地图就是一件很简单的事情，我们只需要找到一张大小合适的图片，每次都在 game loop 的最开始把它画出来，其次再画出物体，由于后画出的图片会盖在之前画出来的图片上，所以这样就能实现简单的

背景地图的导入。

但是，该坦克大战不同的是，根据游戏的实际需求，我们的地图不能仅仅是停留在图片上的“虚背景”，而是必须做成能与坦克对象进行碰撞和交互的具有逻辑功能的“实实在在”的地图。所以，上面第一种方法是行不通的。

到这里，想要生成具有实际逻辑功能的“活地图”。或许很自然的，我们又有一种猜想：对 16*16 格的地图的每一格都手动地进行定义和初始化。如图 2-10 所示：

```
class Map :public sf::Sprite {

private:
    int x;
    int y;
public:
    //. . . . . .

};
```

图 2-10

比如我们定义一个继承于 Sprite 的地图类 Map，然后类似于之前 tank 的操作，对地图上的某一格进行导入图片、坐标初始化等等。通过这种方法，的确，地图“活”起来了，我们可以很方便地让地图 Map 和 tank、bullet 进行碰撞判断和交互。但是，这种方法有两个致命的缺陷：

第一、对于 16*16 格的地图而言，初始化就要进行 256 次，地图上每一格又可能是不同的障碍物比如河流、泥土、树木，于是初始化的代码量将是无法估量且难以维护和简化的。

第二、该算法会导致设置过多的 Texture 和 Map 类的对象，这对于每次循环会处理大量信息的程序来说是不利的，我们希望尽可能地减少 Texture 的数量，否则当程序很庞大时，流畅性会受到影响。

所以我们希望的是：通过一个数组一次性地导入整张地图的所有信息，用数组元素的取值不同来区分地图上的不同障碍物的类型，比如通过以下代码实现地图的导入，如图 2-11 所示：

```
/*   0草地 1河流 2树木 3泥土     */
int Map::tilesArray[256] = {
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,1,1,1,1,1,1,0,0,0,0,2,0,0,0,2,
    2,1,0,0,0,0,0,0,0,0,0,0,0,3,0,2,
    2,1,0,0,0,0,0,3,3,0,0,0,1,0,0,2,
    2,1,1,0,3,3,3,0,0,0,0,0,1,2,0,2,
    2,0,1,0,0,0,2,0,0,0,0,0,1,1,2,2,
    2,0,0,0,0,0,2,2,2,0,0,0,0,0,1,2,
    2,0,0,0,0,2,2,2,0,0,3,3,1,1,1,2,
    2,3,3,0,3,3,0,0,3,3,3,3,3,2,0,2,
    2,0,1,0,3,0,0,2,0,0,1,1,0,0,2,2,
    2,0,1,0,0,0,0,2,2,0,1,0,0,0,1,2,
    2,0,1,0,3,0,0,2,0,0,0,0,0,0,1,2,
    2,1,1,0,3,3,3,0,0,0,0,0,0,0,0,2,
    2,0,1,3,0,0,0,0,0,0,0,0,0,3,0,2,
    2,0,1,0,0,0,0,0,0,0,0,0,0,0,0,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
};
```

图 2-11

我们让 0 代表草地，1 代表河流，2 代表树木，3 代表泥土。这样，该数组就包含了整张地图的所有信息。

那么，有没有可能让这种猜想变为事实呢？答案是：可以！在这里，隆重推出“TileMap”地图导入算法！通过这个算法，我们的 Texture 可以降低为 1 个，如下图 2-12 所示：

图 2-12

这张被称为“tileset”的小图片包含了四种不同的地形，每种地形只在迷你地图上出现一次。随之衍生出了一种新的算法：先在上面的数组中取得第 i 行第 j 列那一格处 tilesArray 数组的元素的值，比如说是 2（代表树）；于是根据算法在上面 tileset 迷你地图中取出左下角的树的那一小部分图片，打印到屏幕上。再通过 i,j 两层循环，就能很简便地把整张地图都打印下来了，并且只用了一个 Texture 变量。同时，数组元素不同的取值可以区分不同的障碍物类型，方便以后实现碰撞检测等逻辑功能。

以下是通过该算法生成的具有实际逻辑意义的“活”地图，函数的参数是上面的 tilesArray 数组和 tileset 迷你地图，如图 2-13 所示：

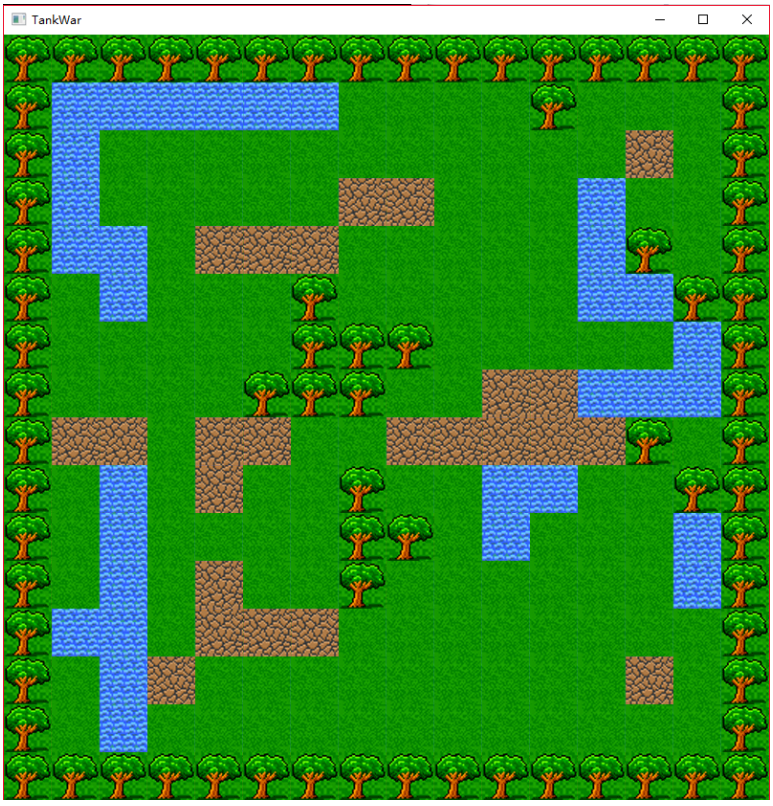


图 2-13

2.4 本章小结

本章从大体上介绍了坦克大战游戏的图形界面的实现原理。实现了坦克、子弹的图片导入，具有逻辑意义的地图的导入。但是，此时的坦克和子弹都可以穿墙，因为并没有实现碰撞检测也没有实现对象之间的交互，目前所实现的只有最基础的导入功能。

图形界面的实现原理也大概就介绍完了，然而，想实现比如坦克开火、移动，子弹飞行，坦克不能穿墙等更多符合实际情况的、有意思的功能，仅仅靠图形版块的函数时无法实现的。为此，我们将在第三章介绍这些复杂的但功能强大的逻辑层面的函数。

第三章 游戏逻辑功能的实现

仅靠基本图形界面是绝对不够的，本章开始将介绍该坦克大战游戏中的碰撞检测，坦克移动、开火，子弹飞行和命中敌人等逻辑功能的实现，同时将介绍各种坦克、子弹和他们的父类的继承关系等等。有了这些功能，该游戏才真正地具有现实意义和生命力。

3.1 父类 Object

考虑到不管是坦克还是子弹，它们都不能走出窗口的边界，都不能穿过树木等障碍物，而且需要实现这些功能难度都不小，代码量都挺大。考虑到以上问题，也为了增强代码的简洁性和可维护性，把子弹和坦克都作为 Object 的子类。类图如图 3-1：

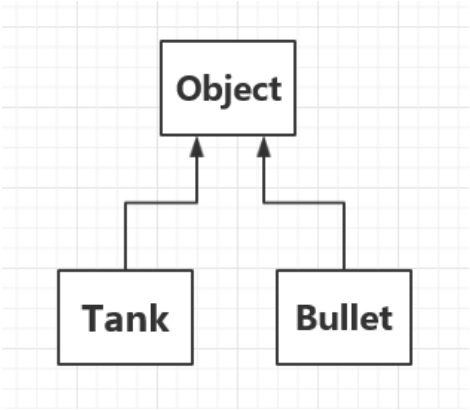


图 3-1

```
class Object : public sf::Sprite {

private:
    int speed;

public:
    void setObstacleSpeed(int sp) { speed = sp; }
    int getObstacleSpeed() { return speed; }

public:
    /*判断是否走到边界*/
    bool ifLeftBoundary();
    bool ifRightBoundary();
    bool ifUpBoundary();
    bool ifDownBoundary();

public:
    /*判断前方一个墙的接近程度*/
    int ifLeftReach();
    int ifRightReach();
    int ifUpReach();
    int ifDownReach();

public:
    /*判断当贴墙时前面是否有障碍物*/
    bool ifLeftObstacle();
    bool ifRightObstacle();
    bool ifUpObstacle();
    bool ifDownObstacle();

};
```

图 3-2

于是，Object 类需要实现的只是坦克和子弹共有的属性（比如移动速度）和逻辑功能比如边界判断和碰撞判断，Object 类的声明如图 3-2。

父类定义好了之后，就可以定义具有不同性质的派生类了，比如派生出子弹和坦克两个类。

3.1.1 子类 Tank & Enemy

考虑到父类 Object 已经将底层共有的碰撞检测等功能都实现了，上层的特异性功能由子类实现。对于己方坦克，它的移动和开火都是由玩家控制；而对于敌

```
class Tank : public Object {

private:
    int health;
    int damage;
    char direction;
    int fireController;

public:
    void setDamage(int da) { damage = da; }
    void setHealth(int he) { health = he; }
    void setDirection(char di) { direction = di; }
    void setFireController(int fi) { fireController = fi; }
    void controllerAdd() { fireController++; }

public:
    int getHealth() { return health; }
    int getDamage() { return damage; }
    int getFireController() { return fireController; }
    char getDirection() { return direction; }

public:
    void tankMove();
    void tankFire();
    void tankAction();
    Tank(int heal, int da, char di, int fi);

};
```

图 3-3

```
class Enemy : public Object {

private:
    int health;
    int damage;
    char direction;
    int bulletControll;

public:
    void setHealth(int heal) { health = heal; }
    void setBulletControll(int n) { bulletControll = n; }
    void ControllerAdd() { bulletControll++; }

public:
    char getDirection() { return direction; }
    int getDamage() { return damage; }
    int getHealth() { return health; }
    int getBulletControll() { return bulletControll; }

public:
    void enemyMove();
    void enemyFire();
    void enemyAction();
    Enemy(int heal, int da, char di, int fi);

};
```

图 3-4

方坦克，它的移动是由其寻路算法实现，开火是当检测到附近有敌人的时候会启动。所以比较合适的选择是把己方坦克 Tank 和敌方坦克 Enemy 都并列写为 Object 的子类。如图 3-3 和图 3-4：

先看看 Tank 子类吧，己方坦克有一定生命值，当被子弹击中后会减少，生命值小于等于零时游戏就结束了。同时，当它开火发射子弹时，子弹的伤害值 damage 就直接取自 Tank 类的 damage 成员。它的 direction 数据成员改变时，会调用底层的图形界面相关的函数对图片进行旋转，同时当前的 direction 取值还会直接决定开火生成的子弹的运动方向。

它的 fireController 成员实际是一个用于控制子弹频率的一个计数器，因为在实际调试过程中发现，如果不对子弹频率进行约束，将会导致发射子弹过于密集以至于连成一条线。如果每次按下开火键的时候计数器 fireController 就加一，但是并不会立刻发射子弹，只有当 fireController 是 30（也可以自己选择一个合适值）的倍数的时候才发射，这样就能让子弹的发射频率处于一个合理的范围。以下是加入 fireController 计数器前、后的对比如图 3-5 和 3-6：

图 3-5

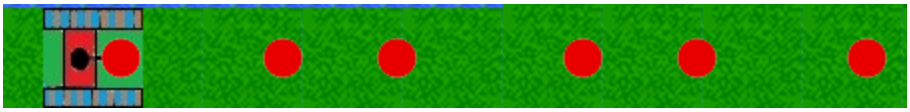


图 3-6

同理，Enemy 子类的大部分代码和 Tank 子类的代码是差不多的，区别只是在于它们移动和开火的激活方式是不一样的：Enemy 的是由系统根据己方坦克的位置而自动执行的，而 Tank 的是由玩家按键盘进行手动控制的。

谈到玩家通过键盘手动控制坦克，这里其实又会用到 sfml 图形库提供的键盘监听的函数 sf::Keyboard::isKeyPressed()函数。该函数的方便之处在于它是对键盘进行实时监听，这样如果玩家根据即时的游戏情况控制坦克，实时监听就会保证他所按下的操作能够以最快的速度反馈给程序，并展示到屏幕上。这样一来，游戏的可玩性就大大增加了，游戏的延迟得以降低到最小。

3.1.2 子类 Bullet

当坦克相关的代码写完了之后，另一个大板块就是关于子弹类 Bullet 的了。同样的道理，父类已经将底层的碰撞检测、边缘判断等逻辑功能实现了，于是在 Bullet 子类里面就只需要考虑属于子弹的特异性的功能了。

Bullet 子弹类的声明如图 3-7：

```
class Bullet : public Object {

private:
    int damage;
    char direction;

public:
    Bullet(int da, char di);
    int getBulletDamage() { return damage; }
    char getBulletDirection() { return direction; }

public:
    void bulletFly();
    void bulletVanish();

};
```

图 3-7

不难发现，Bullet 子类的函数其实挺简单，数据成员就只有两个：伤害值和飞行方向。需要重点写的就是子弹飞行的函数和子弹消失的函数。在这里我进一步体会到了 C++的两大特性：面向对象和继承。正因为有了继承的思想，我才会想到把子弹和坦克还有敌人都作为 Object 的子类，这样做之后，我可以直接避免把功能相同但属于三个不同类的函数重复写三遍，这极大地简化了代码量，同时也使得 Tank 子类和 Bullet 子类的代码量减少，因为共性的功能已经在父类实现了，现在就只需要写特异性的功能了；正是因为有了面向对象的思想，我们才能将如此庞大的一个上千行的程序代码分解成多个小的板块，并且每个小板块之间是相对独立的，这大大降低了编程难度。

到这里，己方坦克、敌人、子弹这三大主体的基本功能都已经实现，总结一下它们的关系，类图如图 3-8：

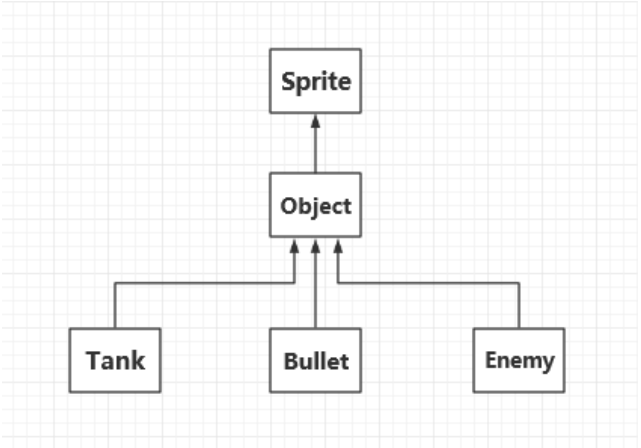


图 3-8

3.2 Universe 类与动态数组 Vector< >的妙用

程序的主体功能在上一节其实就已经完工了，但是现在还存在一些悬而未决的事情没有解决，这些事情必须得先解决掉，不然会对后面写一些函数造成很大的选择性上的难题。

那么这个问题是什么？答案就是关于子弹和坦克的储存问题！因为对于每一帧来说，我们都得需要对每个坦克、每颗子弹进行判断，看它们是否有撞墙的现象，如果有，就必须得让它停止运动。所以，我们需要时时刻刻对所有坦克和子弹进行遍历。所以我们就必须把它们都储存起来。

那么又有一个问题来了：应该选择什么合适的数据结构来对它们进行储存呢？最容易想到的是用数组来存储（比如把所有的子弹放进一个子弹数组，把所有的坦克放进一个坦克数组），数组的确可以方便我们对子弹和坦克进行遍历。但是，选择数组又有一些问题。一、比如我们必须得事先就确定下数组的长度大小，如果长度太小会导致数组元素溢出，长度太大又会过度浪费内存。二、更严重的是，对于坦克大战这个游戏来说，子弹生成速率是非常快的，消失速率也是很快的，只要当子弹碰到障碍物或者敌方坦克时它就会消失。对于数组来说，经常执行删除操作的开销也是十分惊人的。综合以上两点考虑，用数组储存对象不可行。

于是我们想到了用链表来储存坦克和子弹。因为链表的长度是动态的，可以根据需要随时进行调整，能最大可能地节省内存；同时，链表在频繁地遍历并执行删除操作的时候，开销也只是 O(n)。所以，链表是不二之选。

幸运的是，C++刚好提供了一个 Vector<>动态数组模板（虽然它名字叫数组，但其实功能更类似于链表），有了它我们就可以直接利用现有的模板，实现链表的各种功能了。于是我就不用再去重新写一套链表相关的整套函数了。

在 C++使用 Vector<>动态数组模板时，需要提前包含相应的头文件，写上#include <vector>即可。这里还有最后一个问题：我们把这些动态数组定义成什么范围的变量呢？局部的还是全局的？考虑到很多类都会访问这些动态数组，所以写成局部变量是肯定不可行的。于是似乎把它声明为全局变量会方便很多，各个类都可以对动态数组直接访问。但是，全局变量会破坏 C++程序的封装性和安全性，应该尽可能少的使用全局变量。

于是，我们可以用类对动态数组进行封装：我把这个类取名为 Universe，它里面没有 private 数据成员，也没有 public 成员函数，唯一有的就是作为 public 的动态数组。我把己方和敌方的坦克、子弹都分别存在一个动态数组里面（把己

方、敌方子弹分开储存是防止敌方坦克互相残杀..）如图 3-9：

我们会惊喜地发现，由于 Universe 类里面的动态数组是 public 属性的，所

```
class Universe {

public:
    static std::vector<Tank> MyTank;
    static std::vector<Bullet> BulletBuf;
    static std::vector<Enemy> EnemyBuf;
    static std::vector<Bullet> EnmeyBulletBuf;

};
```

图 3-9

以我们可以在其他任何类里面对它进行访问，访问之前只需要 #include <Universe.h>就行了，于是这间接地实现了全局变量带来的方便。更重要的是，它并不是全局变量，不会破坏程序的封装性，相反，它作为一个类，很好地实现信息的屏蔽和隔离。简直就是两全其美的做法！

3.3 子弹命中坦克功能的实现

根据游戏的需要，我们必须在每一帧的时候，都去检查，是否有任意一颗子弹命中任意一个敌人。所以我们需要两层循环去遍历所有的坦克和所有的子弹以及他们之间的碰撞交互情况。考虑到这个功能是普遍性的函数，并不针对某一个特定的类或者某一个特定的对象，该函数每次执行就一定会涉及到坦克类和子弹类的所有对象，所以一个合适的选择是：不把它作为成员函数，就仅仅写为一个普通的函数（前文已交代，Universe 类的封装使得任何函数都可以像访问全局变量那样去访问动态数组的元素）。类似的函数还有自动生成敌人的函数。因为该函数不依赖于某个具体对象的值。所以没必要写为成员函数。该函数关心的只是外部的信息，比如敌方坦克的数量，只要敌方数量小于一个特定值的时候就开始自动生成新的敌人。

这三大功能的非成员函数如图 3-10，这三个函数分别实现检测敌方坦克是否中弹、己方坦克是否中弹、自动生成敌人：

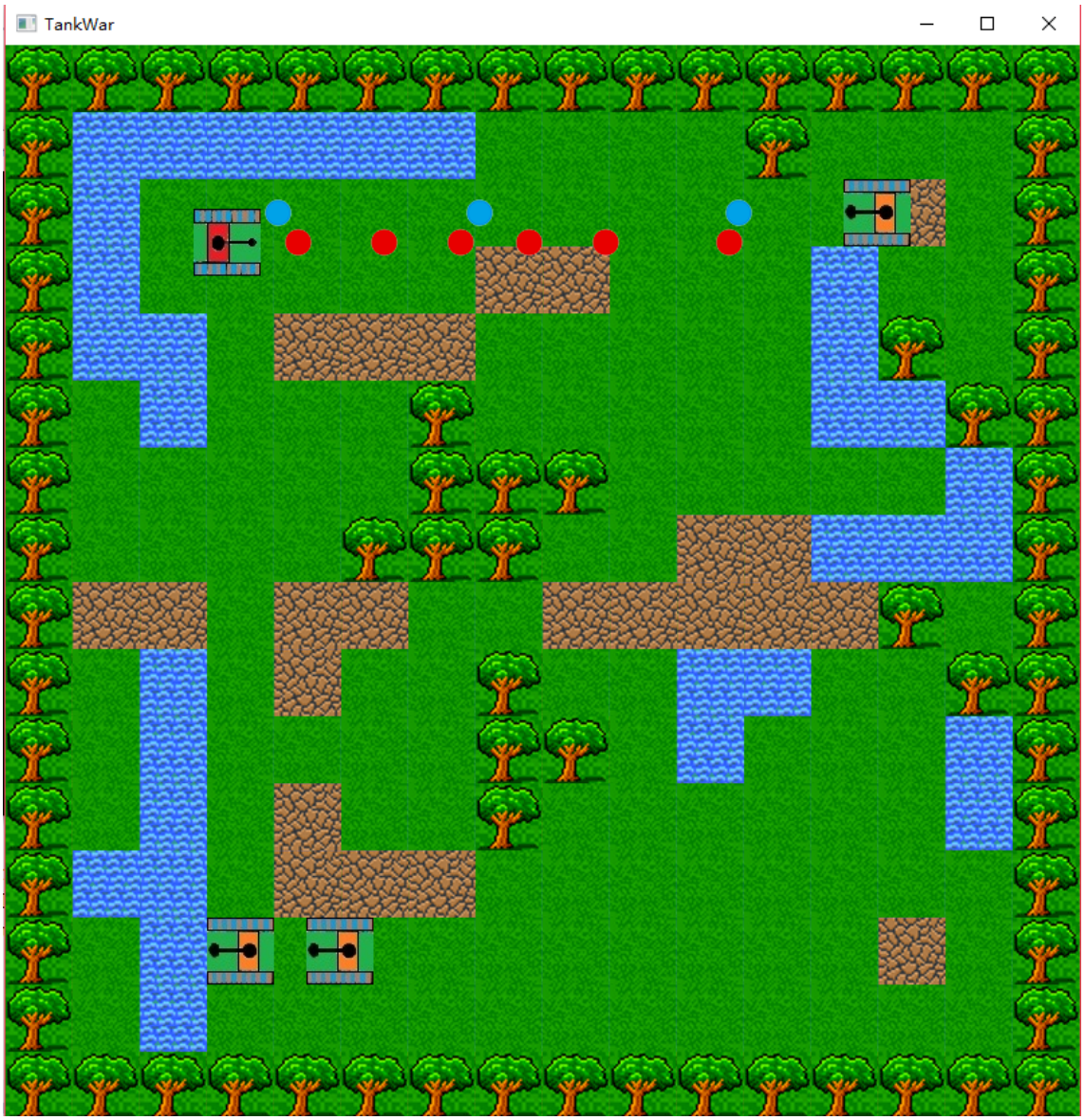
```
void CheckEnemyAttacked() { ... }

void CheckMeAttacked() { ... }

void CreateEnemy() { ... }
```

图 3-10

3.4 本章小结

本章重点介绍了坦克大战游戏在逻辑功能层面上的实现。只有当真正地逻辑功能与图形界面相互结合的时候，这个游戏才是完整且符合实际的。图形界面的使用让我们的游戏变得可视化和更有趣味性，不过这只是表层的现象。实际上能支撑表层的图形界面的东西往往是底层的逻辑功能，而往往编写难度最大的地方也就是在逻辑功能这一地方。图形界面层和逻辑层既互相分离，又互相关联，二者息息相关相互促进。

在最后附上一张实战截图：



第四章 全文总结与功能测试以及展望

4.1 全文总结

本论文大体从两个方面剖析了该坦克大战游戏的实现。第一方面是从图形界面的角度进行论述，重点讲了图形界面的实现原理和 sfml 图形库的使用方法；第二方面是从论述游戏逻辑功能上的实现，重点讲了如何设计各类之间的继承派生关系，以及开火、移动和碰撞检测等功能的实现。

4.2 游戏功能测试

游戏后期测试也是程序设计必不可少的一环，本节将对该坦克大战游戏的逻辑功能进行测试，并给出程序截图。

4.2.1 坦克碰撞检测

当坦克遇到障碍物比如墙的时候，理论上就不能再继续前进了，我们操纵坦克到达墙角，如图 4-1：

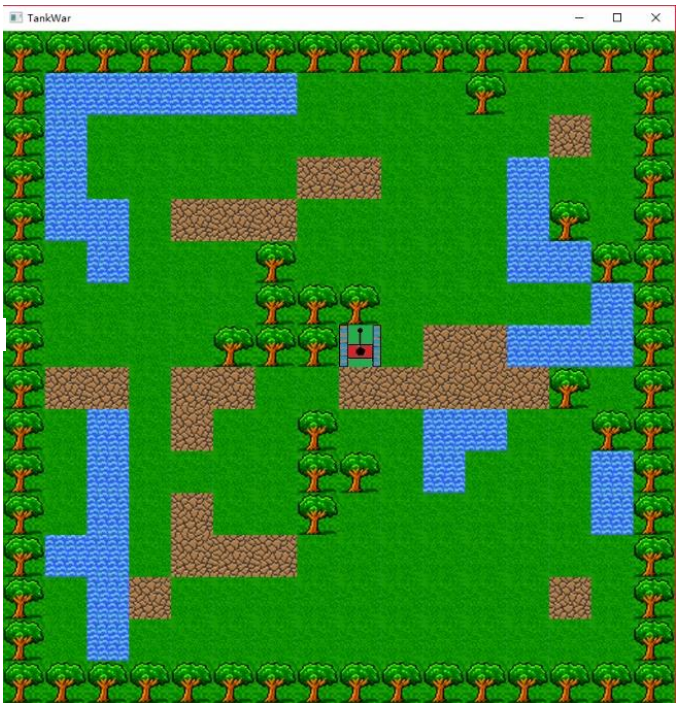


图 4-1

继续按左键或者上键，发现坦克不移动。测试合格!!

4.2.2 子弹碰撞检测

当飞行的子弹遇到障碍物（比如墙）的时候，理论上不能再继续飞行了，而且应该消失。我们让坦克开火发射子弹，子弹遇到障碍物前的状态，如图 4-2：

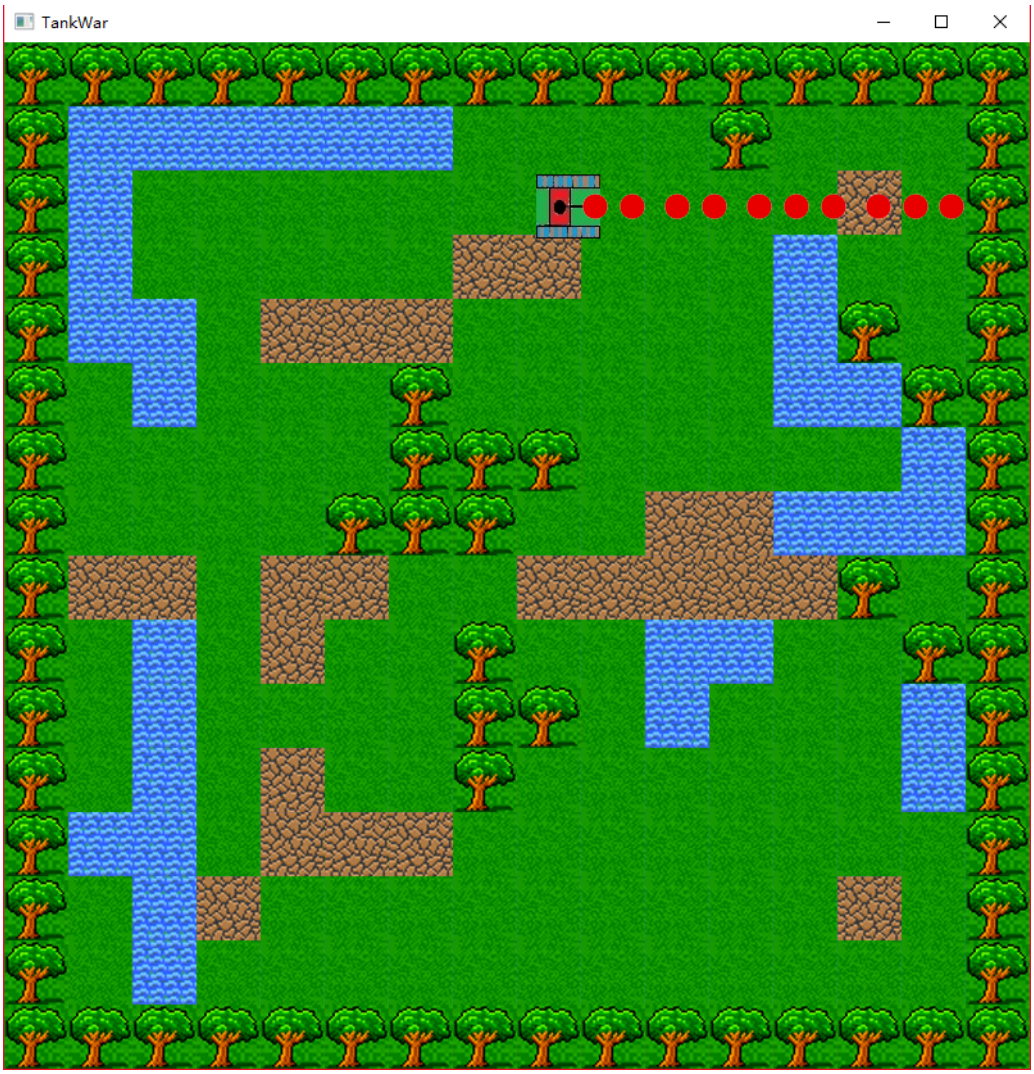


图 4-2

当子弹碰到墙以后消失，如图 4-3 所示：测试成功!!

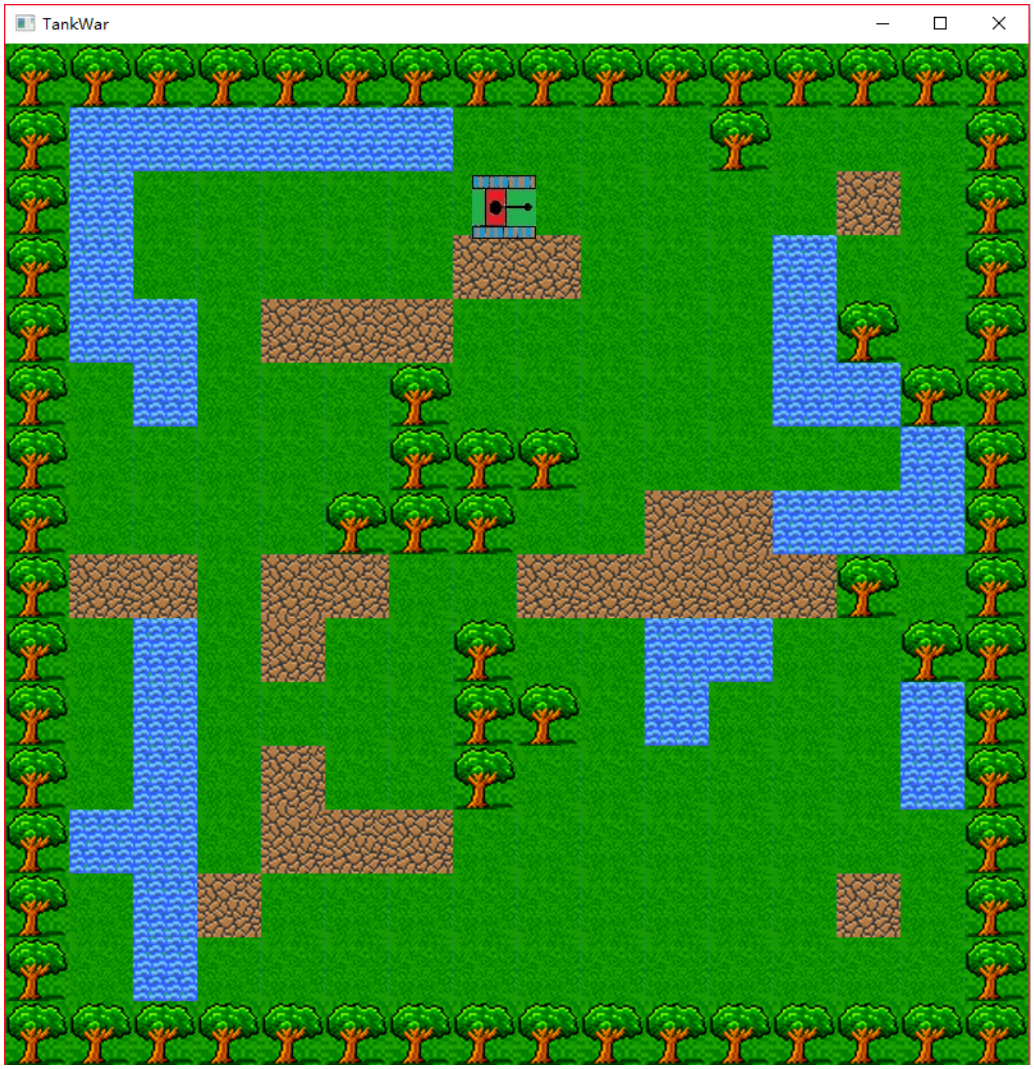


图 4-3

4.2.3 击毁敌人测试

本程序中己方坦克的功力力是 7，敌方坦克的生命值是 20。理论上命中敌人三次会击毁敌人。

如图 4-4 所示，己方坦克朝敌人发射了 3 枚子弹：

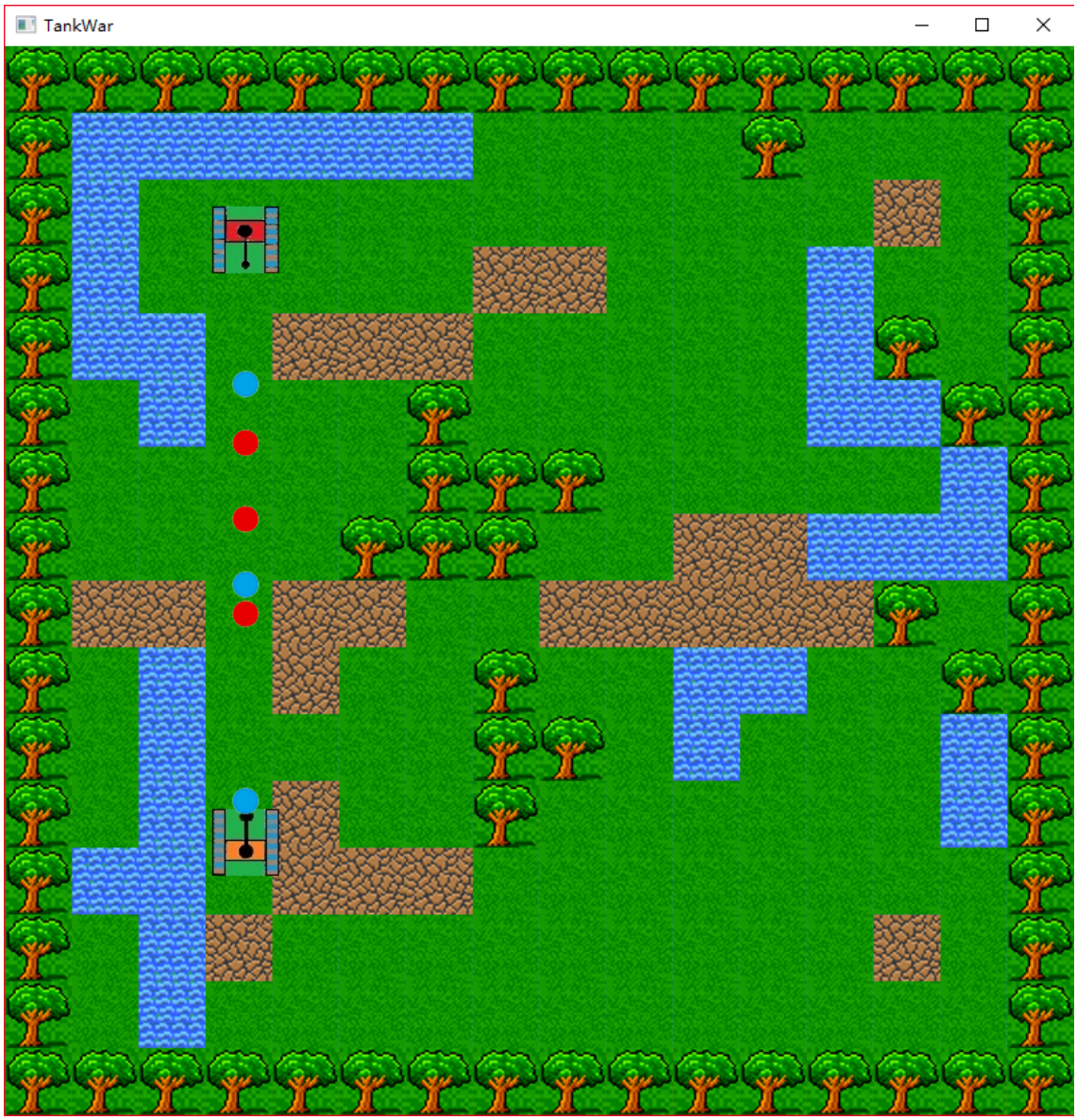


图 4-4

连续 3 次击中敌人后，敌人消失（右方敌人是新生成的）。测试成功！如图 4-5：

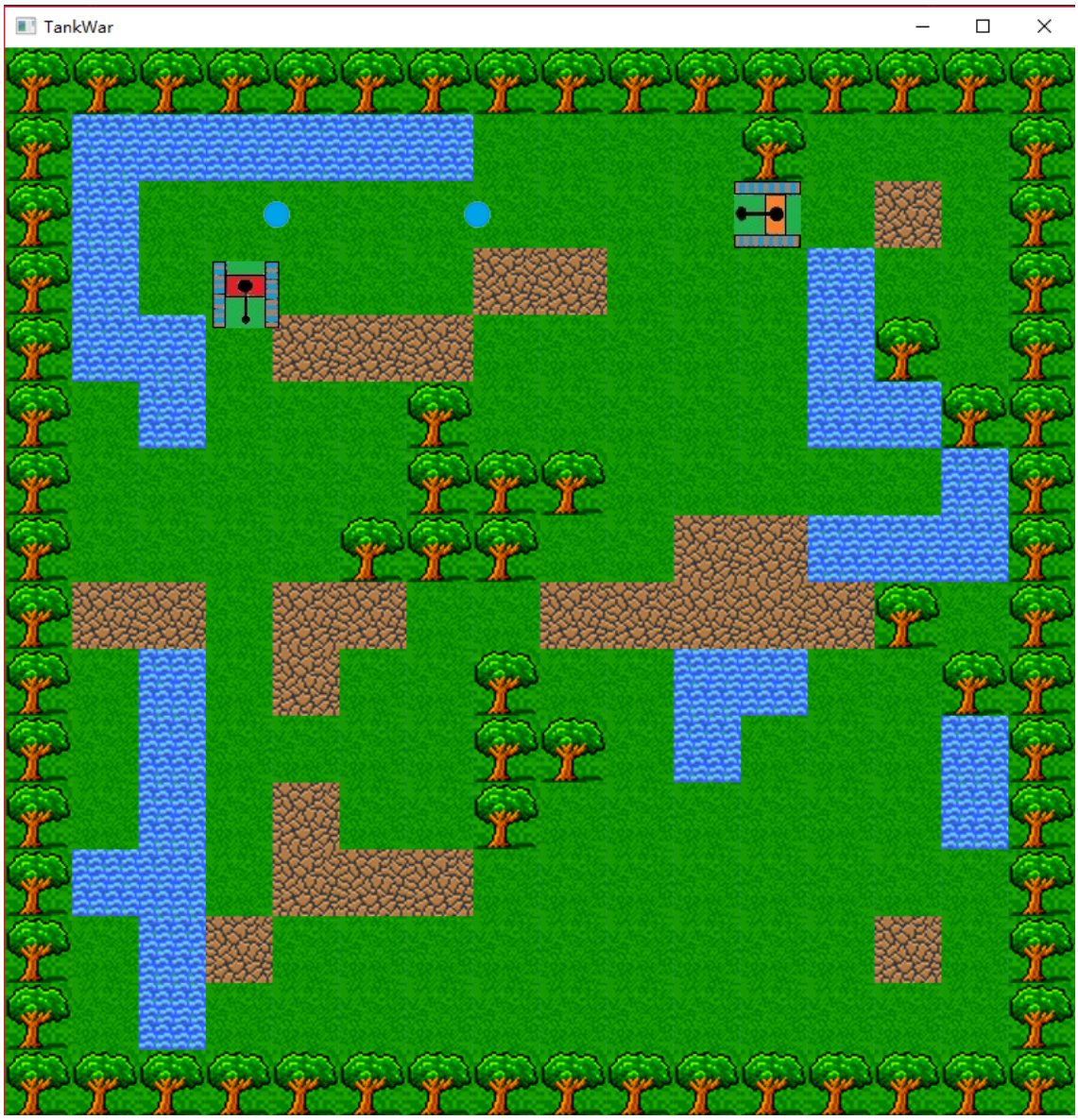


图 4-5

4.2.4 己方坦克被击毁测试

在本程序中，己方坦克生命的初始值为 100，敌方坦克伤害值为 7，理论上被累计命中 15 次后游戏会结束。

最初被击中一次后显示生命值还有 93，如图 4-6 和图 4-7：

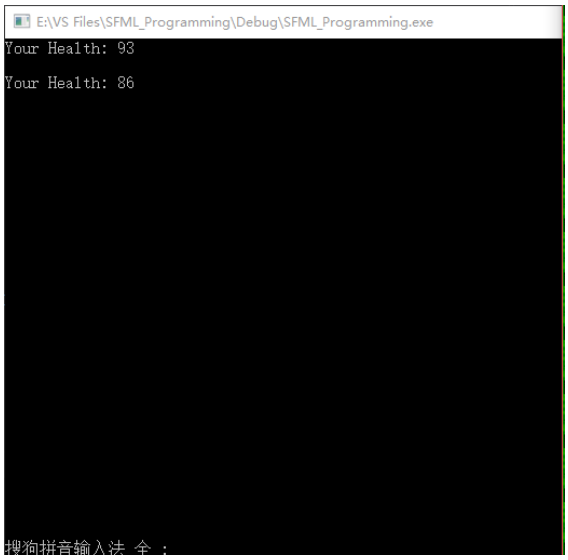


图 4-6

图 4-7

累计被击中 15 次后游戏结束，显示“Game Over!”测试成功!! 如图 4-8：

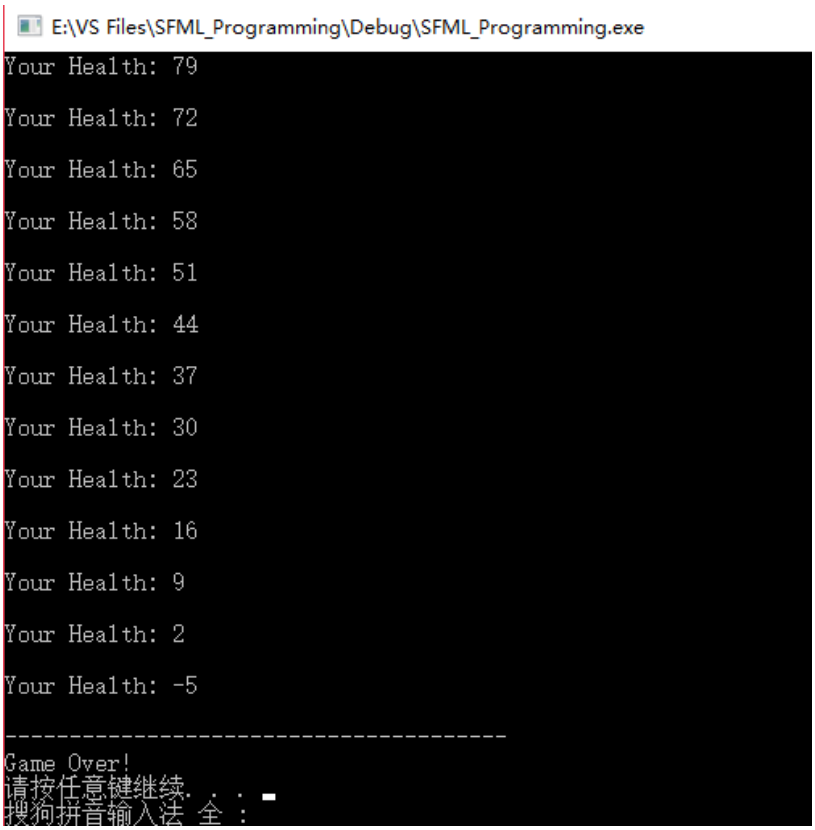


图 4-8

4.3 后续工作展望

由于时间紧迫，对子弹进行了一定的简单化处理：子弹遇到任何障碍物或者敌人都会停下，实际上当子弹遇到河流的适合不应该停下。以后有时间一定把功能做完善！

致　谢

从该游戏刚开始动工，到最后的大体功能完全实现，总共经历了整整五个星期。因为之前完全没有接触过图形库相关的知识，C++也忘记了大半。所以其实我几乎是从零开始学起的。中途经历的挫折和困难一般人都是无法想象的，别人无法想象我踩过多少坑，走过多少弯路，曾经很多次都想过要放弃或者去网上找别人的代码糊弄过去，也经常怀疑自己是不是选题的难度有点大了，但最后还是都挺了过来。

现在回看这段历程感触特别多，经常回想起自己一步一步地搭建起这个大程序的画面，现在满满地都是欣慰和自豪。感谢自己能坚持下来；也感谢老师提供的学习图形库的资料和网站，感谢老师中途对我的进度的关心，也感谢老师就敌人寻路算法这一点对我的点拨让我豁然开朗；感谢邵同学踩白块游戏对我坦克大战游戏带来的一些启示和醒悟，让我能在后期一段迷茫和瓶颈期的时候迅速找到优化和重构的思路，让我的程序得以变得更清晰明了；最最最需要感谢的是周室友，可以说没有他的帮助我肯定不能完成这个游戏，感谢他在我很多踩坑走弯路的时候及时提醒我，也感谢他在很多次关键性的敌方给我提供的宝贵的很有价值的建设性意见，以及在我多次改 bug 的时候提供的高效的帮助。