南京大学课程设计报告

高级程序设计项目报告（二）

基于控制台的植物大战僵尸游戏

学　　号：181840070

院　　系：计算机科学与技术系

姓　　名：葛睿芃

提交日期：2019/11/17

目录

前排提示：由于本项目代码量很大（所有 cpp 文件代码总行数 2290 行），我尽量以详细的语言将每一个部分都用文字说明清楚，也尽量详细地阐释我这样设计的原因，因此本报告可能有一点长，如果时间有限，请助教和老师根据目录挑选重要部分阅读并评分，感谢谅解！

一、概述

1.1 项目简介

本项目要求模仿风靡一时的塔防游戏《植物大战僵尸》，仿制一个简化的控制台版本游戏。由于是基于命令行的程序，加上游戏元素过多，所以本题所涉及到的数据结构的复杂度会非常复杂，这对于面向对象编程是一个极大的考验。

按照课程设计要求的内容，本项目最终成果要求实现游戏的基本操作：在一个 n 行 m 列的草坪上，用户在控制台操纵，运用商店系统种植植物，系统使用合理的策略释放僵尸，植物和僵尸之间能够合理交互，并且能够计算得分、正常终止，实现一个较为完整的《植物大战僵尸》小游戏。

1.2 心路历程

拿到这个题目，激动了一时，但是真正着手开始做的时候就有一点懵，甚至在数据结构的设计上面就遇到了很大的阻碍。游戏不同于数据库，所涉及的数据结构的嵌套性没有那么明显，且比较复杂，各个数据结构之间的交互比较频繁，如何适当地处理这些成了一个比较关键的问题。

其次，如何让游戏能够成功地跑起来？这个项目要求我们从底层设计到上层调用完整地设计出来，所以各个模块之间的处理又是一个巨大的难点。加之程序的多线程运行是以前从来没有涉及过的编程领域，所以这次的设计对我来说是一个巨大的挑战。

接下来的部分我将详细说明我的编程思路和心得体会，特别是在设计上对“为什么要这么做”的详细讨论。此次设计不仅是一次作业，也将成为我程序设计里程上的一次重要的经验积累。

1.3 需求分析

本项目在游戏的基础功能设计上有如下要求：

- 设计若干植物、若干僵尸；
- 设计商店系统，使得用户可以通过商店购买、种植植物；
- 设计合理的僵尸进攻策略，实现白天前院无尽模式的模拟；

同时，本项目在程序运行与用户交互上有如下要求：

- 显示当前草地的情况，包括已经种植的植物及其状态，已经登上草坪的

僵尸及其状态，尽量模仿原作设计；

- 植物、僵尸能够根据自己的特性进行交互（包括攻击、移动、阳光收取），使得游戏正常运行和中止；
- 实现用户操作与系统运行的多线程设计，尽量做到界面清除、交互友好。

二、设计思路

2.1 数据结构分析

2.1.1 写在前面

本题涉及的数据丰富而繁杂，包括各种角色、地块、技能等等。由于没有游戏编程的经验，本项目的数据结构是根据我平常玩游戏和作游戏功能拓展的时候的经历而设计的，中途也做了很多次大规模的修改，可能不是很妥当，还请助教与老师谅解。

2.1.2 角色(roles)设计

《植物大战僵尸》，顾名思义，就是植物和僵尸两大阵营的激烈对抗，那么毫无疑问“**植物(plants)**”和“**僵尸(zombies)**”就是本游戏的两大主要角色，对于游戏角色的底层设计，很容易想到用两个类分别对植物和僵尸的属性进行定义。

可是仔细一想，“植物”“僵尸”这两个名词代表的真的只有这一个含义吗？并不是，我们要知道游戏是一个动态的，不断运行的程序，在游戏中，看上去相同类型的数据其实在底层表示上很不一样。

比如“植物”，它在游戏中的表现就有两个数据形式，一个是**展示在商店里，表示着若干静态属性，还有冷却时间和花费等性质的“植物属性(plants' property)”**，另一个是**实实在在种植在草坪上的，具有一定属性，能够实时与草坪上的僵尸进行交互的“植物角色(plants' role)”**。那么同理，僵尸在游戏中也被表现成**“僵尸属性(zombies' property)”**和**“僵尸角色(zombies' role)”**。

因此，在角色方面的数据结构，对于每种角色我们定义了两个类，一个代表静态属性，一个代表实体角色，对于植物，我们定义**“植物属性类”class Plant_ppt**

```
1.  class Plant_ppt{   // defined in property_define.h
```

```
    private:
        string name;
        PType type;
        Bullet bullet;
        int pcd;
        int charging_time; // the cd time remaining.
        int spend;
        int maxhp;
        string init_effect; //we acknowledge that one plant has mostly one effect and one skill initially.
        string init_skill;
        string describe; //the description of the plant
    public:
        Plant_ppt(string n, PType t, Bullet b, int pcd, int spend, int maxhp, string describe,  string effect_name, string skill_name);
        //other functions
};
```

解释：

- string name 是植物的名称，属于静态属性，每个植物都具备属于自己的名称，且在程序运行时不会改变，因此 name 也兼具身份识别（id）的功能。
- PType type 是一个枚举类型，表示植物类型（进攻型、效果型、防御性、爆炸型），也是静态属性。
- Bullet bullet 是一个结构体，说明了植物是否拥有子弹，以及子弹的基本数据（攻击力，释放间隔等）。
- int pcd 是植物的冷却时间，是静态属性，下面的 int charging_time 则表示了植物当前所剩余的冷却时间，是动态变化的，所以可想而知，我们在设计商店的时候，有一个数据成员肯定是 Plant_ppt 类的向量，且 Plant_ppt 类的对象只会在商店类中创建，不会在全局函数及其他类中创建。
- int spend 与 int maxhp 表示植物的阳光花费与最大生命值，是植物的静态属性。
- string init_effect 和 string init_skill 表明了植物所具有的的效果与技能，关于什么是效果，什么是技能，它们又是如何在程序中被激活的，请看 2.1.3 部分关于技能数据结构的介绍。
- string describe 是植物的描述，用于图鉴展示。

马上我们又能类比地写出**“僵尸属性”类 class Zombie_ppt** 如下

```
class Zombie_ppt{  // defined in property_define.h
  private:
    string name;
    int maxhp;
    int attack; //the attacking index per frame
    int speed; //time to move to the next field
    string init_effect;
    string init_skill;
    string describe;
  public:
    Zombie_ppt(string n, int maxhp, int attack, int speed, string describe,
            string effect_name, string skill_name);
    //other functions
};
```

关于这两个“属性”类的成员函数，大部分都是获取私有成员的函数（还有一些与技能相关，详情 2.1.3），基本不会出现修改私有成员的函数，因为它们的数据成员表示的大多是一个角色的静态属性，除了植物属性中需要一个和冷却相关的函数，这个函数不是这个项目的重点，具体的定义可以查看代码文件。

在游戏中我们需要处理大量的这样的属性对象，因此就需要有一个“属性库”管理这两个类，我们定义**“角色图鉴类”class RoleGallery**：

```
class RoleGallery{  //defined in game_system.h
  private:
    string filename_plant;
    string filename_zombie;
    vector <Plant_ppt> plant_list;
    vector <Zombie_ppt> zombie_list;
    int plant_num;
    int zombie_num;
  public:
    RoleGallery(string pfile, string zfile);
    Plant_ppt* get_plant(string name);
    Plant_ppt* get_plant(int id);
    Zombie_ppt* get_zombie(string name);
    Zombie_ppt* get_zombie(int id);
    //other functions
    ~RoleGallery(){
      plant_list.clear();
      zombie_list.clear();
    }
};
```

其中，string filename_plant 和 string filename_zombie 是植物属性和僵尸属性存

放的本地文件，他们分别被永久地存放在**”sysfile\plants.txt”**和**”sysfile\zombies.txt”**中。从文件中读取而不是使用一堆代码创建属性对象无疑是减少了大量的重复性质的代码，使得代码相对易读友好。

举个例子，植物属性文件**”sysfile\plants.txt”**中就是以以下形式存放数据的：

```
1. 豌豆射手 1 true 1 20 4 1 30 100 300 null null 豌豆射手可谓你的第一道防线，他们朝来犯的僵尸射击豌豆。
2. 向日葵 4 false 0 0 0 0 30 50 300 sun_generating null 向日葵是你收集额外阳光必不可少的植物。为什么不多种一些呢？
3. 双发射手 1 true 1 40 4 2 30 200 300 null null 双枪豌豆很好斗。一次可以发射两颗豌豆。
4. 寒冰射手 1 true 1 20 4 3 30 175 300 null slow_you_down 寒冰射手的雪花豌豆发射的冰冻豌豆不仅能够伤害僵尸，还能够放慢他们的步伐。
5. 坚果墙 2 false 0 0 0 0 120 50 4000 null null 墙果具备坚硬的外壳，你可以使用他们来保护其他植物。
```

其中每一行都是一个植物属性，然后以名称，类型，子弹激活，子弹速度，子弹攻击，子弹间隔，子弹类型，冷却时间，花费，生命，初始效果，初始技能，植物描述的顺序，中间用 tab 隔开，存放植物信息，这些信息在构建 RoleGallery 类时被获取，用于创建对象。关于这些数据的具体含义，我们将在 2.2 主要算法分析部分作深入探讨。

这个类中的成员函数中有获取相应植物和僵尸属性地址的重载函数 Plant_ppt* get_plant 和 Zombie_ppt* get_zombie，他们在创建角色对象时具有一定作用。所以接下来我们来谈谈那些实实在在上战场，去进行交互的**“植物角色”类 class Plant_role** 与**“僵尸角色”类 class Zombie_role**：

```
1.  //both defined in role_define.h
2.  class Plant_role{
3.    private:
4.      Plant_ppt* property;
5.      int health;
6.      int bullet_time;
7.      vector <PEffect> effect;
8.      vector <TSkill> skill;
9.      int row;
10.     int column;
11.     bool die;
12.   public:
13.     Plant_role (Plant_ppt* ppt, int row, int column);
14.     //other functions
15.     ~Plant_role();
```

```
16. };
17. class Zombie_role{
18.     private:
19.         Zombie_ppt* property;
20.         int speed;
21.         int health;
22.         int attack;
23.         int move; //next time to move to the next field
24.         Status status;
25.         vector <PEffect> effect;
26.         vector <TSkill> skill;
27.         int row;
28.         int column;
29.         bool die;
30.     public:
31.         Zombie_role(Zombie_ppt* ppt, int row, int column);
32.         //other functions
33.         ~Zombie_role();
34. };
```

这时候我们发现，除了指向静态属性的指针，两个类还具有一些属于角色的动态特征，比如生命值 health，安放位置 row, column 等。

僵尸类中的 Status status 是一个结构体，表明僵尸的状态（无、减速、冰冻），用于与寒冰射手等植物的交互上。

另外，PEffect 和 TSkill 是定义效果和技能的两个结构体，这个细节我们同样在下一节展开，这里可以理解为角色拥有的特殊技能（比如向日葵生产阳光、寒冰射手让人减速等）。

不同于属性类，角色类中的成员函数大多都与改变数据成员有关，而不仅仅是获取数据成员。

通过以上的定义，我们可以推测（实际上是我提前设计好了的）将来种植植物和释放僵尸时，首先我们会获取到 RoleGallery 类中的相应角色的静态属性地址 Plant_ppt* (or Zombie_ppt*) ppt，然后通过以下方式创建动态对象：

```
1. Plant_role* role = new Plant_role(ppt, row, column);
2. //zombies are created like this as well.
```

植物或僵尸死亡时，再通过 delete 删除即可。这样的设计看似巧妙，其实隐患无穷（手动滑稽，具体请看本报告特别章节 2.2.4）。

至此，角色类的定义算基本阐述完毕了，具体的欢迎看我的源代码（涉及到

property_define.h, role_define.h, game_system.h 三个头文件和 property_define.cpp, role_define.cpp, role_gallery.cpp 三个源代码实现文件，具体在报告的第三部分我会具体阐释我的模块划分技巧）。

2.1.3 技能(effect/skill)设计

众所周知，植物大战僵尸中的角色都各有特点，具体来说，每个角色都有一定的“技能”，使得游戏玩法多样而有趣，那么如何实现这些技能就成了关键的问题。如果说一种每个角色都定义一个数据结构的话，就太过于繁琐，而且代码的复用程度很低，于是就要有一个整体的数据结构来管理这些技能。

然而这些技能千差万别，发动效果、发动时机都不一样，如何有一个统筹的管理机制来设计这些技能呢？

受到守望先锋(overwatch)游戏中地图工坊(workshop)的启发，我们将技能分成两类：一类是**全局持续判定，有一定触发时间间隔的“效果（effect）”**，另一类是**依赖于游戏特殊交互进程判定，触发需要条件的“技能（skill）”**。

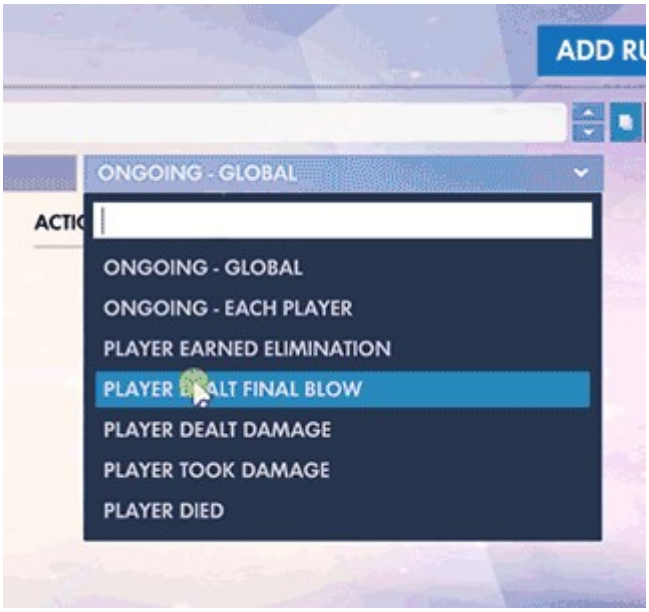

图 2.1 守望先锋地图工坊的事件类型

具体来说，效果（effect）依赖于全局判定，游戏每进行一帧，都会检查效果是否能够被触发，而这类效果的触发条件往往依赖于时间，每隔一段时间才会触发被动效果，适用于这类的效果有：向日葵的阳光(sun_generating)，土豆雷的装填(potato_activating)等，我们定义效果结构体如下 **struct PEffect** (passive effect):

```
struct PEffect{  //passive effects, defined in property_define.h
    string effect_name;
    int role;  //1:plant 2:zombie
    int remaining_time;
    int time;
    PEffect():remaining_time(0),time(0){}
};
```

其中，time 是指每触发一次效果的时间间隔，为静态属性，remaining_time 是用于动态判定距离下一次触发的剩余时间的，这个结构体的向量被放在角色类的数据成员中，用于每个角色对象的效果触发判定，同样，“技能”结构体 **struct TSkill** (triggered skill)也有类似的定义：

```
struct TSkill{  //triggered skill, defined in property_define.h
    string skill_name;
    int role;  //1:p->z 2:z->p
    TSType trigger;
    TSkill():trigger(TS_NONE){}
};
```

其中，TSType trigger 是指该技能的触发条件，根据本项目的要求，所有技能的触发条件可以被分为两类：**ATTACKING**（造成攻击）和 **ATTACKED**（被攻击），技能结构体的向量也被放在了角色类的数据成员中，这是为了在角色做出相应的动作的时候，对这些技能进行判定。

PS：即使看上去还可能存在其他的触发条件，但是我们都可以简化为这两个条件，比如窝瓜的压扁技能(pump_exploding)，可以看做 ATTACKED 触发条件，因为我们设定僵尸与植物走在同一格时才能进行攻击，且窝瓜的攻击范围也是它所在的这一格。

然而我们可以看到，在属性 property 类中，我们只将角色的效果和技能名（字符串）进行读取，实在的技能实现并没有被传入，但是不同技能的触发条件和效果是不一样的，但是我们需要一个统一的方式来创建和调用这些技能。从一个字符串到一个实实在在的技能，需要经历以下步骤：

Step1：创建角色 role 对象时，将已有初始技能从字符串解释为 PEffect 或 TSkill 结构体，并加入向量中，这一步是由两个全局函数（整个项目为数不多的全局函数）完成的：

```
PEffect _make_effect(string name, int role){
    PEffect ef;
    ef.effect_name = name;
```

```
    ef.role = role;
    if(name.compare("sun_generating")==0){
        int rand_num = rand()%29+12;  //12-40
        ef.remaining_time = rand_num;
        ef.time = 96;
    }
    if(name.compare("potato_activating")==0){
        ef.time = ef.remaining_time = 60;
    }
    //.....
    return ef;
}
TSkill _make_skill(string name, int role){
    TSkill sk;
    sk.role = role;
    sk.skill_name = name;
    if(name.compare("pump_exploding")==0){
        sk.trigger = TS_ATTACKED;
    }
    if(name.compare("water_split")==0){
        sk.trigger = TS_ATTACKING;
    }
    //...
    return sk;
}
```

通过这两个函数，根据所传入字符串的名称，确定结构体相应参数的值，最终将结构体返回，在构建角色对象，以及获得额外技能的时候就会调用这个函数

Step2：在满足触发条件时，将所要触发的效果或者技能的名称，和对象指针传入 Game 类的技能触发函数中：

```
void Game::do_effect(string effect_name, Plant_role* p){
    if(effect_name=="sun_generating"){
        sun_generating(p);
    }
    if(effect_name=="potato_activating"){
        potato_activating(p);
    }
    //...
}

void Game::do_skill(string skill_name, Plant_role* p, Zombie_role* z){
    if(skill_name == "pump_exploding"){
        pump_exploding(p,z);
    }
```

```
    //...
}
```

（PS：还有两个重载函数是对僵尸而言的，这里以植物发动技能为例）

Game 类是游戏引擎类，其中包含关键游戏参数（地块信息、得分、阳光）和推进游戏进程的函数，具体会在 2.1.4 以及 2.2 部分展开。

当系统判定满足某技能的触发条件时，会将这些技能的名称以及技能实施的对象传入对应的技能调用函数中，指示调用对应的函数，那么具体定义技能的函数到此才真正地被调用。

Step3：根据传入字符串，调用具体技能函数。

在 Game 类中定义了一系列技能函数，这些函数声明在 game_system.h 中的 Game 类中，定义在 skill_definition.cpp 文件中，这些函数不依赖于某一个具体的角色对象，也不作为角色类的成员函数，这样设计既增加了代码的灵活性（后续设计新角色的时候，只需要在系统依赖文件中写上相应角色参数和技能名称，然后到 skill_definition.cpp 中定义对应技能，填充 Step1 和 2 涉及的函数即可，不需要再到其他代码段去写其他函数调用，非常地方便），又减少了代码的复杂性，使得数据结构之间的联系更加清晰。

关于如何判定触发条件，我们在 2.2 部分会详细说明。

2.1.4 地块(Field)及游戏系统(Game)设计

定义了植物和僵尸的属性、角色、技能，还有一个就是关键的草坪。游戏采用白天前院，所以我们要模拟一个 5x8 的游戏场景，并且在这个草地上能够种植植物、走僵尸，因此，我们定义**“地块”结构体 struct Field** 如下：

```
struct Field{  //defined in game_system.h
    int row;
    int column;
    bool strike; //dici effect
    vector <BAttack> blts;
    Plant_role* plants; //only one plant can be set in one field.
    vector <Zombie_role*> zombies;
};
```

其中，int row 和 int column 表示地块编号(0-4,0-7)，Plant_role* plants 表示这一地块上种植的植物，用指针表示，指向动态空间的某一个创建的动态对象。由于一块地只能种一株植物，所以不像 zombies 需要用一个向量表示。

我们还可以从这个结构体中看到一些设计上的精妙之处，比如对特殊植物**“地刺”**的设计，真正底层设计的时候，我们不将地刺看成一个植物（尽管它在图鉴中会被展示），而是看成一个地块特效 bool strike，这样僵尸不会吃它，但地刺仍然能对僵尸造成伤害。

vector <BAttack> blts 是指子弹，其中 BAttack 是定义子弹动态对象的结构体，其中包含了子弹的释放者（“Plant_role* 变量”），子弹的威力等参数，当子弹与僵尸处于同一格时，子弹就会对僵尸造成伤害。

在后续的设计中，我们不难想象用一个二维数组 Field land[5][8]来模拟一个 5x8 的前院草坪，由于地块的存放已经涉及到游戏的动态运行效果了，所以这就涉及到我们游戏的最终引擎：**游戏系统类 class Game**，它的定义如下：

```
1.  class Game{
2.    private:
3.      RoleGallery roles;
4.      Field land[5][8];
5.      int point;
6.      int wave;
7.      int sun;
8.      bool game_running;
9.      bool extra_point;
10.     string log;
11.     string shop_log;
12.     string sun_log;
13.     string wave_log;
14.     int chosen_row;
15.     int chosen_column;
16.     int chosen_plant;
17.     //...
18.   public:
19.     Game(string s);
20.     //other functions…  including skill definitions
21. };
```

其中包含了角色图鉴 RoleGallery 的实例对象 roles，一个 Field 类型的二维数组 land[5][8]，游戏运行参数 bool game_running，还有表示游戏运行状态的参量，包括进攻波数 int wave，得分 int point，阳光数量 int sun，与计分相关的变量 bool extra_point（判定获得额外分数的条件），还有各种运行时会显示在屏幕上的 log 字符串，以及用户在选择植物和地块时的选择信息。

关于 Game 类的成员函数，除了与游戏进程相关的函数，还定义了所有角色的效果与技能，在 Game 类的构造函数中，一定有对 RoleGallery 类对象的创建，所以 RoleGallery 类的对象一定不会在其他地方被定义。同时，由于 RoleGallery roles 中定义了所有植物的静态属性，所以这个对象就可以被当做商店使用，不需要再额外定义商店类了。

Game 类对象在 main 函数的一开始就被定义，然后 Game 类对象的引用将被作为参数，传入界面展示函数，程序将由此开始运行，以下为 main 函数的关键操作定义：

```
1.  int main() {
2.      Game g;
3.      enter_main(g);
4.      return 0;
5.  }
```

其中，**void enter_main(Game& g)**是一个界面展示全局函数，表示进入游戏的主界面。

至此为止，关于本游戏的所有关键数据结构就定义完成，下面我们就要想办法将所有的数据结构，通过一定的算法联系起来。

2.2 主要算法分析

2.2.1 游戏的底层设计

在本项目中，我们力求在角色本身的设计上就做到尽量的详细，以在编写 Game 类的关键函数时，只需要调用底层类的成员函数即可，无需另外编写程序实现。

在上一个部分，我们探讨了角色效果和技能从字符串到真正定义实现的过程，这是本项目中一个比较典型的底层设计，类似的底层设计还有很多，他们都位于角色类 Plant_role 和 Zombie_role 的成员函数中，我们这里举几个例子来探讨一下。

比如，在实现僵尸往前走的时候，我们再 Zombie_role 类中有一个成员函数 **bool move_forward();**

首先我们简单了解一下这个游戏的僵尸运行的规则。游戏将一帧一帧循环进行，在设置僵尸静态属性的时候，我们定义了一个 speed 变量，表示僵尸向前前进一格之间的间隔时间（单位：帧）。在 Zombie_role 的成员变量中也有一个动态变量 move，它会记录距离僵尸下一次前进的剩余时间，因此我们看到这个成员函数的定义：

```
1.  bool Zombie_role::move_forward(){
2.      bool suc = true;
3.      if(status.type == S_FROZEN){
4.      //.....
5.      }
6.      else if(status.type == S_SLOW){
7.      //.....
8.      }
9.      else{
10.         move -- ;
11.     }
12.
13.     if(move==0) {
14.         column --;
15.         if(column < 0) return false;
16.         move = speed;
17.     }
18.     return suc;
19. }
```

前面两个 if 语句是对僵尸特殊状态的处理，我们在报告中暂时忽略这一点，来看正常情况下僵尸的移动情况。

首先，move 减 1，然后判断 move 是否等于 0，等于 0 说明到了僵尸前进的时间了，那么僵尸的位置将从原来的(row, column)变为(row, column-1)，同时将 move 变量重置。这是要判断特殊情况，如果僵尸走到底线，即 column<0 的时候，游戏结束了，我们将这一个信息反应在函数的返回值上面，Game 类对象调用这一函数的时候将会得到这一返回值，从而做出相应的处理方式。

除了僵尸的运行，游戏中“伤害”是另一个尤为重要的底层设计，在设计上不仅要考虑伤害量、伤害者、受伤者，还要考虑伤害与角色技能之间的联动关系。

我们将伤害造成的生命值变化封装为一个整体的数据结构 class Damage，并定义一个 void dealt()函数处理生命值变化，这个函数在 Game 类中的伤害重载函

数 void damage 中会被调用，这样设计是为了游戏进程中出现造成伤害时，调用一个 damage 函数就可以完成生命值变化、击杀、发动技能的一整套操作。

```
class Damage{
    private:
        Plant_role* p;
        Zombie_role* z;
        int attack_type; //1: p->z;  2:z->p
        int index;
    public:
        Damage(Plant_role* p, Zombie_role* z, int i); //1: p->z
        Damage(Zombie_role* z, Plant_role* p, int i); //2: z->p
        void dealt();
        ~Damage();
};
//在 damage 函数中调用 Damage::dealt 的示例
void Game::damage(Zombie_role* z, Plant_role* p, int index){
    Damage d(z,p,index);
    d.dealt();  //调用函数，生命值变化
    bool skill_activated = true;
    if(p->get_health()<=0 && !z->is_killed()){
        skill_activated = false;

        log = z->get_property()->get_name();
        log += " 击败了 ";
        log +=p->get_property()->get_name();
        p->be_killed();
        extra_point = false;  //击败判定
    }
    vector <string> zs = z->get_attacking_skill();
    int size1 = zs.size();
    for(int i=0;i<size1;i++){
        do_skill(zs[i],z,p);  //进攻方发动技能
    }
    if(skill_activated){
        vector <string> ps = p->get_attacked_skill();
        int size2 = ps.size();
        for(int j=0;j<size2;j++){
            do_skill(ps[j],p,z);  //受伤害者发动技能
        }
    }
}
```

解释：vector <string> get_attacking_skill() 与 vector <string> get_attacked_skill()

函数是定义在角色类中获取所有触发类型为 ATTACKING 或 ATTACKED 技能的名称（字符串），这些字符串将作为索引在 void Game::do_skill 重载函数中调用相应的定义技能的函数（2.1.3 中的 Step3 的步骤）。 void be_killed()函数是击败函数，用于在下一帧开始之前移除目标角色（具体的设计详见 2.2.4）

2.2.2 角色的出现与消失

角色对象的创建

我们定义了 Plant_role 和 Zombie_role 类的对象，在游戏进行的时候我们会对这些对象进行创建，那么具体的创建是如何的呢？

我们分植物和僵尸讨论，在种植植物的时候，一般是由用户通过操作，在相应地块(row, column)种植。通过之前的介绍，我们知道 Game 类中的 RoleGallery 对象 roles 兼有商店的功能，而 int chosen_plant 数据就表明了用户目前选择的植物的序号，这个序号将由用户通过键盘操作改变，最终会选择一个第 i 号植物进行种植。

在种植按键按下去后，系统将根据这个序号 i，以及对应的地块编号 chosen_row 和 chosen_column，创建相应的对象。具体过程将分为以下几步：

Step1：获取植物静态属性地址。我们可以看到 roles 有一个重载函数 **Plant_ppt* get_plant(int id)**; 通过将用户所选植物的序号传入，可以获得静态属性地址 Plant_ppt* ppt; 这个静态属性中包含了冷却时间、阳光花费，以及植物的其他属性。

Step2：对比当前游戏参数，判断是否可以种植。植物可以种植，当且仅当本地块无其他植物已经被种植，冷却时间为 0，且阳光数量充足。

Step3：如果可以种植，创建动态对象。首先在动态堆区，通过 Plant_role 的构造函数，传入 ppt，chosen_row 和 chosen_column 创造动态对象，然后将对应地块 land[chosen_row][chosen_column]的 plants 变量设置为该动态对象的地址，至此，动态对象创建完成。

Step4：阳光调整，冷却重置。

具体定义如下：

```
if(land[chosen_row][chosen_column].plants!=NULL || land[chosen_row][chosen_column].strike){
    shop_log = "此地块无法种植！";
    break;
}
Plant_ppt* pt = roles.get_plant(chosen_plant);
if(pt->get_charging()!=0){
    shop_log = "冷却时间未到！无法种植！";
    break;
}
if(pt->get_spend()>sun){
    shop_log = "阳光不足！无法种植！";
    break;
}
if(pt->get_name()=="地刺"){ //地刺不看做一个植物，看成一个地块效果
    land[chosen_row][chosen_column].strike = true;
    pt->recharge();
    sun -= pt->get_spend();
    break;
}
else{
    Plant_role* pr = new Plant_role(pt,chosen_row,chosen_column);
    set_plant(pr);
    pt->recharge();
    sun -= pt->get_spend();
    break;
}
```

这部分是在 Game 类的成员函数 **void keyboard_operation()**的一部分，期间调用了 Game 类的函数 **void set_plant(Plant_role* p)**，这个函数是为了将地块的 plants 变量进行调整，具体的定义如下所示：

```
bool Game::set_plant(Plant_role *p){
    int row = p->get_row();
    int column = p->get_column();
    if(land[row][column].plants!=NULL) return false;
    land[row][column].plants = p;
    return true;
}
```

对于僵尸来说，创建僵尸的过程十分类似，不过这不需要用户操作，系统自动生成，所以我们可以用 roles 对象的 **Zombie_ppt* get_zombie(string name)**这一重载函数获得静态属性，然后类似地创建动态对象。创建僵尸对象的操作被定义在 Game 类成员函数 **void wave_attack()**中，这个函数定义了释放僵尸的策略。

角色对象的消亡

创建的角色对象在被铲除或被击败时将会消失，由于是动态变量，需要用 **delete** 操作符释放空间。

Game 类的成员函数中定义了 bool Game::remove_plant(Plant_role* p)与 bool Game::remove_zombie(Zombie_role* z)两个函数进行对象消亡。具体的对象消亡可以分成以下步骤：

Step1：获取需要消亡的角色对象地址；

Step2：如果是植物，直接 delete 并把地块中的 plants 变量设为 NULL，如果是僵尸，在对应地块中用 vector::erase 函数将僵尸从向量中清除，然后 delete 释放空间。

具体的定义如下：

```
1.  bool Game::remove_zombie(Zombie_role* z){
2.      int row = z->get_row();
3.      int column = z->get_column();
4.      Field* f = &land[row][column];
5.      int size = f->zombies.size();
6.      for(int i=0;i<size;i++){
7.          if(z == f->zombies[i]){
8.              f->zombies.erase(f->zombies.begin()+i);
9.              delete z;
10.             return true;
11.         }
12.     }
13.     return false;
14. }
15.
16. bool Game::remove_plant(Plant_role* p){
17.     int row = p->get_row();
18.     int column = p->get_column();
19.     Field* f = &land[row][column];
20.     if(f->plants!=p) return false;
21.     else {
22.         f->plants = NULL;
23.         delete p;
24.         return true;
25.     }
26. }
```

2.2.3 游戏进程的推进

在底层方面设计完成，角色创建和消亡都到位了之后，我们就要考虑游戏进程的问题了。

首先我们考虑游戏中角色之间的自动交互和游戏的推进。通过分析，我们可以作以下模型简化处理：我们假设游戏是一帧一帧进行的，我们就可以一帧一帧对游戏中的各个对象进行解析，然后不断循环，直到满足一定条件，游戏结束，循环停止。

所以，从框架上来说，游戏一定是通过这种方式来进行运作的：

```
1.  void Game::loops(){
2.    int i=0;
3.    while(is_running()){
4.       if(i%2==0){
5.          draw();
6.
7.       }
8.       Sleep(200) ;
9.       frame_forward();
10.      i++;
11.   }
12. }
```

我们假设每一帧时间 200ms，每两帧向用户进行一次展示（调用 **void Game::draw()**），每一帧就通过 **void Game::frame_forward()**函数进行逐帧解析。直到游戏满足一定的终止条件（**bool Game::is_running()==false**）的时候，循环结束，函数返回。

下面我们讨论逐帧解析的过程。根据我们的游戏场景模型，我们在解析的时候主要是对两个部分进行调整：一是图鉴中植物冷却时间的调整，二是对每一地块的植物、僵尸、子弹的解析。

对冷却时间，我们只需要将调用一下底层已经写好的函数 **void Plant_role::discharge()**就可以了，对于每一地块的解析，我们要从多重角度且尽量完整地进行操作，总体说来，我们可以分为以下几个部分：

1. 对于之前已经调用了 void be_killed()函数的对象（用底层写好的函数 bool is_killed()进行判断），用 remove_plant 或 remove_zombie 函数作消亡处理。这里我们采取的技巧是，当生命值为 0 的时候，没有立即消亡对象，而是等到下一帧

的开头做统一处理，这样做的目的在 2.2.4 部分将作说明。

2. 对角色的被动效果(effects)进行解析，调用底层写好的 **vector <string> effect_count()**函数，获取所有可以被触发的效果，并根据返回的效果名称（字符串），通过 void do_skill()重载函数调用相应的技能函数发动技能。

3. 对地块特效（地刺）进行解析，如果存在地块特效，调用 **void Game::damage()**函数造成伤害，这个函数会自动发动满足条件的触发技能(skill)。

4. 对于僵尸与植物处于同一块的地块，对于每一个僵尸，分别传入僵尸与植物对象指针，调用 **void Game::damage()**函数造成伤害。

5. 对于有僵尸没有植物的地块，调用底层写好的 **void Zombie_role::move_forward()**函数，解析僵尸的移动，并且对产生移动的僵尸调整位置（因为要求设计的僵尸都是往左走，而遍历是往右遍历的，所以不会产生循环冲突问题）。

6. 对于有植物的地块，若该行存在僵尸，调用底层写好的 **bool Plant_role::do_bullet()**检查植物的子弹状态，如果可以发射，则创建 BAttack 对象加入地块的 blts 向量中。

7. 对于存在子弹的地块，将子弹根据移动速度前移，这里因为子弹移动方向和遍历方向一致，可能存在循环冲突问题，于是乎我们在 BAttack 结构体中加入 bool move 变量判断子弹是否已经移动。

至此，我们已经基本完成对一个地块的解析。

但是游戏进行的同时，还有三个进程需要同时进行，一个是从天上获取阳光，一个是读取用户的键盘操作，选择相应的地块和植物进行种植，还有一个就是释放僵尸。这三个操作我们用另外三个函数封装，分别是 Game 类下的 void suninc()，void keyboard_operation()和 void wave_attack()函数，在游戏的启动函数（也就是 main 中调用的函数 void run()中），我们以多线程方式连接这四个游戏进程：

```
1.  void Game::run(){
2.      system("cls");
3.      reset();
4.      game_running = true;
5.      sun = INIT_SUN;
6.      //... other operations
7.      thread t1(&Game::loops, this);
```

```
8.      thread t2(&Game::keyboard_operation, this);
9.      thread t3(&Game::suninc, this);
10.     thread t4(&Game::wave_attack, this);
11.     t1.join();
12.     t2.join();
13.     t3.join();
14.     t4.join();
15.     //... game over operations
16. }
```

这就是整个游戏推进的基本原理了。

2.2.4 “Segmentation Falut”一生之敌

当写完程序，高高兴兴调试，玩了不到一波，跳出来一个窗口：

竟然是我们的老盆友“Segmentation Fault”出现了！

十分纳闷的我又测试了几局，发现了同样的问题，游戏往往无法进行到第二波甚至更多，且每一次几乎是在类似的地方（比如放窝瓜，僵尸阵亡等）出现提示。

程序员的本能告诉我一个可能的原因：被 delete 释放掉了的空间被访问了！由于本项目是我第一次搞多线程编程，所以我做出了推测：在其他线程被 delete 掉了的变量，在另一个线程中被访问了？这就导致函数 move_forward()不能正常进行，而 move_forward()中又有大量访问动态堆区空间的操作。

带着这样的怀疑，我在 move_forward()中写了大量对指针进行空指针判断，刷新 vector 向量数组元素个数的操作，结果还是如此。

渐渐烦躁的我开始冷静下来，陷入了沉思……

我想：我们使用 delete 操作只有在 remove_plant 和 remove_zombie 这两个函数中才有，而它们只在 damage 重载函数中被调用了，也就是当生命值降为 0 的时候，对象会被释放。有可能是这里起了冲突。于是我改变了思路，没有在 damage

函数中直接调用 remove_xx，而是在角色类中设置了一个标记 bool die ，如果生命值降为 0，die 被激活，然后在下一帧的开头，把所有 die 被激活的角色对象消亡。统一在帧起始位置消亡，可以有效减少冲突！就是这样！

改了 damage 函数后，还是时不时会出现这种情况。只不过出现的频率减少了，在窝瓜集火一堆僵尸的时候，还是会出现问题。

第二天，我重新开始 debug 之旅，反思之前的想法，其实我发现了我分析中一个错误的地方——线程冲突不会导致被释放的空间被访问。因为即使是调用既能函数，也是由 frame_forward 这一操作调用的，它仍然处于一个线程中，所以我之前的分析可能都是有误的。

这次我在出现 segfalut 的地方查看了 CPU 窗口，追踪被调用的函数，发现有三处地方出现问题：

1. remove_zombie 出错。原因很简单。对于一格内有多个僵尸需要 remove 的情况，在对数组遍历的情况下调用 erase 删去数组元素后，下标直接++判断下一个是会漏掉被前移的元素的，而且可能产生数组越界的情况。这是一个很容易犯下的错误，最终被我查出来了。

2. 西瓜投手的技能 water_split 出错。因为 water_split 的触发条件时 ATTACKING 即造成伤害，技能效果是对僵尸所在十字格内的所有僵尸造成溅射伤害，而溅射伤害也是伤害，也会调用 damage 函数，这样就会无限递归下去，直至指针访问出现错误或栈溢出。

3. 唯一可能导致线程冲突的地方，在豌豆射手被种植后，发射子弹后用户操作铲除植物，子弹对僵尸造成伤害的时候，由于植物已经被 delete 掉了，访问不到伤害者的地址 Plant_role*，解决方法是创建一个动态的虚拟对象（与原植物同名），让这个虚拟对象对僵尸造成伤害后，立即消亡，相应的方法也被用在第 2 点上（即创造一个没有技能的西瓜投手虚拟对象进行伤害）。

这个项目的 debug 花了我大量的时间，但我仍然不能保证程序的正确运行，不过这次项目对我的锻炼还是极大的。

2.2.5 程序运行总结

下图直观展示了各个数据结构之间的运行方式：

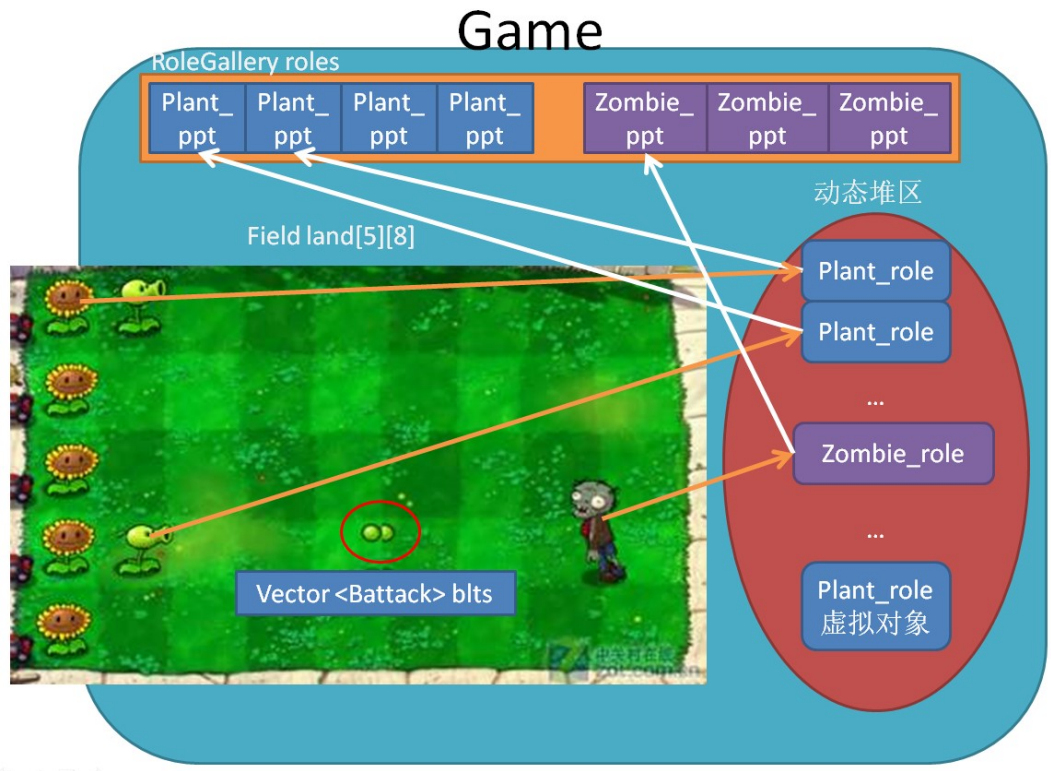

图 2.2 数据结构视图

三、代码框架

3.1 项目文件目录

本项目编写环境：**Windows 下 Dev-C++ 5.11 集成环境**，由于在 Windows 下编程，我们将字符编码集统一规定为 **GBK**。

项目目录如下图所示：

```
pvz
    |-- include  //源代码的头文件
    |    |-- property_define.h
    |    |-- role_define.h
    |    |-- game_system.h
    |
    |-- src  //源代码的实现部分
    |    |-- main.cpp
    |    |-- property_define.cpp
    |    |-- role_define.cpp
    |    |-- game_system.cpp
    |    |-- role_gallery.cpp
    |    |-- skill_definition.cpp
```

```
|   |-- XXX.o  //系统自动生成的.o 文件
|   |-- ......
|
|-- sysfile  //项目依赖文件
|   |-- plants.txt //存储植物和僵尸静态属性
|   |-- zombies.txt
|-- pvz.exe //可执行文件
|-- pvz.dev //工程文件
|-- ......
```

查看完整源代码的方式有：

1. 使用 Dev-C++，直接打开 pvz.dev
2. 到相应的目录里自己翻^_^

3.2 模块划分介绍

3.2.1 property_define.h

该头文件下包括如下声明和定义：

- 对植物类型的枚举类型 enum Ptype 的定义；
- 对被动效果结构体 struct PEffect 的定义；
- 对触发技能结构体 struct TSkill 的定义（包括触发类型 enum TSType）；
- 对僵尸状态结构体 struct Status 的定义（包括状态类型 enum STType）；
- 对植物子弹结构体 struct Bullet 的定义；
- 对植物静态属性类 class Plant_ppt 和僵尸静态属性类 class Zombie_ppt 的声明
- 对全局函数 PEffect _make_effect(string name, int role) 和 TSkill _make_skill(string name, int role)的声明，定义在 skill_definition.cpp

对于类的定义，全部在 property_define.cpp 中。

3.2.2 role_define.h

该头文件引用了 property_define.h 并且进行了如下声明和定义：

- 对植物对象类 class Plant_role 和僵尸对象类 class Zombie_role 进行了声明，定义在 role_define.cpp 中。

3.2.3 game_system.h

该头文件引用了 role_define.h 并且进行了如下声明和定义：

- 对图鉴类 RoleGallery 进行了声明，并且除了 string get_skill_desc(string name) 函数被定义在 skill_definition.cpp 中之外，其余定义都在 role_gallery.cpp。（因为 game_system.h 设计的声明很多，所以分了好几个.cpp 文件对其进行定义，这样更清楚一点。）
- 定义了子弹对象(struct BAttack)和地块(struct Field)结构体
- 对伤害类(class Damage)进行了声明，定义在 game_system.cpp
- 对游戏引擎类(class Game)进行了声明，除了与技能有关的函数定义在 skill_definition.cpp 中外，其余定义都在 game_system.cpp 中。

这里我们将技能定义有关的函数全部集中在 skill_definition.cpp 文件中，这样在进行代码扩写与复用的时候就非常的方便了。

3.2.4 代码的扩写与复用

在创建新角色的时候，或者为已有角色添加技能的时候，巧妙地运用代码框架的排版，很容易就可以创建一个新的对象了。

具体步骤如下，以写一个新植物为例：

1. 在\sysfile\plants.txt 中另起一行，写上植物的静态属性，其中技能以技能名称代替。

2. 如果拥有技能，那么去 game_system.h 中 Game 类声明的最后，对技能函数进行声明，建议函数名与技能名称一样。

3. 去 skill_definition.cpp，在全局函数_make_effect 或_make_skill 的定义中，补充技能的信息（如：效果冷却时间，技能触发时机）。

4. 在当前 cpp 文件下，在 string RoleGallery:: get_skill_desc(string name)中，根据已经写好的部分的格式，添加该技能的技能描述（用于图鉴展示）

5. 找到对应的重载函数(do_skill 或 do_effect)，根据已经写好部分的格式，添加该技能的调用。

6. 在文件的最后，添加已经声明好函数原型的函数定义，实现该技能。

我们看到新写一个植物（或僵尸），不需要调整已经写好的底层代码，只需

在 skill_definition.cpp 中添加好技能属性，技能就可以正确地被触发了，这样使得以后游戏功能的拓展变得极其方便。

3.2.5 程序运行调用结构：

该项目程序运行结构如下图所示：

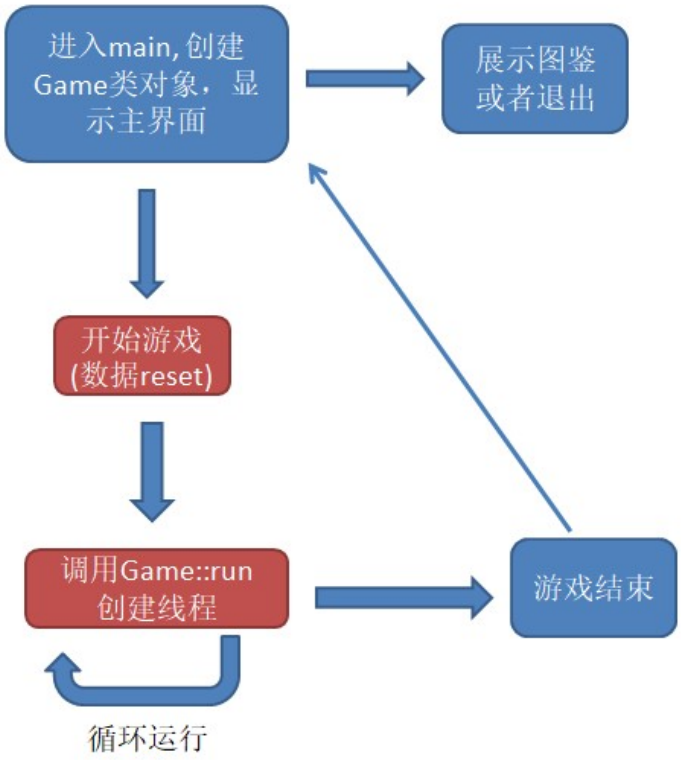

图 2.3 程序运行视图

四、游戏规则

4.1 基本规则

玩家开始游戏后，需要种植植物保护草地，抵挡僵尸的一波又一波进攻，最终坚持尽可能长的时间。

游戏初始阳光 100，过程中，每隔 6.25s，天空中会掉落 25 阳光（将被自动拾取），植物具有一定的冷却时间，游戏开始时所有植物处于冷却完成状态，种植后会冷却时间会被重置，玩家不能将植物种在已经被种植植物的地块上，也不能在阳光不足，冷却时间未到的情况下种植植物。

4.2 植物设计一览

本项目一共设计了 13 个植物，如下所示：

1. **豌豆射手** 攻击型植物

最大生命 300，冷却时间 30 帧，阳光消费 100

子弹情况：拥有子弹，子弹速度 1 帧 1 格，攻击力 20，子弹发射间隔 4 帧

技能情况：没有技能

2. **向日葵** 特效型植物

最大生命 300，冷却时间 30 帧，阳光消费 50

子弹情况：没有子弹

技能情况：拥有被动特效 sun_generating: 等待时间 96 帧，初始种植后等待时间 12-40 帧随机，效果：阳光+25

3. **寒冰射手** 攻击型植物

最大生命 300，冷却时间 30 帧，阳光消费 175

子弹情况：拥有子弹，子弹速度 1 帧 1 格，攻击力 20，子弹发射间隔 4 帧

技能情况：拥有触发技能 slow_you_down：触发时机 ATTACKING，被攻击者获得 40 帧 SLOW 状态。

4. **双发射手** 攻击型植物

最大生命 300，冷却时间 30 帧，阳光消费 200

子弹情况：拥有子弹，子弹速度 1 帧 1 格，攻击力 40，子弹发射间隔 4 帧

技能情况：没有技能

5. **西瓜投手** 攻击型植物

最大生命 300，冷却时间 30 帧，阳光消费 300

子弹情况：拥有子弹，子弹速度 1 帧 1 格，攻击力 80，子弹发射间隔 8 帧

技能情况：拥有触发技能 water_split：触发时机 ATTACKING，被攻击者十字范围内所有敌人（不包括自己）受到 26 点溅射伤害。

6. **坚果墙** 防御型植物

最大生命 4000，冷却时间 120 帧，阳光消费 50

子弹情况：没有子弹

技能情况：没有技能

7. **高坚果** 防御型植物

最大生命 8000，冷却时间 120 帧，阳光消费 125

子弹情况：没有子弹

技能情况：没有技能

8. **窝瓜** 炸弹型

最大生命：9999，冷却时间 120 帧，阳光消费 50

子弹情况：没有子弹

技能情况：拥有触发技能 pump_exploding：触发时机 ATTACKED，消灭被攻击者本格所有僵尸

9. **土豆地雷** 炸弹型

最大生命 300，冷却时间 120 帧，阳光消费 25

子弹情况：没有子弹

技能情况：拥有被动特效 potato_activating，等待时间 60 帧，效果：失去该特效，获得技能“pump_exploding”。

10. **樱桃炸弹** 炸弹型

最大生命：9999，冷却时间 150 帧，阳光消费 150

子弹情况：没有子弹

技能情况：拥有被动特效 cherry_exploding，等待时间 2 帧，效果：自己和自己临近九宫格内所有僵尸消灭。

11. **火爆辣椒** 炸弹型

最大生命：9999，冷却时间 150 帧，阳光消费 125

子弹情况：没有子弹

技能情况：拥有被动特效 chilly_exploding，等待时间 2 帧，效果：自己和自己所在排所有僵尸消灭。

12. **大蒜** 效果型

最大生命：400，冷却时间 30 帧，阳光消费 50

子弹情况：没有子弹

技能情况：拥有触发技能 smell_attack，触发时机 ATTACKED，将僵尸移动到临近排同等列的位置上。

13 **地刺** 攻击型

不看成植物，看做一个地块特效。

4.3 僵尸设计一览

本项目一共设计了 7 种僵尸，如下所示：

1-3 **僵尸、路障僵尸、铁桶僵尸**：生命值分别为 200,570,1300，移动速度均为 20 帧前进一格，攻击力均为 25

4 **摇旗僵尸**：生命值 200，移动速度 17 帧前进一格，攻击力 25

5 **橄榄僵尸**：生命值 1500，移动速度 8 帧前进一格，攻击力 25

6 **读报僵尸**：生命值 350，移动速度 20 帧前进一格，攻击力 25，技能：newspaper_crazy，触发条件 ATTACKED，当生命值<=200，修改移动速度为 8 帧前进一格，并失去该技能。

7 **撑杆僵尸**：生命值 550，移动速度 10 帧前进一格，攻击力 0(有撑杆时不会吃植物)，技能 bar_jumping，触发条件 ATTACKING，速度修改为 20 帧前进一格，攻击力修改为 25，且如果被攻击者不是高坚果，移动至下一格。

4.4 僵尸释放规则

本项目模拟的是白天无尽模式，进攻的僵尸数量与进攻波数有关。在第一波的进攻之前，防守方有 28.5s 的准备时间，此后开始第一波攻势。

对于每一波进攻，僵尸方有 5 次进攻动作，其中第一次的进攻动作（除了第一波）是攻势较为强大的集火进攻，在第 1-4 次进攻动作开始时，防守方如果将该次出现的僵尸全部击杀，则立即进入下一次进攻，否则在 45s 时间之后，僵尸的下一次进攻动作将强制开启。一直到第 5 次进攻动作被激活。

当第 5 次进攻动作被激活后，该波将被持续到场上的僵尸全部阵亡，随后屏幕将输出提示“一大波僵尸正在来袭…”，防守方有 5s 的准备时间，随后开始进行下一波进攻。

每一波进攻的僵尸数量如下所示，僵尸被释放的位置是随机决定的：

第一波：

进攻 1：僵尸 x1

进攻 2：僵尸 x2

进攻 3：僵尸 x2，路障僵尸 x1

进攻 4：僵尸 x3，路障僵尸 x2

进攻 5：僵尸 x4，路障僵尸 x3

第二波：

进攻 1：摇旗 x1，僵尸 x6，路障 x3

进攻 2：僵尸 x6，路障 x2

进攻 3：僵尸 x6，路障 x2

进攻 4：僵尸 x6，铁桶 x2

第三波：

进攻 1：摇旗 x1，僵尸 x7，路障 x4，铁桶 x3，读报 x2

进攻 2：僵尸 x7，路障 x3，铁桶 x2，读报 x1

进攻 3：僵尸 x7，路障 x3，铁桶 x2，读报 x1

进攻 4：僵尸 x7，铁桶 x3，橄榄 x1，读报 x1，撑杆 x1

进攻 5：僵尸 x7，铁桶 x3，橄榄 x1，读报 x1，撑杆 x1

第 n 波（n>=4）：

进攻 1：摇旗 x1，僵尸 x8，路障 x(n+1)，铁桶 xn，橄榄 x(n-2)，读报 x(n-1)，撑杆 x(n-1)

进攻 2：僵尸 x8，路障 xn，铁桶 x(n-1)，读报 x(n-2)，撑杆 x(n-2)

进攻 3：僵尸 x8，路障 xn，铁桶 x(n-1)，橄榄 x(n-2)/2，读报 x(n-2)，撑杆 x(n-2)

进攻 4：僵尸 x8，铁桶 xn，橄榄 x(n-1)/2，读报 x(n-2)，撑杆 x(n-2)

进攻 5：僵尸 x8，铁桶 xn，橄榄 x(n-1)/2，读报 x(n-2)，撑杆 x(n-2)

4.5 计分方式

在击杀僵尸和通过波数的时候，将获得加分：

击杀僵尸获得的积分如下：

僵尸：10p 路障：12p 铁桶：15p 橄榄：18p 读报：12p 撑杆：15p

其次，**通过第 n 波时**，将可以获得 **400+((n/2)*50)**分，如果在此波内**没有植物被僵尸击杀或者被故意铲除**（爆炸类植物爆炸不算），则可以额外获得 **n*50** 分。

游戏结束时，将显示获得的分数和最终的坚持波次。

五、程序运行展示

5.1 主界面

主界面如下图所示，按’w”s’上下移动光标，按 enter 进入相应的菜单。

图 2.4 程序主界面

5.2 图鉴一览

主界面选择第二个菜单后进入，则可选择进入植物&僵尸图鉴，操作方法类似：

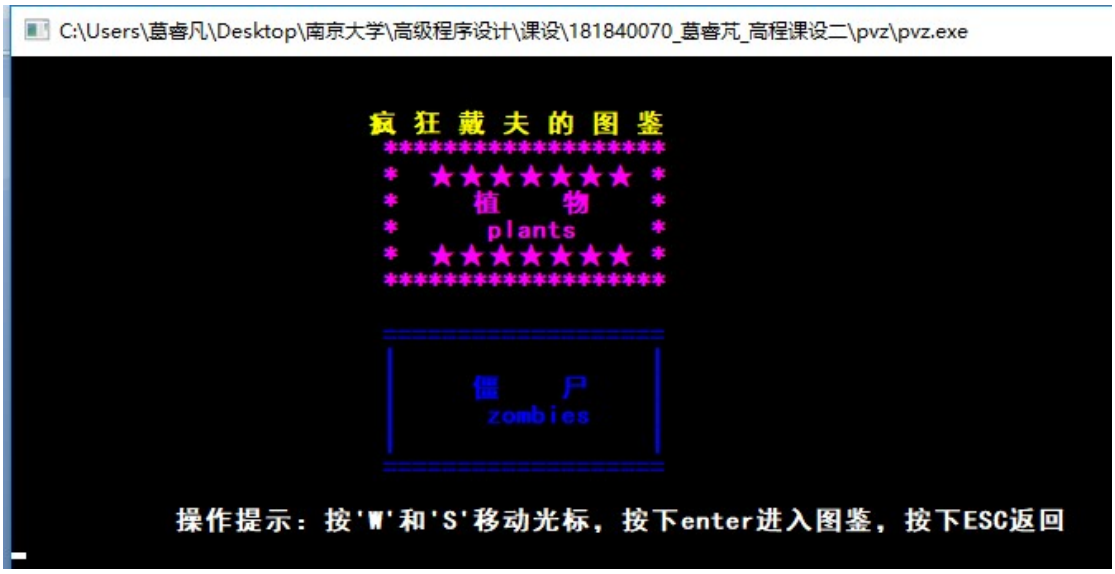

图 2.5 图鉴界面

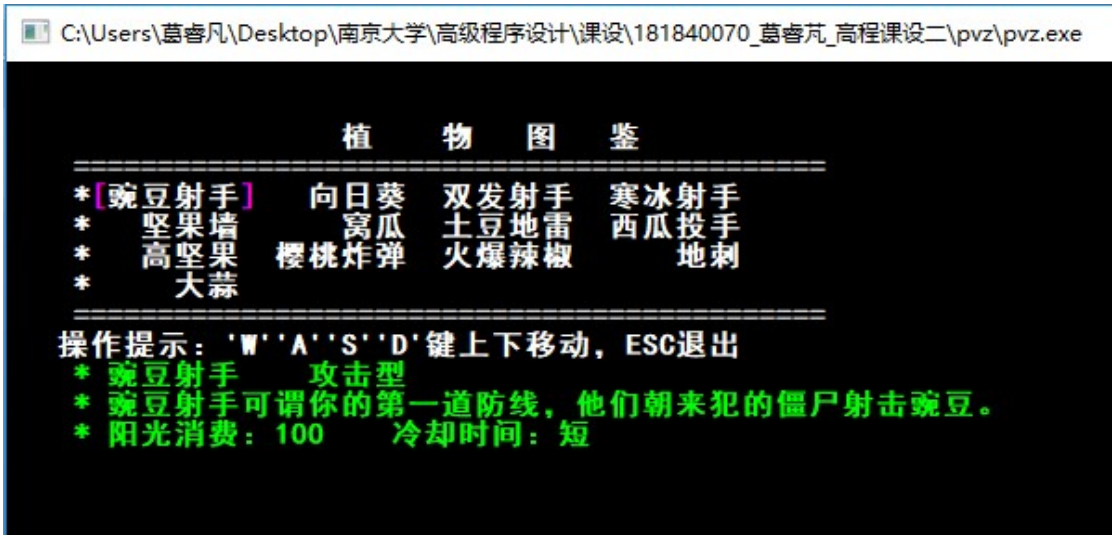

图 2.6 植物图鉴界面

5.3 进入游戏

5.3.1 界面展示

游戏界面由商店、草坪、得分栏三部分构成：

植物商店：第一行是所获得的阳光总数，然后的几行就是商店中植物的具体内容，标记为红色的植物是正在冷却的植物，后面的括号数值是它的冷却进度，标记为黄色的植物是冷却完毕但阳光数量不足以种植的植物，这两种植物都不能被种植。标记为绿色的植物是冷却完毕且阳光数量充足的植物，可以被种植。商店的下面两行是所选地块和植物的索引，被选中的地块和植物将被以紫色标记标识。右侧的提示符将提示非法种植操作（图中未显示），以及当前波次信息。

草坪：草坪是一个 5x8 的矩阵，每一格中将以绿色标记处植物和植物信息，以及他们的子弹，以红色标记出僵尸和僵尸信息，多于一个僵尸一格将会显示走在最前面的僵尸的信息和僵尸数量。僵尸被减速，其状态将会标记成蓝色，显示“慢”字样。地刺会以白色标出（图中未显示）

得分栏：第一排显示当前得分，第二排显示最近一次得分的原因。右侧是波次信息。

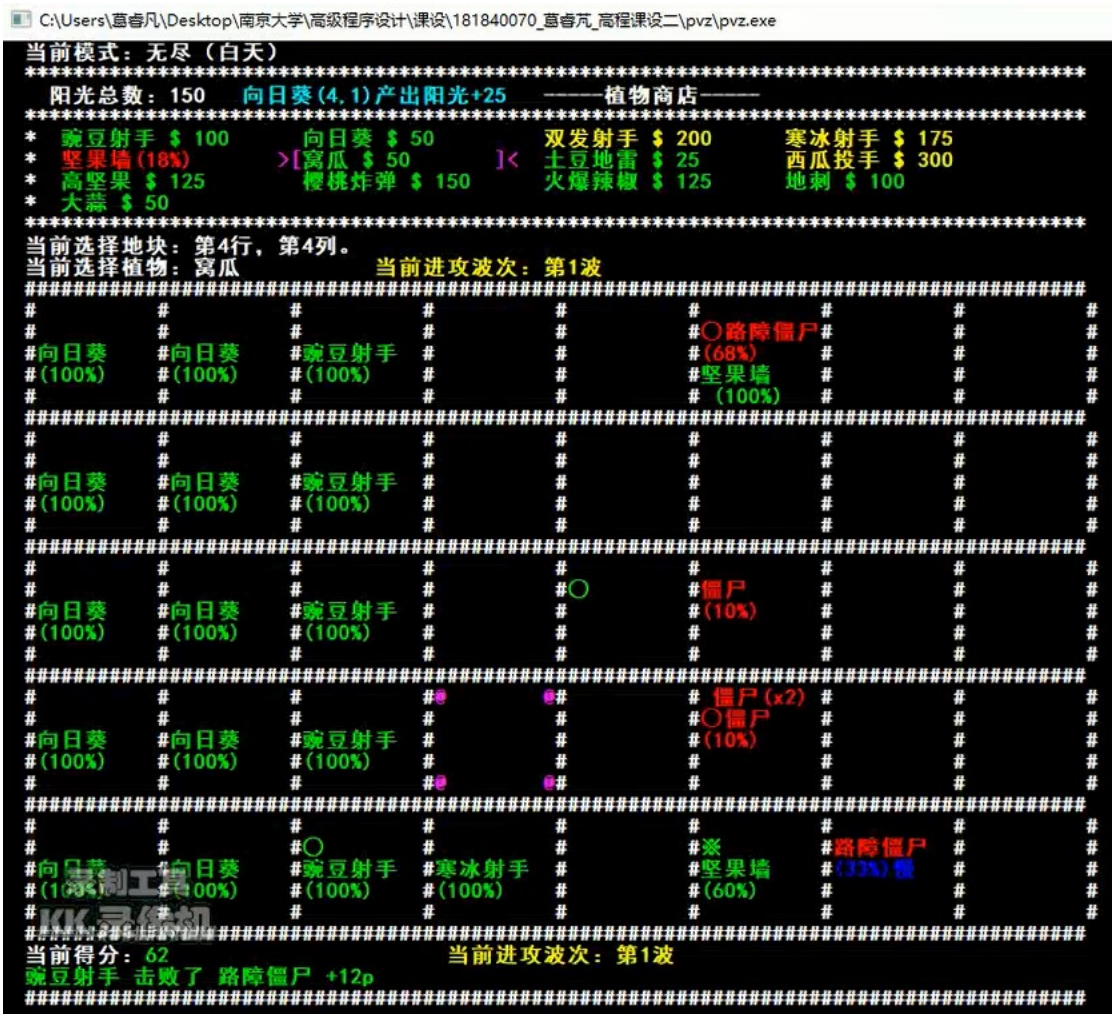

图 2.7 游戏界面展示

5.3.2 操作方法：

按下’a”s”d”w’选择上下左右地块，按下’j”k”l”i’选择上下左右植物，按下空格种植，无法种植会给出提示信息，按’c’铲除所选地块植物。

六、后记

本项目是我学习编程以来耗时最长的一个项目，在设计和 debug 上面均有着相当大的难度，也极大地锻炼了我的能力。

感谢助教和老师看完了我的报告，对于本项目中的不足之处，还请各位谅解！一并感谢各位对本项目提出的宝贵意见和建议！

181840070 葛睿芃

2019 年 11 月 17 日