高级程序设计课程设计二实验报告

1.实验目标

实现一个GUI版本的植物大战僵尸游戏

2.实验内容

1.游戏功能

- 和实验二的实现基本一致，实现了2种僵尸（普通僵尸、路障僵尸）和4种植物（豌豆射手、坚果墙、寒冰射手、向日葵），主要实现了游戏功能的GUI化。

2.GUI增强

- 为僵尸的行进过程增加了动画（gif）。
- 寒冰射手的会冰冻僵尸的行进动画。
- 向日葵产生的阳光不再是自动增加，而是会在地图上显示，需要点击后增加。

3.实验过程

本实验使用qt进行图形化，熟悉qt的各种机制是完成实验中花费时间较多的部分。

1.框架设计

- 本实验选择使用了QGraphicsView-QGraphicsScene-QGraphicsItem框架，一个scene是多个item（图元）的容器，而view是scene的显示。
- 本游戏中只有一个场景，就是游戏的进行场景，使用一个view对其进行显示。
- 游戏主体类Game，程序的所有行为由其成员函数组成。

```
class Game
{
private:
    GameScene *gameScene;
    QGraphicsView *view;
    QTimer *timer;
public:
    Game();
    void run(); //游戏执行
};
```

- 在gameScene中，完成游戏的主要逻辑，timer为定时器，定时向gameScene发送消息。
- run函数，其中执行的两个connect函数由定时器定时发送消息给gameScene的nextFrame和advance函数。nextFrame主要复用了实验二中的内容，有部分修改以适合qt框架。

```
void Game::run()
{
```

```
    gameScene = new GameScene();  //场景
    view = new QGraphicsView; //视图
    timer = new QTimer;
    gameScene->setSceneRect(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
    gameScene->run();

    view->setScene(gameScene);
    view->setMouseTracking(true);
    QObject::connect(timer, &QTimer::timeout, gameScene,
&QGraphicsScene::advance);
    QObject::connect(timer, &QTimer::timeout, gameScene,
&GameScene::nextFrame);
    //QObject::connect(timer, &QTimer::timeout, gameScene, &GameScene::end);
    timer->start(50);

    view->show();

}
```

2.类设计

- 类设计基本按照实验二的设计，分为shop类（管理所有的植物购买和铲子），info类（管理帧数信息和点击信息），map类（管理地图），plant类（管理植物），zombie类（管理僵尸），shovel类（主要用于显示），card类（植物商店中的卡片，也用于显示）
- 类之间的交互主要通过gameScene来实现，gameScene管理了该场景上的所有元素。
- 此外，在类之间存在着一些不通过gameScene的直接交互，主要通过item的scene()函数获取同一场景上的其他item，来进行item之间的直接交互，这一交互破坏了整体通过gameScene来实现对象间交互的结构，但是由于在同一个scene下的所有item理应有相互交互的功能，因此这样能够大大简化设计，一些对象间的交互（如植物和僵尸、子弹与僵尸的碰撞）可以在对象的成员函数间完成，无需将参数回传至gameScene。

3.具体设计

- 在gameScene的mousePressEvent中，实现对于卡片选择或是解除选择的判断。

```
void GameScene::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    .......
    QGraphicsScene::mousePressEvent(event);
    vector<Card> *currentShop = shop->cards();
    if(event->button() == Qt::LeftButton)
    {
        for(int i = 0;i < (int)currentShop->size(); i++)
        {
            //如果更改了选择的植物，将原先选中的解除选中
            .......
        }
        click(event->scenePos());
    }
    if(event->button() == Qt::RightButton)
    {
        info->unclick();
        shop->unclick();
    }
}
```

- 由于需要使选中的植物跟随鼠标移动，在game中已经将view设置为追踪鼠标轨迹了。

```
view->setMouseTracking(true);
```

然后，由于需要跟踪整个屏幕，因此需要重写gameScene的鼠标移动事件，使info的坐标跟随鼠标移动。

```
void GameScene::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
{
    QGraphicsScene::mouseMoveEvent(event);

    info->setPos(event->scenePos());
    update();
}
```

- 以僵尸为例，僵尸类继承QGraphicsItem，重写paint和boundingRect函数来显示僵尸的图片，同时，定义僵尸的Type，以便在植物处于僵尸对象进行直接交互

```
class Zombie : public QGraphicsItem
{
protected:
    QMovie *movie;
    ZombieInfo info;    //僵尸信息
    ....
public:
    enum { Type = UserType + 1 };
    Zombie(int frame, QPointF startPos, ZombieInfo info);
    virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem
*option, QWidget *widget) override;
    QRectF boundingRect() const override;
    int type() const { return Type; }
    ...
}
```

重写的piant函数实现的显示，能够显示僵尸的移动动画以及在路障僵尸生命值小于一半时将图片变为普通僵尸。

```
void ConeheadZombie::paint(QPainter *painter, const QStyleOptionGraphicsItem
*option, QWidget *widget)
{
    Q_UNUSED(option)
    Q_UNUSED(widget)

    QString s(info.name.c_str());

    if(!movie)
    {
        movie = new QMovie;
        movie->setFileName(":/img/" + s + ".gif");
        movie->start();
    }
    if(this->hp <= 100 && cone == true)
    {
        movie->stop();
        delete movie;
```

```
        movie = new QMovie;
        movie->setFileName(":/img/Zombie.gif");
        movie->start();
        cone = false;
    }
    QImage image = movie->currentImage();
    painter->drawImage(QRectF(0, -10, 100, 100), image);

    update();
}
```

- 子弹类的功能主要是攻击僵尸，因此在它移动前判断是否有碰撞的item，检查其是否为僵尸，如果是，则直接将僵尸的生命减少，并将自己的状态设为死亡等待后续函数将其消除，植物与僵尸的碰撞也是同理。

```
void Pea::nextFrame(int currentFrame)
{
    QList<QGraphicsItem *> items = collidingItems();
    if (!items.isEmpty())
    {
       for(int i = 0; i < items.size(); i++)
       {
           //强制类型装换寻找僵尸，破坏了消息传递的结构以简化消息传递
           Zombie *zombie = qgraphicsitem_cast<Zombie *>(items[i]);
           if(zombie != NULL)
           {
               if(zombie->Type == Zombie::Type)
               {
                   zombie->attacked(atk, 0);
                   death = true;
               }
           }
       }
    }
    if ((currentFrame - frame) % speed == 0)
        move();
}
```

- 本游戏的选中功能使用鼠标实现，因此需要判断点击的位置和选中的具体内容之间的关系，这可以用一个函数的映射来实现，选中的目标内容存储在info的clickState中。
- 使用数组的形式存储所有植物和所有僵尸，由于植物和僵尸的种类各不相同且有着不同的功能，因此将植物Plant作为基类，每一种植物类型作为其派生类，而存储植物的vector类型则为基类的指针类型，这样可以通过指针指向派生类来实现类型的多态。
- 由于在Plant基类中并没有任何对于下一帧有关的操作，植物与帧数相关的操作都需要根据具体的植物类型来确定，因此将nextFrame函数声明为虚函数，同样的，返回新生成豌豆的PeaShooter的shoot函数也被声明为了虚函数。
- 游戏中僵尸的产生被设置成了按照固定的文件输入产生僵尸，当文件被读完后，无限的根据最后一次的文件读入来产生僵尸。

3.功能实现的特点

- 鼠标选中的实现方式较为简单，读取了鼠标的点击输入后，只需要根据输入坐标坐落的区间，交给对应的成员函数实现判断即可。这样的输入方式可以使得种植和铲除植物的逻辑无需通过多层的函数调用实现，只需要通过同层次的click函数根据条件进入不同的处理函数即可。

- 植物和僵尸的数据通过文件来读入，这样可以方便系统的后续添加和进行数值的修改。
- 无论是植物发射豌豆、生产阳光，僵尸移动还是攻击，所有有时间间隔的行为都被设置了bool型变量来控制其是否进行，这样设计是为了代码逻辑更加清晰。如在Game类的nextFrame函数中，会首先调用Plant、Zombie、Pea的nextFrame函数完成对其内部bool型变量的操作，也就是判断是否要移动、发射、攻击的bool型变量已经在nextFrame函数中已经完成判断，接下来只需要直接调用对应的move、attack函数完成操作即可，这样就无需要Game类中进行繁琐的条件判断了，这样使得Game类中的交互方式显得更加清晰。
- 利用qt实现了各对象自我绘制的方法，并且使用了一些qt的方法减少了Game类的交互工作，简化了代码。

4.遇到的问题和解决方法

- qt的各种功能十分强大，但是使用过程中会出现许多无法预料的问题，如在scene类中是无法使用QPainter进行绘制的，还有在Scene层重写moveMouseEvent等函数后，需要在重写的函数中调用QGraphicScene的moveMouseEvent以便将鼠标事件传至下层item。这些qt的基础学习在这个实验中占据了很多的时间。
- 在改进实验二的代码时，由于原本的结构不适合现在的输出形式，被迫更改了原本info和shop类的结构，但在回头反思代码结构时，发现这样的更改并不合理，比较合理的方式时将需要修改输出形式的部分单独拆分作为一个类，再与其他的类进行聚集。

4.实验截图


PVZ_GUI
599
50
100
150
50
WELCOME