植物大战僵尸PVZ 第一阶段

课程设计分析

课程设计目标

仿制一个基于控制台的植物大战僵尸小游戏；游戏规则大致如下

- 前院场景（纯草地无水池）+白天（系统会产生自然光并自动收集）
- 无尽模式（即僵尸来到花园最左侧游戏才会结束），展示记分牌

课程设计基础内容

本次设计要实现基础的游戏框架如下：

- 庭院至少三行七列
- 系统每隔一定时间产生并自动收集自然光
- 实现普通僵尸，包括生命值、攻击力、速度三种属性
- 实现豌豆射手，包括购买所费阳光数、生命值、攻击力
- 实现植物的购买、地块选择和植物种植
- 实现记分牌和游戏失败的机制

设计分析

板块设计思路：让各个功能模块尽可能独立，通信和互相依赖是明确的。

主要的板块分为植物、僵尸、和花园。

花园

花园是整个游戏的基础界面，最基本的植物和僵尸的显示、分数的显示都由花园来完成。更进一步，购买植物后种植位置的选择，植物和僵尸攻击彼此时所依赖的位置信息也依赖于花园。所以花园的功能在本次设计中最为复杂，主要是在控制台的输出和交互（鼠标），以及处理和植物/僵尸关于位置信息的交互上。由于显示信息和时钟信息是全局共享的，为了方便，商店的信息以及游戏整体状态的刷新也交给花园实现。

这里需要特别注意的是，花园关于位置信息的功能应当与植物/僵尸的攻击功能较好的区分开；即交互仅限于通知植物和僵尸攻击对象的信息，具体的攻击/受伤操作不应该由花园来完成，花园只是提供位置信息供植物和僵尸访问，根据是否会发生攻击取出指针后，剩下的攻击由植物和僵尸各自完成。

> 这里花园信息的吞吐量还不算很大，还有一种实现的方式是将植物和僵尸的信息直接交给对方，不经由花园处理，但是这样在僵尸动态移动的过程中，僵尸需要不断更新对植物距离的认知（只有相邻才会开始攻击即每一次移动都要判断是否相邻），这样大大增加了二者交互的频率，所以最后没有采用

植物

为了在花园中统一表示所有植物，设计了一个总的父类Plant，其他植物都由plant继承而来。植物中统一的属性有生命值，购买需要的阳光数；攻击值取决于是否是攻击性的植物；为了区分不同植物，设置了plant_name属性。

植物被攻击时需要获取僵尸的攻击值

Shooter & Bullet

本次需要实现简单的射手，射手通过子弹对僵尸攻击。子弹的移动与攻击和僵尸的移动与攻击类似；以一个游戏周期（回合）为单位来定义速度。子弹的攻击值取决于所属射手的攻击值，子弹与僵尸相遇时子弹发动攻击。子弹的显示与花园独立，避免互相覆盖冲突的情况。

植物是否产生子弹，取决于所在列上是否有僵尸。当前是若有，则在花园的提示下每回合增加一个子弹。

僵尸

僵尸类统一的属性有生命值、攻击值和速度。与植物不同，僵尸有移动的功能，所以每一轮刷新时都要与花园进行位置更新的信息交互。同时僵尸遇到植物时会被阻塞而不再移动

核心类的设计展示

花园

```
class GardenBoard
{
    friend class Plant;
    friend class Zombie;
    friend class Shooter;
    friend class Bullet;
    //花园承担了与所有对象交互的功能，需要频繁访问他们的数据，因此设置成友元有助于提高程序运行效率
private:
```

```
    int row_total;//3 0~2
    int col_total;//8 0~7
    int sun_deposit;//拥有的太阳数 暂时公开
    int point_cnt;//计分
    vector<int> zombie_cnt;//现有僵尸总数  根据每一行来记录
    int zombie_max;//限定僵尸总数 可以根据游戏的难度来调整
    int sunflower_cnt;//向日葵数量统计，阳光生成速率与之成正比
public:
    //int sunflower_cnt;
    //int sun_deposit;
    HANDLE hOutput;
    HANDLE hIn;
    vector<vector<void*>> garden_pos;
    GardenBoard(int row_total, int col_total,HANDLE hIn,HANDLE hOutput)
:point_cnt(0), sun_deposit(100),zombie_max(4),
        sunflower_cnt(0)
    {
        for (int i = 0; i < row_total; ++i)
        {
            vector<void*> tmp(col_total);
            garden_pos.push_back(tmp);
        }//此处填满了NULL指针
        zombie_cnt.resize(row_total, 0);
        this->row_total = row_total;
        this->col_total = col_total;
        this->hIn = hIn;
        this->hOutput = hOutput;
    };
    void print_garden();//打印面板
    void SetPos(HANDLE HOutput,int x, int y);
    bool ClearConsole(HANDLE hOut);//refresh
    void refresh_state(HANDLE& hOutput, HANDLE& hIn, DWORD start);//刷新位置 让植物和僵尸互相攻击 位置判断在花园中完成 消除死去的植物和僵尸并且对僵尸计分
    void generate_sun();//refresh时更新太阳数，一开始按一定数目递增 增加向日葵后与向日葵数成正比
    void get_score(int score) { point_cnt += score; }//更新记分牌
    void zombie_counting(int cnt,int row){ zombie_cnt[row] += cnt; }//统计现有僵尸数
    bool random_generate_zom();//生成僵尸 生成僵尸 可以根据总的僵尸数和每行的僵尸数调整
    void open_shop(HANDLE& hIn);//在捕获相应键盘之后 购买植物 这里的位置与row/col_total保持一致 是格子数的意思
};
```

植物

```
class Plant //整体的父类
{
    friend class GardenBoard;
public:
    int row;
    int col;//位置信息
    int life;//生命值
    int sun_price;//购买所需阳光数
    char type;//供花园区分植物和僵尸
    string plant_name;//细分不同的植物类别
    Plant() :row(0), col(0), life(30), sun_price(0), type('p'),
plant_name("plant") {};
```

```
    int get_col()const { return col; }//供外界调试时获取列坐标
    void get_hurt(Zombie* zombie);//被僵尸攻击
    int price()const { return sun_price; }//供外界调试时获取购买植物所需阳光数
};

class Sunflower :public Plant
{
    friend class GardenBoard;
public:
    Sunflower(int r, int c) {
        row = r; col = c; plant_name = "Sunflower";
        sun_price = 10;
        life = 20;
    }

};

class Shooter :public Plant
{
    friend class GardenBoard;
    friend class Bullet;
private:
    int attack;
public:
    vector<Bullet*> bullet_set;//暂时public 之后给garden调用
    Shooter(int r, int c) :attack(shooter_attack_val)
    {
        row = r;
        col = c;
        life = shooter_life_val;
        sun_price = shooter_sun_price_val;
        plant_name = "Shooter";
        bullet_set.clear();
    }
    void attacking();//通过产生子弹来攻击
    int get_row()const { return row; }
    int get_col()const { return col; }
};

class Bullet
{
    friend class Shooter;
    friend class GardenBoard;
private:
    int row;
    int col;
    int attack;//子弹的攻击值由植物（射手）决定
    int fly_speed;//每回合移动的格子数
public:
    Bullet() :row(0), col(0), attack(9),fly_speed(bullet_fly_speed) {};
    Bullet(Shooter* shot,int fly)
    {
        row = shot->row;
        col = shot->col;
        attack = shot->attack;
        fly_speed= fly;
    }
    void move(vector<vector<void*>>& garden_pos);//子弹自己更新位置
```

```
    void print_bullet(HANDLE hOutput);//子弹自行打印位置
    void attacking(Zombie* zombie);//子弹攻击僵尸
    int get_col()const { return col; }//供外界调用
};
```

僵尸

```
class Zombie
{
    friend class GardenBoard;
private:
    int life;//生命值
    int attack;//攻击力
    int speed;//移动速度 同样是一回合移动的格数
    //有多种僵尸之后再命名

public:
    int row;
    int col;
    char type;
    Zombie(int r, int c) :life(25), attack(5), speed(1), type('z')
    {
        row = r; col = c;
    }
    Zombie(int r, int c, int _life, int _att, int _speed) :type('z')
    {
        life = _life; attack = _att; speed = _speed; row = r; col = c;
    }
    void get_hurted(int attack_value);//便于之后增加其他带有攻击性的植物 所以只传入攻击值
    bool move(vector<vector<void*>>& garden_pos, int x0, int y0);//自行移动
    int attacking()const
    {
        return attack;
    }//获取攻击值
    int get_row()const { return row; }
    int get_col()const { return col; }//获取位置
};
```

程序的亮点

可扩展性强

在设计时，考虑难度调整，增加新类的问题。如考虑到攻击比例和速度，设置的一些全局变量可供以后开局设置难度时调整。

```
const int X_INTERVAL = 15;
const int  Y_INTERVAL = 7;
extern int shooter_attack_val;
extern int shooter_life_val;
extern int shooter_sun_price_val;
extern int bullet_fly_speed;
```

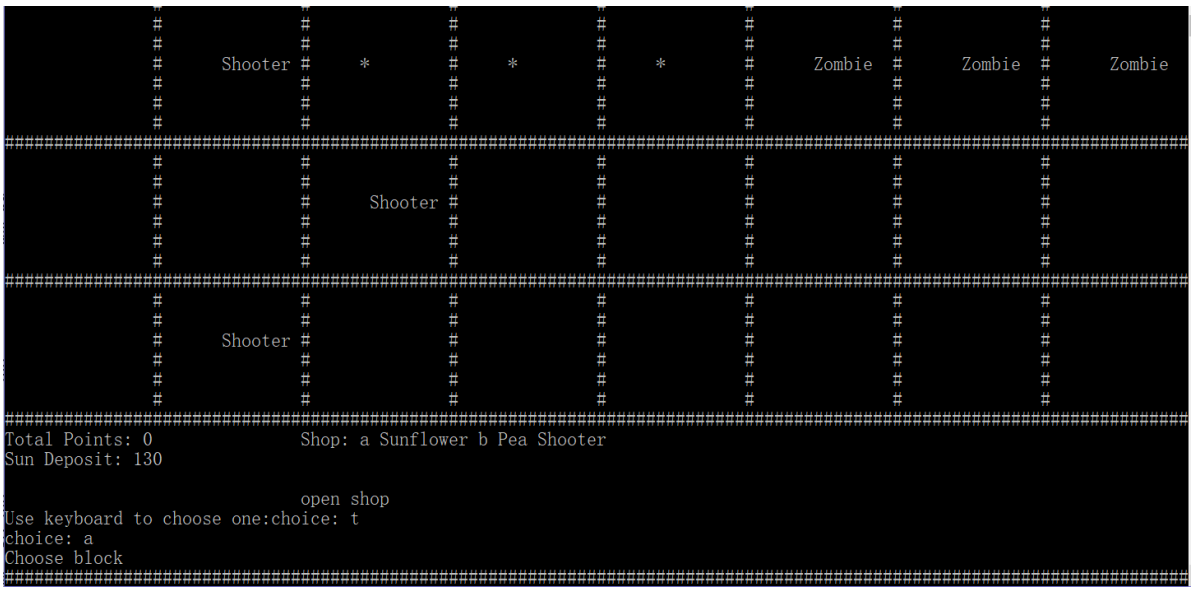
同时在类的设计上考虑到之后可能使用的继承，以及花园API的通用性。如所有植物和僵尸都可以从基础的Plant类和Zombie类继承而来；花园地块中的指针是通用类型；植物和僵尸互相攻击留的接口可以进一步拓展。

用户友好

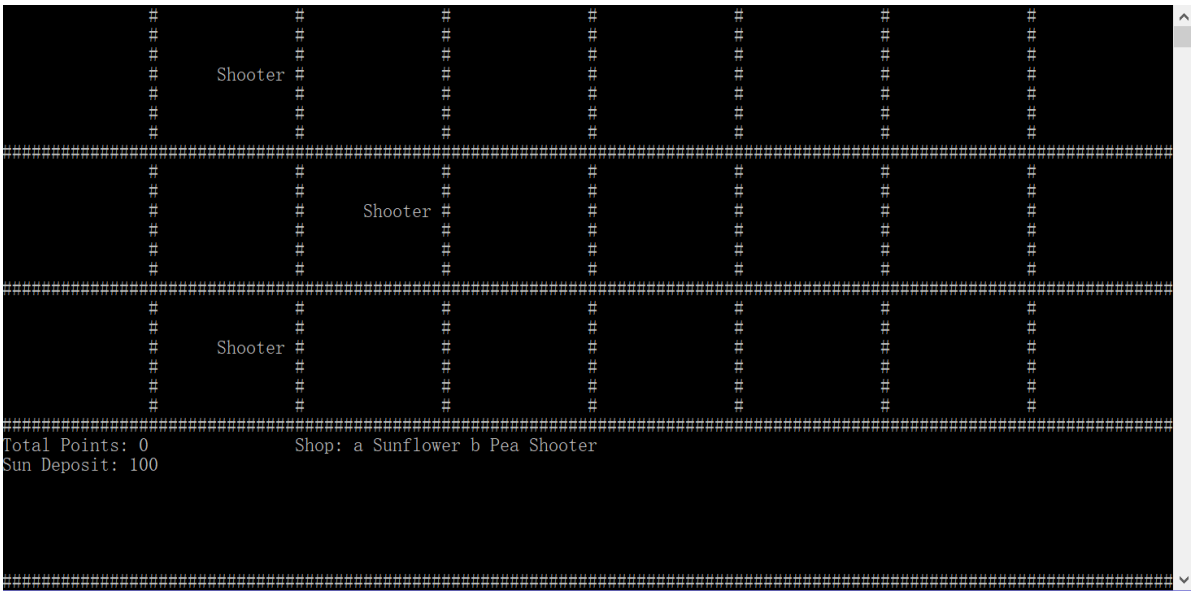
用户在使用商店时游戏处于冻结状态，留有思考时间。

与课设中提供的参考实现不同，地块的选择用直接点击鼠标来完成选择，并且会回显选择的地块信息供用户确认

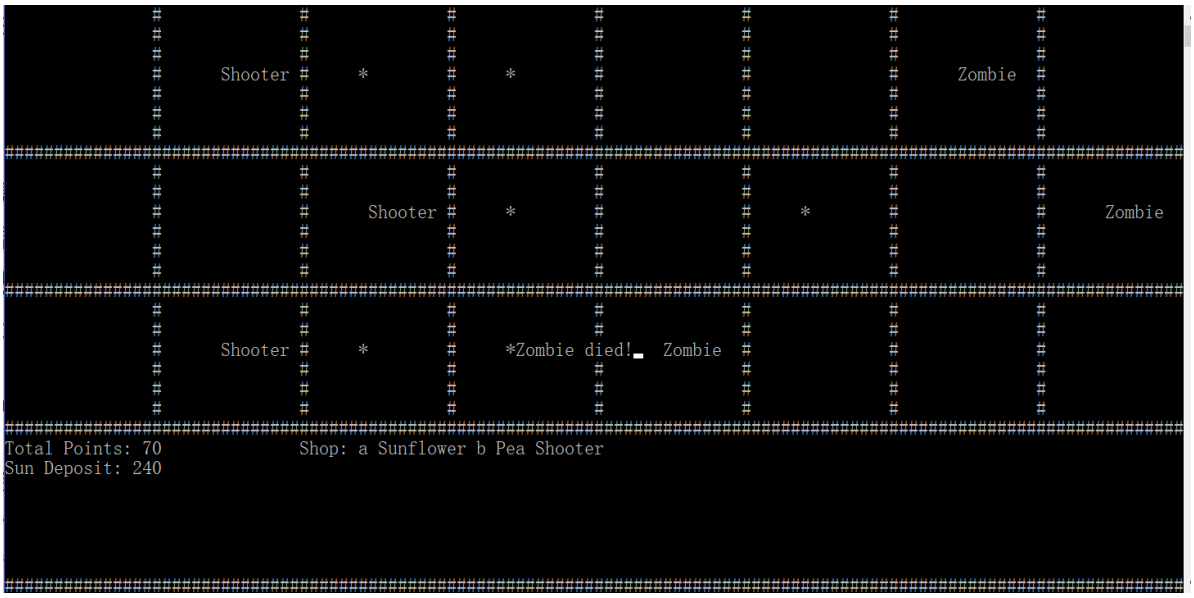
在购买的过程中有误触也没关系，直到输入合法才会选定购买的植物类型



提前给用户准备好了初始化的植物，避免因为来不及种植而落入下风



僵尸死亡时会有提示

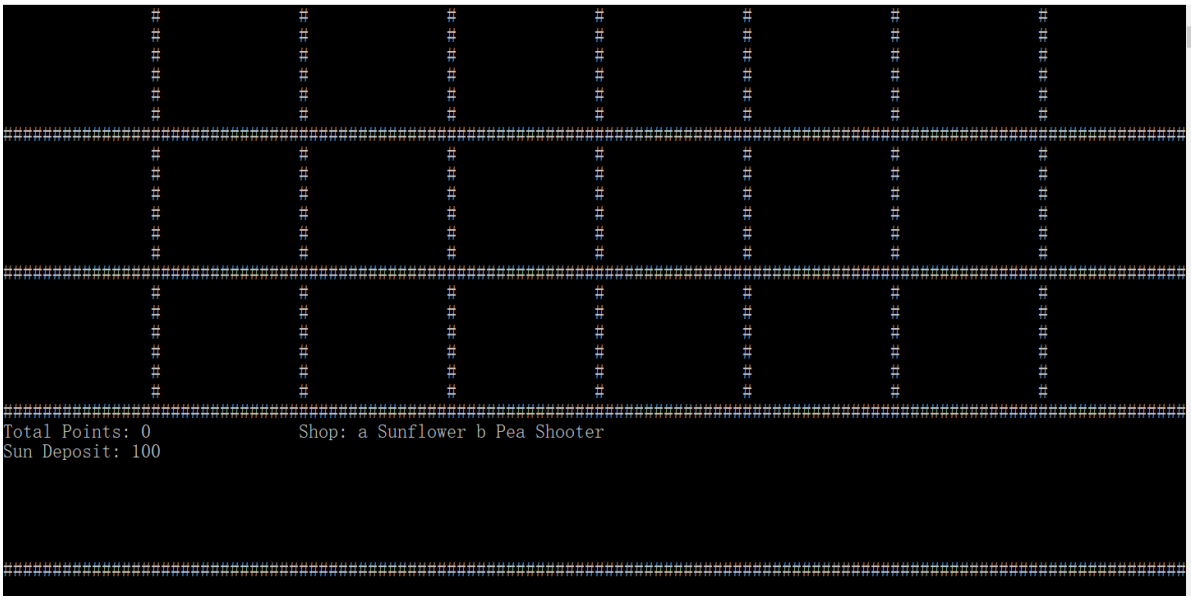


僵尸随机生成时会考虑当前行已有的僵尸数量和全局已有的僵尸数量，不会一直大量生成在同一行，平衡了难度

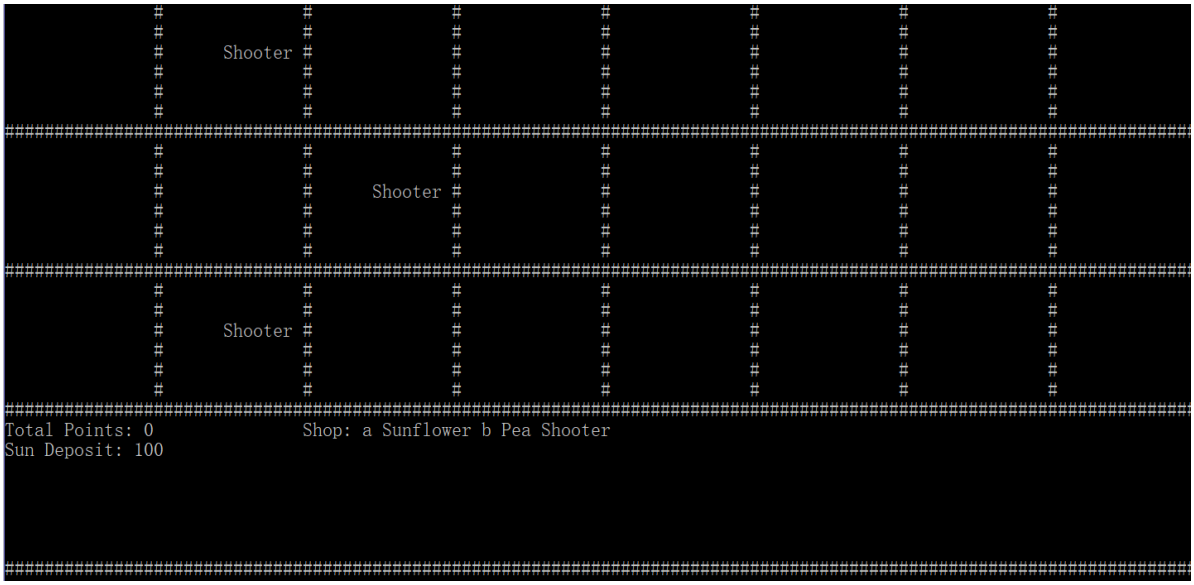
```
    int row_ini = rand() % row_total;
    for (int i = 0; i < 2*row_total&& garden_pos[row_ini][col_total - 1]
        && zombie_cnt[row_ini] >= zombie_max / 2 != NULL; ++i)
        //避免一行集中太多僵尸
    {
            row_ini = rand() % row_total;
    }
```

程序的运行方法

打开exe或者在项目按下ctrl+F5

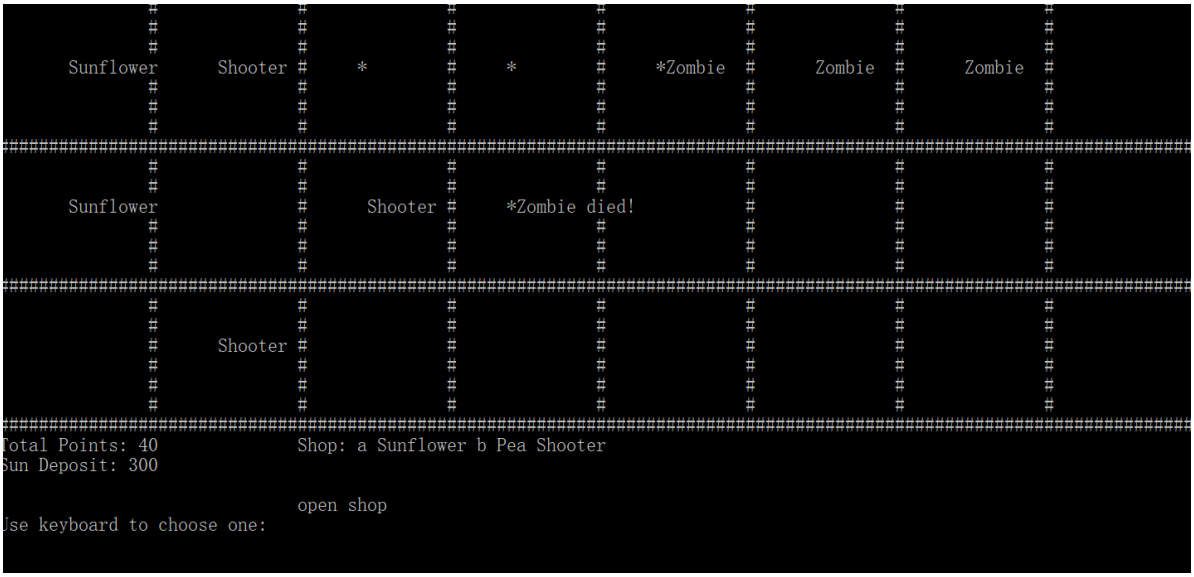


双击回车进入初始化的花园：



在exe端，需要按下键盘任意按键继续驱动游戏，一次敲击能让游戏运行两个回合；在项目新生成的控制台，可以敲击键盘或者移动鼠标，一次这样的操作可以让游戏运行三个回合。如果希望游戏暂停使用商店，可以不做任何键盘鼠标输入。

暂停后需要商店：单击鼠标，按下键盘b（或者双击键盘b），根据展示的shop；按下键盘上对应的字母选择植物，然后用鼠标点击希望种植的地块，单击选择，摁下回车键确认；如果第一次点错了，在按下回车键前都可以通过双击新位置来更改位置。



遇到的问题与困难

控制台应用的设计

https://zhuanlan.zhihu.com/p/148894267 对控制台有详细的介绍

输出上，从以下博客当中学习了相关window.h的API用法

https://www.cnblogs.com/flowingwind/p/8159035.html

https://blog.csdn.net/liluo_2951121599/article/details/66474233

采用的方法是先定位光标再定向的输出；美中不足的是会受到操作台显示大小的影响 需要在不同的平台提前设置

清空：一开始采用system("cls"),出现了句柄无效的问题；后来采用了填充空格并且把光标移到(0,0)的方式

读取鼠标位置和键盘状态

https://www.shuzhiduo.com/A/WpdKwD7ndV/

解锁鼠标的读取

https://blog.csdn.net/bnb45/article/details/8042819

头文件的关系

出现了互相包含的情况，为了方便把游戏相关所谓类的定义放在了同一个头文件

子弹和僵尸相对移动

完成子弹 一开始采用的是休眠 需要改成其他部分保持不动但是子弹在移动的状态；发现无法实现，把速度交给花园，由花园实现刷新。 花园提醒植物产生子弹

僵尸的移动和子弹的移动有相似之处，由于时间单位由花园决定；所以僵尸和子弹的移动刷新由花园完成，速度则是僵尸和子弹自身的性质

找到一个可以在vector中一边遍历检查一边删除的方式 虽然想用队列但是需要遍历所以不行 另外为了防止意外情况需要遍历来检查是否攻击僵尸和越界 其实正常情况下可以不用

子弹和僵尸错位了？发现是遍历删除中if-else情况分类不当导致迭代器错位；

游戏错误的结束

僵尸到左侧向右侧数第二个格子就结束游戏: 注意刷新和显示的先后关系

可供改进之处

- 窗口大小是写死的
- 选购:目前是程序写死的选项，考虑能不能自动生成：采用宏?

商店的输出目前一样写死

- 移动鼠标或敲击键盘 才能捕获事件继续游戏；为了等待商店的调用牺牲了游戏自动运行的特性，而且根据exe和新生成的控制台的不同，驱动游戏运行的方式还不一样（有没有能让游戏捕捉中断的方式？）