第一章 绪论

1.1 工程背景

手游指在手机等各类手持硬件设备上运行的游戏类应用程序，其需要具备一定硬件环境和一定系统级程序作为运行基础。2004 年，手机游戏均为 WAP 游戏，到 2005 年，图形化手机游戏激增，已经超过 40 余款。2005 年 6 月，盛大英特尔宣布携手共同开发国内手机游戏市场，手机网游行业阵营开始空前壮大，继盛大、北京掌讯、美通之后，网易、空中，标派等也纷纷加入，目前国内手机游戏厂商已经近 30 家[1]。手机游戏尚处于市场导入期，在未来几年内，手机游戏将步入快速发展阶段。

1.2 课题的价值和意义

近年来，随着智能机的普及以及 4G 的覆盖率增加，手机网游日益兴起，现已经有近两千万的手机网游玩家了。2016 年，国内移动互联网竞争格局未定，运营商拒绝管道化欲谋更多话语权，转型力度、资源投入日趋加大；在部分细分领域，围绕运营商转型的业务和渠道价 值骤然放大，引发产业整合日趋频繁。在此背景下，国内移动互联龙头拓维信息通过多年技术积累及运营商渠道优势，确定以手机动漫与游戏业务为两大核心发展方向之一[2]。

全球使用的移动电话已经超过 10 亿部，而且这个数字每天都在不断增加。在除美国外的各个发达国家，手机用户都比计算机用户多。手机游戏潜在的市场比其他任何平台，比如 PlayStation，都要大。在控制台游戏时代，PlayStation 热销的一个原因就是便携性——人们可以随时随地沉浸在自己喜欢的游戏中，还可以随时随地抢购自己喜欢的装备或宠物。和游戏控制台或者 PC 相比，手机虽然可能不是一个理想的游戏设备，但毕竟人们总是随 时随身携带，这样手机游戏很可能成为人们消遣时间的首选。手机便携性、移动性的特征更能满足用户随时随地玩游戏的需求，用户利用排队、等车的时间进行游 戏，手机游戏碎片化的特性凸显。调查显示，29。8%的用户在用手机玩游戏以后电脑端玩游戏的时间减少，手机游戏已经开始抢夺电脑游戏时间。22。4%的 用户手机游戏时间越来越长，仅有 10%的用户时间变短[3]，手机游戏已逐渐成为一种普遍的娱乐方式。因为手机是网络设备，在一定限制因素下可以实现多人在线游戏。随着移动网络的发展，移动游戏也越来越多的被大家接受，对于之前长期通知市场的掌机来说造成了不少的冲击。市场研究公司 IDC 和 App Annie 报告显示 2013 年第一季度 iOS 和 Android 平台游戏业务营收是掌机的 3 倍。手机游戏市场潜力大，投入资金少，吸引了很多市场进入者，但中小 SP 在激烈的竞争中生存问题是需要考虑的主要问题。手机游戏开发商、游戏应及服务提供商不重视市场宣传和推广工作，忽视对于游戏产品，用户的体验和习惯培养重要性。手机游戏市场竞争激烈，该竞争涉及国内，也涉及国

外游戏开发商。追求低成本和短期利益，现游戏产品的质量粗糙。手机游戏的同质化也越来越严重，创新力不足。

1.3 课题重心

本课题的主要工作任务有两个，一是完成雷霆战机游戏的设计与开发，二是通过开发来对引擎尽心深入的学习。

我们知道 cocos2d-x 引擎是一个开源的游戏引擎，我们可以通过学习引擎的源代码，了解引擎的整体架构和组织形式。同时可以查看引擎是如何处理内存管理、渲染、事件处理等项目开发过程中都会遇到的问题，对于帮助自己进步有着巨大的好处。

1.4 本章小结

在本章中，介绍了手机游戏的背景，和进行移动端游戏开发的前景，同时对项目任务也做了初步的说明。

第二章 基本原理

2.1 基础架构

在 cocos2d-x 中，主要的包括 Director、Scene、Layer 和 Sprite 这几个重要的概念，它们之间的关系如下。

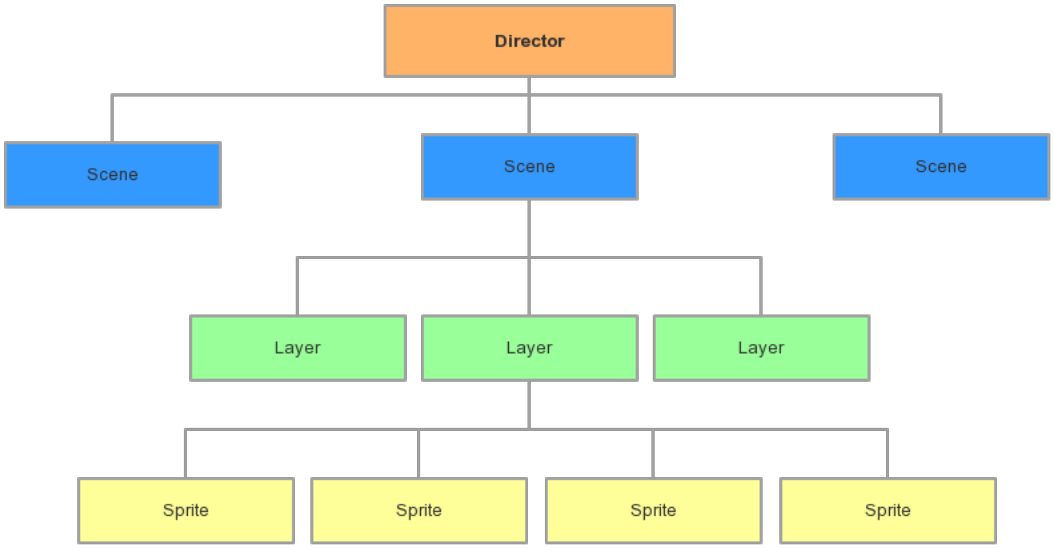

图 2-1 cocos2d-x 架构

这幅结构图非常重要，开发过程中，都是基于这个结构图来理清项目结构。

2.1.1 Director（导演）

Director 控制着整个游戏，会经常进行一些调度控制；在 cocos2d-x 3.x 中，Director 使用的是单例模式，在整个游戏运行期间，它就只有一个实例。Director 是整个 Cocos2d-x 3.x 的核心，是整个游戏的导航仪，在游戏中，一般由 Director 来完成以下操作的控制：

OpenGL ES 的初始化；

场景的转换；

游戏的暂停、继续的控制；

世界坐标和 GL 坐标之间的切换；

对节点的控制；

保存和调用保存的游戏数据；

屏幕尺寸的获取，等等。

2.1.2 Scene（场景）

Scene 场景是 Cocos2d-x 3.x 中必不可少的元素，游戏中我们需要构建不同的场景（至少一个），游戏里关卡、板块的切换就是一个个场景之间的切换。场景的一个重要的作用就是流程控制的作用，我们可以通过 Director 的一系列方法控制游戏中不同的场景的自由切换。同时场景是层的容器，包含了所有需要显示的游戏元素。通常，当我们需要完成一个场景的时候，会创建一个 Scene 的子类，并在子类中实现我们需要的功能。比如：我们可以在子类的初始化中载入游戏资源，为场景添加层，启动音乐播放等等。

在代码中，我们一般是这么做的：

auto scene = Scene::create();

通过上述代码来创建一个 Scene 场景对象，然后往这个场景中添加 Layer 层。对于一个场景，我们并不是可见的，而场景只是一个 Layer 的容器，用来存放和管理 Layer 层的。比如：

auto layer = HelloWorld::create();

scene->addChild(layer);

往场景中添加了一个 Layer 层。

Director 导演在控制着这一切，这就对了，有这么个导演，它决定着这一场用什么场景，下一场该用什么场景。在 AppDelegate::applicationDidFinishLaunching 函数中，设置了程序启动时要进行的初始化工作，也设定了程序启动时的第一个场景。

director->runWithScene(scene);

在游戏中，如果存在多个场景的情况下，就会有场景之间的切换。在 Cocos2d-x 中，我们可以设置场景切换之间的效果。

在上面总结的函数中，很多都涉及到场景的切换，比如 replaceScene 就会使用新的场景替换当前正在运行的场景。但是，如果直接进行替换，则没有场景之间切换的一些“很炫”的切换效果。如果需要在切换场景时，指定一些切换场景的动画效果，则需要了解一些切换类。

通过 replaceScene 函数替换当前正在运行的场景，我们给参数添加了一个 TransitionSlideInT 切换外包装类。这样的话，这个场景就不是直接显示了，而是在场景效果动画播完以后进入场景，起到过渡的效果。

切换效果类都是类 TransitionScene 的子类，TransitionScene 的 create 函数有两个参数，第一个参数表示切换的时间，第二个参数是要进入的场景。

2.1.3 Layer（层）

Layer 是处理玩家事件响应的 Node 子类。与场景不同，层通常包含的是直接在屏幕上呈现的内容，并且可以接受用户的输入事件，包括触摸、键盘输入等。我们需要在层中加入 Sprite（精灵）、文本标签或者其它游戏元素，并设置游戏元素的属性，比如位置、方向和大小；设置游戏元素的动作等。在我们设计好层以后，只需要把层按照顺序添加到场景中就可以显示出来了。

布景层 Layer 是 Node 类的子类，Layer 的特殊之处在于它可以响应事件。在实际项目中，Layer 也是一个容器，作为精灵、菜单、等等的容器。Scene 是 Layer 的容器，Scene 可以用来管理 Layer，而 Layer 又是其它对象的容器。Layer 就是因为它可以响应事件而变的这么厉害。它可以处理触摸、键盘和重力计事件。

在实际开发中，我们建立自己的布景层，通常会建立一个新的类，这个新的类的父类为 Layer。

2.1.4 多层布景层类 LayerMultiplex

在游戏开发中，一般会把游戏分为两部分：一部分是游戏界面部分，也就是常说的 UI 部分；另 一部分就是游戏本身部分。有时 UI 有很多页面，在页面中用的图也并不是很多，不需要使用切换场景，只需把不同页面做成不同的布景层，然后切换布景层。那么这就需要一个“管理者”来管理这些界面，这时候就要使用 LayerMultiplx 类。

在很多游戏中都需要在不同的界面中使用相同的几个变量，如果不使用 LayerMultiplx，就需要做大量的变量保存工作，那样你就会深陷变量的泥潭当中。

2.1.5 Sprite（精灵）

cocos2d-x 3.x 中的精灵和其它游戏引擎中的精灵是类似的，

精灵类 Sprite 是一张二维图片对象，可以使用一张图片或者一张图片的一块矩形部分来定义。精灵对象可以移动，旋转，缩放，执行动画，并接受其他转换。cocos2d-x 的 Sprite 由 Texture，Frame 和 Animation 组成，由 OpenES 负责渲染。

一般使用 Texture2D 加载图片，使用 Texture2D 生成对应的 SpriteFrame（精灵帧），SpriteFrame 添加到 Animation 生成动画数据，用 Animation 生成 Animate（就是最终的动画动作），最后用 Sprite 执行这个动作。

我们在游戏中，比如战场中的英雄、怪物等这些都是 Sprite，对于 Sprite 的使用，直接关系到你对游戏的编写和控制。好吧，言重了，反正是比较重要的东西。理解 Sprite 的不同创建与加载方式，对游戏的性能都有很大的关系。

精灵帧缓存类 SpriteFrameCache 用来存储精灵帧，提前缓存起来有助于提高程序的效率。SpriteFrameCache 是一个单例模式实现的，它不属于某个精灵，而是所有精灵共享使用的。

我们可以向缓存中添加精灵帧，同时，也可以手动删除指定的，或者没有使用的精灵帧。具体的函数，可以参考 SpriteFrameCache 的定义。

在编码过程中，我们会遇到各种 Cache 类，经常遇到的有 TextureCache、SpriteFrameCache 和 AnimationCache。这几个缓存类都是单例模式实现的，也就是说，它们进行的缓存并不是针对某一个 Sprite 或者 Animation 而做的，而是最整个游戏中的所有 Sprite 和 Animation 而做的。

TextureCache 时最底层也是最有效的纹理缓存，缓存的是加载到内存中的纹理资源，也就是图片资源。有的时候，我们将整张大图加载到缓存中，然后从缓存中得到 Texture2D 对象，并指定 Rect，就可以得到我们需要的小图对象了；这是最常规的用法。

SpriteFrameCache 是基于 TextureCache 上的封装。缓存的是精灵帧，是纹理指定区域的矩形块。各精灵帧都在同一纹理中，通过切换不同的框帧来显示出不同的图案。SpriteFrameCache 精灵框帧缓存。顾名思义，这里缓存的是精灵帧 SpriteFrame，它主要服务于多张碎图合并出来的纹理图片。这种纹理在一张大图中包含了多张小图，直接通过 TextureCache 引用会有诸多不便，因而衍生出来精灵框帧的处理方式，即把截取好的纹理信息保存在一个精灵框帧内，精灵通过切换不同的框帧来显示出不同的图案。跟 TextureCache 功能一样，将 SpriteFrame 缓存起来，在下次使用的时候直接去取。不过跟 TextureCache 不同的是，如果内存池中不存在要查找的图片，它会提示找不到，而不会去本地加载图片。

2.2 调度器

在游戏中，经常需要隔一段时间来更新一些数据或者是人物位置，比如战斗系统，我们需要每一帧的时候，检查战斗场景中每个英雄的血量，位置等等信息。而 cocos2d-x 调度器为游戏提供定时事件和定时调用服务。这样的话，就为我们客户端程序员省去了不少的麻烦。调度器就是指定时间间隔调用指定的函数，去完成特定的功能。

2.3 事件处理机制

一个事件由触发到完成响应，主要由以下三部分组成：

事件分发器 EventDispatcher；

事件类型 EventTouch，EventKeyboard 等；

事件监听器 EventListenerTouch，EventListenerKeyboard 等。

在 cocos2d-x 3.x 中，关于事件的东西，无非就是围绕上述的三个部分展开来的，掌握了上述的三个部分，也就掌握了 cocos2d-x 3.x 中事件处理的精髓。

事件分发器：事件分发器，就相当于是所有事件的“总长官”；它负责调度和管理所有的事件监听器；当有事件发生时，它负责调度对应的事件；一般调用 Director 类中的

getEventDispatcher 获得一个事件调度器，在游戏启动时，就会创建一个默认的 EventDispatcher 对象；

事件类型：在 cocos2d-x 中定义了以下几种事件类型：

```
TOUCH,              // 触摸事件
KEYBOARD,           // 键盘事件
ACCELERATION,       // 加速器事件
MOUSE,              // 鼠标事件
FOCUS,              // 焦点事件
CUSTOM              // 自定义事件
```

事件监听器：事件监听器实现了各种事件触发后对应的逻辑；由事件分发器调用对应的事件监听器。在 Cocos2d-x 中定义以下的几种事件监听器：

```
UNKNOWN,                // 未知的事件监听器
TOUCH_ONE_BY_ONE,       // 单点触摸事件监听器
TOUCH_ALL_AT_ONCE,      // 多点触摸事件监听器
KEYBOARD,               // 键盘事件监听器
MOUSE,                  // 鼠标事件监听器
ACCELERATION,           // 加速器事件监听器
FOCUS,                  // 焦点事件监听器
CUSTOM                  // 自定义事件监听器
```

在处理触摸事件时，有以下两种情况：

对于单点触摸，需要重写这四个方法：

onTouchBegan；

onTouchMoved；

onTouchEnded；

onTouchCancelled.

对于多点触摸，需要重写这四个方法：

onTouchesBegan；

onTouchesMoved；

onTouchesEnded；

onTouchesCancelled.

eventDispatcher 是 Node 的属性，通过它管理当前节点（场景、层、精灵等）的所有事件的分发。但它本身是一个单例模式值的引用，在 Node 的构造函数中，通过以下代码获取：

```
Director::getInstance()->getEventDispatcher();
```

有了这个属性，就能方便的处理事件分发了。

注意：当再次使用 listener1 的时候，需要使用 clone()方法创建一个新的克隆，因为在使用 addEventListenerWithSceneGraphPriority 或者使用另外提供的 addEventListenerWithFixedPriority 方法时，会对当前使用的事件监听器添加一个已注册的标记，这使得它不能够被注册多次。另外，有一点非常重要，FixedPriority listener 添加完之后需要手动 remove，而 SceneGraphPriority listener 是跟 Node 绑定的，在 Node 的析构函数中会被移除。

我们可以通过以下方法移除一个已经被添加了的监听器。

EventDispatcher->removeEventListener(listener);

也可以使用如下方法，移除当前事件分发器中所有监听器。

EventDispatcher->removeAllEventListeners();

当使用 removeAll 的时候，此节点的所有的监听将被移除，推荐使用指定删除的方式。removeAll 之后菜单也不能响应。因为它也需要接受触摸事件。

2.4 内存管理

到现在，内存已经非常便宜，但是也不是可以无限大的让你去使用，特别是在移动端，内存有限，且需要内存的 APP 比较多，如果没有正确处理内存的占用，占的内存过度，系统直接撤销掉你的 APP，因此，内存管理对于游戏开发来说是很重要的。内存管理无论是语言层面，还是类库层面，都有严格的标准和实施，对于 cocos2d-x 来说，也是如此。

对于探究内存管理这种比较抽象的东西，最简单的方法就是通过代码来研究，首先通过创建一个简单的场景来看看 cocos2d-x 在完成创建一个对象的时候，它都干了些什么。

创建一个 Scene：

auto scene = Scene::create();

函数 create 是一个静态函数，

现在就涉及到了 cocos2d-x 的内存管理相关的知识了。在 cocos2d-x 中，关于对象的创建与初始化都是使用的 new 和 init 函数搭配的方式，这种方式叫做二段式创建，由于 C++中，构造函数没有返回值，无法通过构造函数确定初始化的成功与失败，所以在 cocos2d-x 中就大行其道的使用了这种二段式创建的方式，以后在自己的项目中，也可以加以采用。

由于这种方式在 cocos2d-x 中经常被使用，所以引擎内部提供了一个宏：CREATE_FUNC。如果想让我们的类也使用这种二段式创建的方式，只需要在我们的类中加入以下代码：

CREATE_FUNC(classname);

同时，需要定义一个 init 函数。我们来看看这个宏：

#define CREATE_FUNC(__TYPE__) \

```
static __TYPE__* create() \
{ \
    __TYPE__ *pRet = new __TYPE__(); \
    if (pRet && pRet->init()) \
    { \
        pRet->autorelease(); \
        return pRet; \
    } \
    else \
    { \
        delete pRet; \
        pRet = NULL; \
        return NULL; \
    } \
}
```

其中有一行代码是 ret->autorelease()，这就是 cocos2d-x 的内存管理的一部分了[4]。

在 cocos2d-x 中，关于内存管理的类有：Ref 类；AutoreleasePool 类；PoolManager 类。

Ref 类几乎是 cocos2d-x 中所有类的父类，它是 cocos2d-x 中内存管理的最重要的一环；上面说的 autorelease 函数就 Ref 类的成员函数，cocos2d-x 中所有继承自 Ref 的类，都可以使用 Cocos2d-x 的内存管理。

AutoreleasePool 类用来管理自动释放对象。

PoolManager 用来管理所有的 AutoreleasePool，这个类是使用单例模式实现的。

下面就通过对上述三个类的源码进行分析，看看 Cocos2d-x 到底是如何进行内存管理的。

2.4.1 Ref 类

先来看看 Ref 类的定义，其中实现了一个 release 函数。对于 release 函数的实现，这里需要特别总结一下：对于 new 和 autorelease 需要匹配使用，retain 和 release 也需 要匹配使用，否则就会出现断言错误，或者内存泄露；在非 Debug 模式下，就可能直接闪退了。这就是为什么我们在使用 create 函数的时候，new 成功 以后，就顺便调用了 autorelease，将该对象放入到自动释放池中；而当我们再次想获取该对象并使用该对象的时候，需要使用 retain 再次获得该 对象的所有权，当然了，在使用完成以后，你应该记得调用 release 去手动完成释放工作，这是你的任务。

例如：

当使用 create 函数创建对象以后，obj 没有所有权，当再次调用 release 时，就会出现错误的对象释放。

2.4.2 AutoreleasePool 类

AutoreleasePool 类是 Ref 类的友元类。对于 AutoreleasePool 类来说，它的实现很简单，就是将简单的将对象保存在一个 std::vector 中，在释放这 AutoreleasePool 的时候，对保存在 std::vector 中的对象依次调用对应的 release 函数，从而完成对象的自动释放。

当我们在阅读 AutoreleasePool 的源码的时候，在它的构造函数中，你会发现在这里我们把 AutoreleasePool 对象又放到了 PoolManager 里了；PoolManager 类就是用来管理所有的 AutoreleasePool 的类，也是使用的单例模式来实现的。该 PoolManger 有一个存放 AutoreleasePool 对象指针的 stack，该 stack 是由 std::vector 实现的。需要注意的是，cocos2d-x 的单例类都不是线程安全的，跟内存管理紧密相关的 PoolManager 类也不例外，因此在多线程中使用 cocos2d-x 的接口需要特别注意内存管理的问题[4]。

我们 create 一个对象以后，放到了 AutoreleasePool 中去了，最终，在调用 AutoreleasePool 的 clear 函数的时候，会对 AutoreleasePool 管理的所有对象依次调用 release 操作。而调用 clear 的方式是这段在导演类中的代码。

在图像渲染的主循环中，如果当前的图形对象是在当前帧，则调用显示函数，并调用 AutoreleasePool::clear()减少这些对象的引用计数。mainLoop 是每一帧都会自动调用的，所以下一帧时这些对象都被当前的 AutoreleasePool 对象 release 了一次。这也是 AutoreleasePool “自动”的来由。

2.5 本章小结

在这一章中，通过对 cocos2d-x 的源码的研究，进本上实现了设计任务中，对引擎学习的任务。当然这个学习过程是初步的，十分粗略，需要在以后的学习过程中进一步深化。

第三章 系统设计

这是一款飞行射击类游戏，整体环境主要还是围绕太空为主，高保真的音效，为玩家呈现一场不一样射击体验。简单的触屏操作，触屏按住随意一个地方，左右移动，便可自动攻击敌人，上下移动亦可躲避强敌。玩家在游戏中要做的就是驾驶着最新战机，在敌机身前发动攻击。在击毁敌机的同时获得分数，击毁的敌机越多，则相对的获得分数就越高。玩家进行游戏的时候需要注意不能被敌机及敌机子弹碰到，否则玩家控制角色死亡，同时游戏结束。记录玩家获取的积分。

游戏在 Mac OSX 系统下开发，基于 Xcode 的开发平台，采用了 cocos2d-x 引擎进行开发。开发环境信息如下:

System Version: OS X 10.10.5 (14F2109)

Kernel Version: Darwin 14.5.0

Development Language:C++11

Development Instrument: cocos2d-x-3.12， Xcode Version 7.2 (7C68)

Xcode 是运行在操作系统 Mac OS X 上的集成开发工具（IDE），由苹果公司开发。Xcode 是开发 OS X 和 iOS 应用程序的最快捷的方式，Xcode 具有统一的用户界面设计，编码，测试，调试都在一个简单的窗口内完成。

游戏的设计主要分为三个场景，游戏开始场景，游戏主场景，游戏结束场景，以下是其基本介绍。

3.1 欢迎场景的设计与实现

欢迎场景是进入游戏之后玩家看到的第一个场景，其采用单层设计，即一个场景中只有一个层，层创建采用经典的两段式，并将层的实例化代码写入创建场景的静态方法当中。

欢迎场景实现的功能如下所示:

1．加载资源。

2．渲染游戏 Logo，静态背景。

3．渲染开始开始按钮并添加场景切换事件。

4．渲染作者信息。

5．渲染装饰用的飞机精灵，并根据帧数调用射击方法。

其中加载资源在资源管理部分有详细的介绍，在这里不再赘述。而渲染游戏 Logo，背景，信息等比较简单，在这里也不做过多描述。下面着重介绍一下开始事件的设计与飞机精灵的添加。

3.1.1 开始按钮的设计

开始按钮的监听为当按钮被按下，则创建一个延时事件，和一个场景切换事件，在延时时间的同时，创建一个闪光特效。

闪光特效的实现在特效一节介绍，这里不做进一步描述，延时事件保证了闪光特效完全播放完毕后才进入场景切换。两个事件由当前层顺序执行。

3.1.2 飞机精灵的添加

飞机精灵是模型中玩家战机的实例化，但是在本层中并不添加物理刚体，仅做装饰用。在本层的 init 方法中进行实例化，并添加无限循环的左右移动动作。在场景的更新函数中，每十帧调用一次射击方法，用于创建自动销毁的子弹精灵，这个过程在模型设计的部分进行详细的介绍。

3.2 游戏结束场景的设计与实现

游戏结束场景是所有的场景中最简单的一个，其需要实现的功能有

1. 显示分数以及游戏结束的标识
2. 提供重新开始的按钮

场景层采用两段式实例化，并在 init 方法中传递分数。实现一个场景实例化的静态方法，并在此方法中尽心层创建并将层加入场景。

3.2.1 传递分数

我们给结束层添加一个分数属性，在层创建的时候将属性进行初始化，并将分数标签的渲染代码放入 onEnter 方法中，因为如果放入 init 方法中，标签的渲染就会在层实例化之前就已经完毕，而此时的标签值为初始化之前的值，为不确定的。放入 onEnter 方法可以保证标签可以在实例化完成之后并全部进入场景后进行渲染，保证了分数的正确性。

3.2.2 添加重新开始按钮

重新开始按钮的监听为当按钮被按下，则创建一个延时事件，和一个场景切换事件，在延时时间的同时，创建一个闪光特效。延时事件保证了闪光特效完全播放完毕后才进入场景切换。两个事件由当前层顺序执行。

3.3 游戏主场景的设计与实现

游戏主场景是玩家进行游戏的主要场景，是整个游戏设计中最复杂最核心的部分，因为这个部分较为复杂，并且涉及到多种处理逻辑和渲染逻辑，因此我们不能将这个场景简单的使用原先的单层设计，而是要进行游戏逻辑的模块化处理，和渲染逻辑的分层化处理。

3.3.1 简单介绍

对于游戏逻辑的模块化，我们将其分为:

1. 触摸逻辑模块，主要负责实现与用户的触摸事件的响应操作。
2. 碰撞逻辑模块，主要负责实现整个游戏碰撞逻辑响应操作。
3. 精灵模型模块，实现游戏中所使用的 4 中游戏精灵模型。
4. 光效模块，负责设计与实现游戏中使用的光效。
5. 资源管理模块，负责管理游戏中的游戏资源，比如贴图，音效等。
6. 轨迹创建模块，负责创建游戏精灵对象的移动轨迹。

而对于游戏分层的设计，我们可以将其分为:

1. 背景层，这一层实现一个飞机向前飞行的效果，主要是背景图的滚动处理，这一层没有任何的事件的监听。

2. 物理层，这一层是游戏的主要层，本层的创建使用了物理引擎来监听碰撞事件，同时这一层接受触摸事件的监听，并根据监听做出相应，本层的精灵全部创建物理刚体，并设置碰撞标识符。

3. 标签层，这一层比较简单，主要是渲染与游戏有关的两个信息，玩家生命值，与玩家已获得的分数。同样不监听事件，将触摸事件向物理层传递。

3.3.2 背景层设计

背景层的设计分为

1. 背景的自动滚动;
2. 添加堡垒精灵元素，执行移动效果并自动移除;

背景的自动滚动

在我们的飞机游戏中，为了实现飞机飞行的效果，要求我们的背景层可以向下滚动，已达到玩家认为飞机正在飞行的错觉。我们采用两张背景图精灵元素交替执行下移的方法来实现背景层的滚动效果，具体的方法是创建两个上下并列的背景图， 第一张在上，第二张在下，也就是说，当玩家看到第一张图片的时候， 第二张图片恰好处于第一张图片看不到的地方，然后执行第一张图片下移，第二张图片跟随下移， 直至第一张图片完全移出屏幕，即第二张图片完全移入屏幕，

此时将第一张图片的位置重设到第二张图片的上部，并重新执行这个过程。通过这样的方式，在视觉上，玩家无法看到屏幕外的情况，因此可以产生背景图滚动的效果。

以下是实现此效果的伪代码:

```
firstImage.Pos.Y--;
if firstImage.Pos OutOf visibleSize:
    firstImage.Pos = Up to secondImage.Pos;
secondImage.Pos.Y--;
if secondImage.Pos OutOf visibleSize:
    secondImage.Pos = Up to firstImage.Pos;
```

堡垒元素的添加

为了体现雷霆战机游戏的科技感，游戏设计了不同的堡垒，作为背景的添加元素。堡垒的纹理图片一共有四种，如下所示

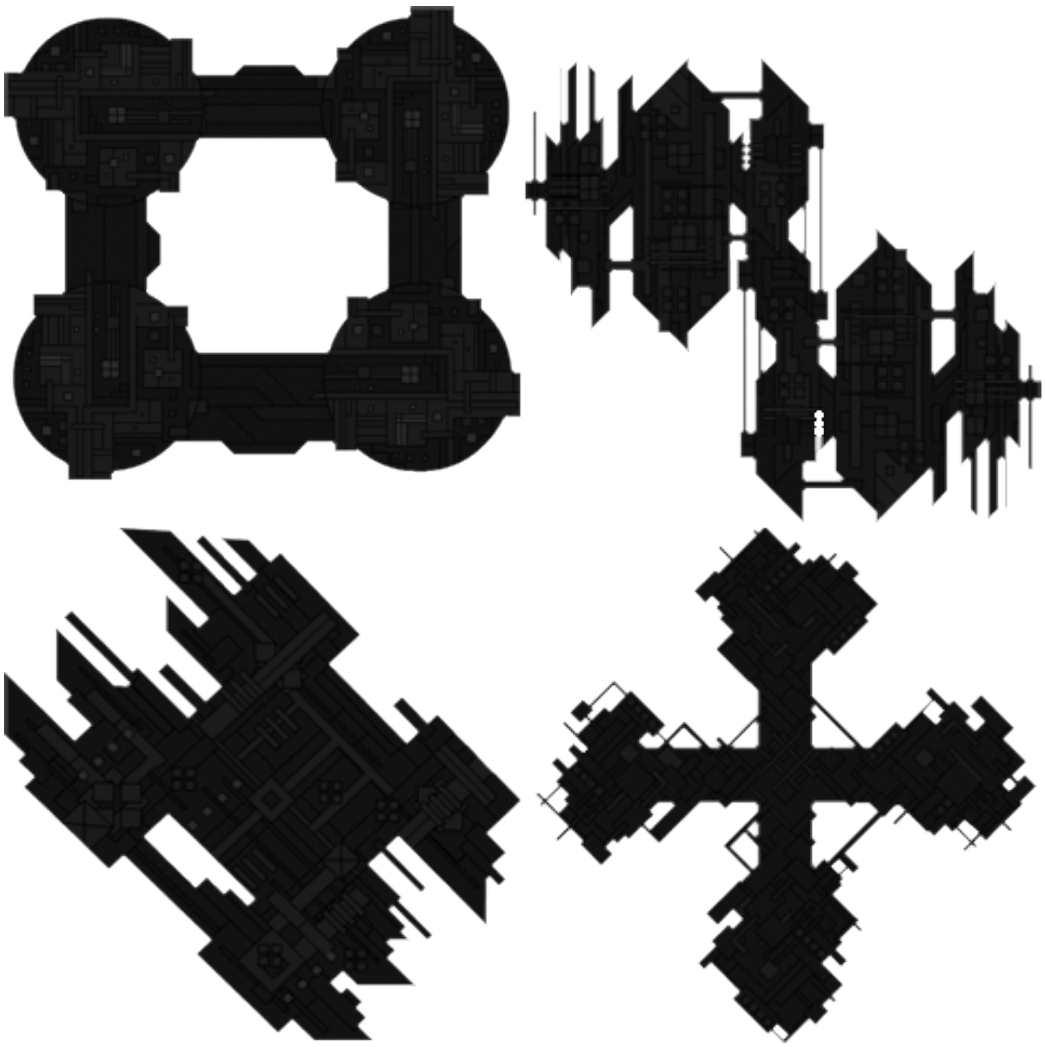

图 3-1 堡垒素材展示

堡垒精灵加入从上到下的移动动作，从屏幕外的上部开始，到从下部完全移出屏幕结束，然后将此精灵从本层删除。精灵下落为垂直下落，起始的横坐标为随机。背景层设置一个堡垒的标志位，取值为 0 – 3，表示不同的堡垒精灵，在添加的过程中，每添加一次，将标志位加一，到达上限后，归零。添加堡垒元素的时机为，两个背景图进行坐标重设的时候。堡垒精灵添加之后，需要在执行完毕动作后进行自动移除， 否则会造成精灵元素在下部不断积累，以至于大量的内存资源不断被浪费。因此，我们需要自动检测完成动作的堡垒元素，并将其从本层移除。

添加堡垒后， 伪代码被更新为

```
firstImage.Pos.Y--;
if firstImage.Pos OutOf visibleSize:
   firstImage.Pos = Up to secondImage.Pos;
   Add BuildingWithMoveAction(BuildingFlag, RandomPos);
secondImage.Pos.Y--;
if secondImage.Pos OutOf visibleSize:
   secondImage.Pos = Up to firstImage.Pos;
   Add BuildingWithMoveAction(BuildingFlag, RandomPos);

CheckAndRemove(All_Buildings);
```

使用的数据结构是四个创建好的精灵元素。 在当前背景层实例化时创建。

背景层的方法重写

我们的背景层继承自 cocos2d-x 引擎的 Layer 元素， 我们重写其 init 方法与 update 方法。init 方法是背景层的初始化方法，在背景层初次创建时会被调用，我们将背景层所需要的各种数据结构的实例化过程放在此方法中。update 方法是背景层的更新方法，这个方法用于在每一帧重新渲染精灵并作出一些修改，调用的次数与当前的帧率相关。我们将以上的伪代码的实现放在这个方法里。

背景层的内存管理方式

背景层采取两段式创建，将类的实例化与初始化分离，采用工厂模式，并在初始化之后调用 Ref 类的 autorelease 方法， 将层对象自动托管到全局的内存池。

3.3.3 模型设计

雷霆战机游戏中的模型，指的是游戏当中所使用的四种对象，即玩家战机，敌方战机，玩家子弹，地方子弹。

四种对象均继承自 cocos2d-x 引擎的 Sprite 类，并重写其中的一些方法。在进行我们的模型设计之前，为了更好的说明我们的设计方法，必须对 cocos2d-x 引擎的两段式对象实例化方法有所了解。

两段式创建

所谓的两段式，是指将对象的实例化与初始化分离，将类的构造方法设为私有不允许直接进行实例化，创建一个静态的工厂方法进行对象的实例化，因为 cocos2d-x 引擎的内存管理机制，对象的获取与管理全部使用指针进行，因此在实例化时要在堆区进行内存分配，并自动将此内存的管理权托管于 cocos2d-x 的内存池，若创建不成功则需要进行安全删除并返回空指针，初始化代码全部写在类的 init 方法中，此方法在类的工厂方法中进行调用。

采用这样的创建方式，避免了对象因初始化失败，或者渲染的问题而导致构造函数抛出异常而内存无法回收，造成内存泄露。

在我们的模型中，因为同一个类的所有对象都具有相同的纹理和动作，因此我们采用直接将固定的模式写到精灵的初始化方法中，而不需要每次进行创建之后才进行手动添加。

敌机子弹与玩家子弹设计

在初始化方法中，我们首先进行图片的添加，并启用颜色混合，这样可以使得图片与背景更好的进行混合。

并且要添加移动方法:

1. 玩家子弹从玩家战机的位置开始垂直向上移动，并在移出屏幕外后将自己移除。
2. 敌机子弹从地方战机的位置开始向下垂直移动，并在移出屏幕外后将自己移除。

同时我们需要在这里对两种子弹的物理刚体进行创建，玩家子弹是矩形，创建为矩形刚体，而敌机子弹为圆形，创建为圆形刚体。并且进行碰撞标识的分配，分配的方式以及使用的方法在碰撞检测设计中有详细的介绍，这里不再重复。

敌方战机的设计

图 3-2 敌机素材展示

这里共有 6 中敌方战机模型，虽然其图片是不同的，但是都拥有相同的属性和动作，因此我们将它们归类到同一对象。我们选择在其创建的时候使用一个 flag 参数来决定创建哪一个敌机，不同的敌机具有不同的图片和 HP 值。

在模型部分，我们不创建敌机的飞行轨迹，因为为了避免单调，我们选择将不同的飞行轨迹与不同类型的敌机进行组合。因此，设计了单独一个模块进行飞行轨迹的创建，在这里也不进行敌机模型的位置设置，一切有关于位置的信息都由轨迹模块进行操作。

除了飞机模型的创建之外，我们需要将部分游戏逻辑在模型中实现，一共要实现以下三个方法:

1. 飞机是否生存，如果被摧毁，则在碰撞处理中会调用相应的过程进行处理，这里不详细描述。

2. 飞机掉血，当碰撞检测接受到飞机的碰撞事件时，会首先调用此方法，然后进行判活。

3. 射击方法，所谓的射击方法是比较形象的说法，实际上是向当前层添加一个子弹对象，子弹对象的创建在上文已经描述。

玩家战机的创建

图 3-3 玩家素材展示

我们可以看到，虽然玩家战机只有一种形式，但是是由三张相关的帧组成，也就是说，玩家战机精灵实际上是一个三帧的动画，并使用无限重复的方式播放。

玩家战机的轨迹是由玩家控制，在有关触摸逻辑的部分进行介绍，玩家战机在游戏层创建时会直接初始化。战机同样要实现以下四个方法:

1. 飞机是否生存，如果被摧毁，则在碰撞处理中会调用相应的过程进行处理，玩家战机的摧毁意味着游戏结束。

2. 飞机掉血，当碰撞检测接受到飞机的碰撞事件时，会首先调用此方法，然后进行判活。

3. 射击方法，向当前层添加一个子弹对象。

4. 获取当前生命值。为了在游戏层中可以让玩家直观的感受到剩余生命值，需要设置一个访问器。

3.3.4 飞行轨迹设计

飞行轨迹指的是敌方战机的运动路径，在游戏中，我们不仅仅只设计从上到下固定的游戏轨迹，丰富的地方战机移动路径可以大大丰富游戏的趣味性。为了实现敌方战机与路径的分别控制，游戏将路径的实现与敌机创建进行分离，使用一个单独的模块对轨迹进行管理。

轨迹模块共实现了六种轨迹，分别是:

1. 从屏幕的右上角开始，进入场景后进行横向移动，在移动的图中发射子弹。然后迅速俯冲向屏幕底部中点。以及其镜像轨迹，轨迹展示如下:

2. 从屏幕的右上角开始，俯冲向屏幕底部，然后回退到上部进行横向移动，移动的过程中发射子弹，最后垂直俯冲。以及其镜像轨迹，轨迹展示如下:

3. 从屏幕顶端中点开始移动，垂直移动到中线上部，然后开始向左方横移，移动到屏幕边缘后反向右方横移，到达右屏幕边缘后再次反向向中线移动，到达中线后俯冲。在这个横向移动过程中，不断发射子弹。轨迹展示如下:

4. 从屏幕底端右部开始，迅速移动到屏幕左上角，然后横移到屏幕右上角，然后俯冲到屏幕左下角，横移过程中不断发射子弹。这一个轨迹与普通轨迹不同，不是传统的从上部开始，移动范围广，速度极快，玩家往往反映不够及时，无法马上躲避，是一个难度很大的轨迹。轨迹展示如下:

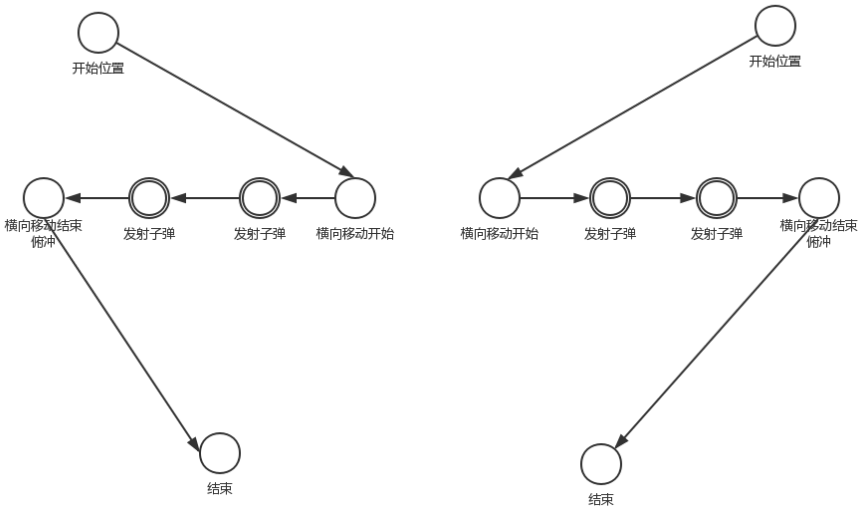

图 3-4 轨迹 1、2

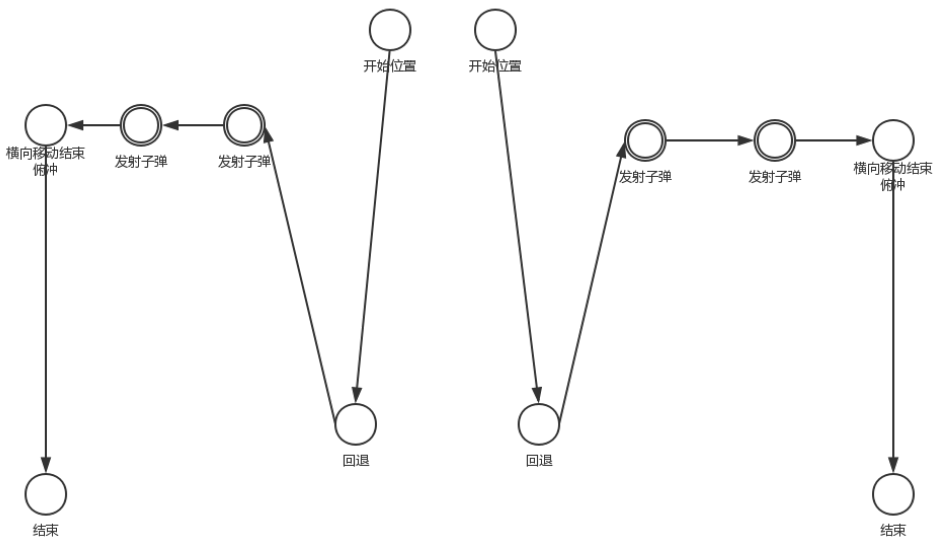

图 3-5 轨迹 3、4

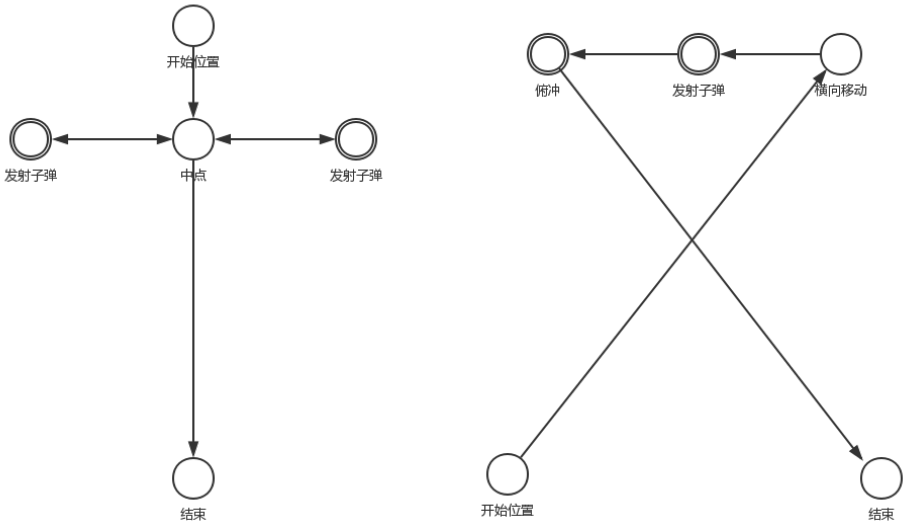

图 3-6 估计 5、6

6 种轨迹每一个都是由工厂方法进行创建，返回一个轨迹对象。但是调用六种方法的设计并不灵活， C++也没有函数反射的设计，因此我使用一个总的工厂方法，利用一个 Flag 来控制调用哪一个。同时，传入一个敌机对象指针，使得可以调用对象的射击方法并设置对象的位置。伪代码实现如下:

```
assert((flag <= 5 && flag >= 0));
switch (flag)
{
   case 0 :
      return CreateA();
   case 1 :
      return CreateB();
   case 2 :
      return CreateC();
   case 3 :
      return CreateD();
   case 4 :
      return CreateE();
   case 5 :
      return CreateF();
}
```

3.3.5 碰撞检测逻辑模块

碰撞检测的简介

碰撞侦测(Collision detection)或称为碰撞检测通常是指一种判断两个或多个对象是否产生交集的的方法，往往应用于电子游戏和其他计算物理学当中，也应用于人工智能当中。除了确定两个对象是否已经碰撞， 碰撞侦测也可以用于计算冲击的时间(TOI)，以及回报对象交叉的位置，碰撞响应一旦侦测到碰撞则处理模拟(物理引擎，布娃娃系统)，解决碰撞侦测问题需要使用广泛的概念，如线性代数和计算几何[3]。

本游戏的碰撞检测

作为一个 2D 游戏， 并且在不涉及到复杂的物理模型情况下，（比如计算摩擦力，弹性系数，力矩等复杂物理参数）我们采用基本的 AABB 盒的检测方式。也叫包围体的检测方式。在碰撞检测中，如果两个包围体没有相交，那么所包含的物体也就不会碰撞。反之，如果两个元素的包围体发生了相交，我们就可以说两个物体发生了碰撞。

AABB 盒检测的好处

由于包围体的几何形状较为简单，而物体通常是多边形或者简化为多边形近似的数据结构所组成，所以对于包围体的检验通常要比对于物体本身的检验速度更快。在其中任一种场合下，如果物体不可见的话，那么根据视体对每个多边形的检验都是无用的计算。不管物体表面是否真的可见，屏幕上的物体必须裁剪到屏幕能够显示的区域[3]。

碰撞逻辑的概要设计

在设计的雷霆战机游戏中，需要进行三个方面的逻辑设计

1. 碰撞过滤:碰撞是否发生，即如何检测有用的碰撞，忽略掉不必要的碰撞，称为碰撞过滤。

2. 碰撞归类:发生碰撞的过程有多个，不同的碰撞过程我们要采取不同的处理策略，这一步要识别碰撞的过程属于哪一类，称为碰撞归类。

3. 碰撞处理:根据碰撞归类的结果，需要采取不同的游戏逻辑来进行处理，部分还需要调用场景切换。称为碰撞处理，

碰撞过滤设计

需要处理的碰撞一共有三种

1. 敌机与玩家战机发生碰撞。

2. 敌机子弹与玩家战机发生碰撞。

3. 玩家子弹与敌机发生碰撞。

除此之外的碰撞事件，比如敌机与敌机碰撞，敌机子弹与敌机发生碰撞等，我们默认碰撞没有发生，即是默认这些元素可以发生重叠。

当场景中同时发生了多个碰撞事件，则按照 1，2，3 的不同的优先级进行顺序处理，先处理序号小的比较重要的事件，然后在处理序号大的次重要事件，但是某些游戏逻辑可能对于场景中的游戏元素做了移除操作，因此必须进行碰撞对象的判空操作，以防碰撞事件的对象已经在上次碰撞处理中被移除，造成野指针的异常。

对于检测到的碰撞，将会获取碰撞中被检测到的碰撞对象，并将其传递到下一步碰撞分类当中。

碰撞分类设计

根据模型设计，碰撞处理一共涉及到四种对象:敌机，敌机子弹，玩家子弹，玩家战机。

为了便于碰撞检测的归类，我将不同的对象类分配一个唯一的标识符，标识符使用一个四位的二进制串来表示:

分配情况如下:

1. 敌机 0b0001
2. 敌机子弹 0b0010
3. 玩家子弹 0b0100
4. 玩家战机 0b1000

这样分配的好处之一是，进行碰撞归类时，只需要将获取到的两个碰撞对象的标识符执行或运算，就可以了解是哪一种碰撞类型。根据碰撞过滤的设计，一共需要处理的碰撞事件有:

1. 敌机与玩家战机发生碰撞，执行或运算后的运算结果为 0b0101。
2. 敌机子弹与玩家战机发生碰撞，执行或运算后的运算结果为 0b0110。
3. 玩家子弹与敌机发生碰撞，执行或运算后的运算结构为 0b1001。

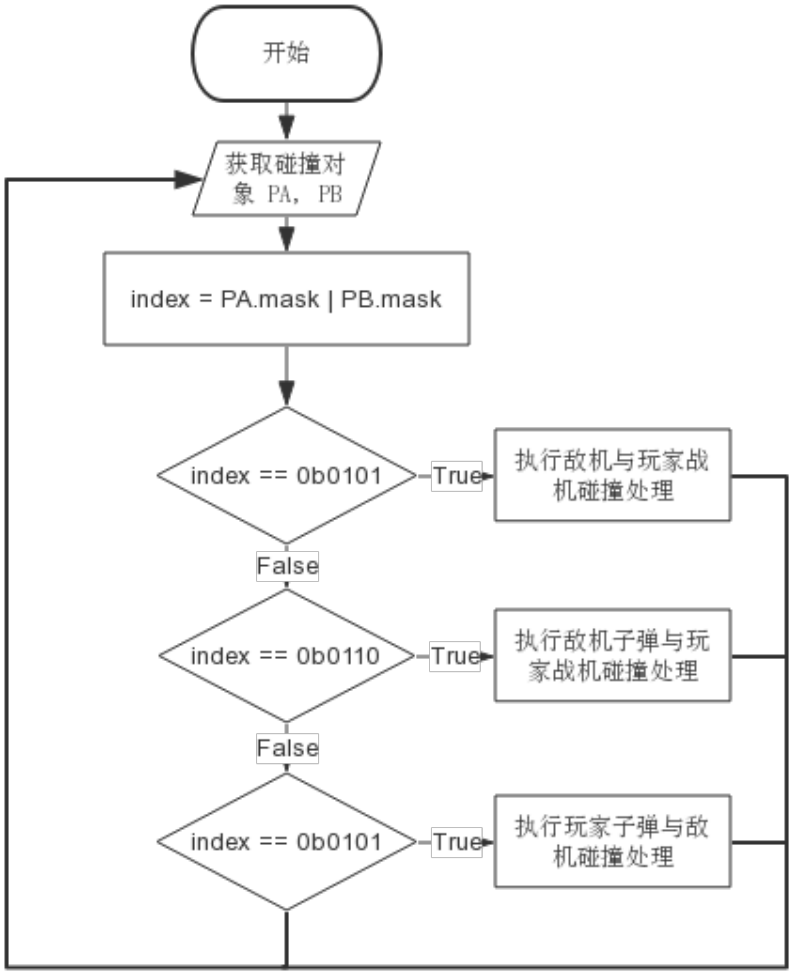

图 3-7 分类流程图

分配的好处之二是，碰撞检测机制仅仅只能获取碰撞的对象，也就是说获取的对象指针是四种模型的基类指针，但是我们在后续的游戏处理逻辑中，需要对不同的对象执行不同的操作，也就是说我们需要进行多态的处理，这样可能会导致 C++ 运行时崩溃，比如在玩家子弹与敌机碰撞的事件中，如果将敌机指针指向了子弹对象，游戏逻辑中需要调用敌机的 Alive 方法来判定敌机是否存活，但是子弹并没有实现这一方法，就会导致崩溃。

有了明确的类标识符，我们就可以通过不同的标识符来进行安全的转型。伪代码实现如下:

```
TypeA * a = ((ATag == 0b0100) ? (TypeA *)nodeA : (TypeA *)nodeB);
TypeB * b = ((BTag == 0b0001) ? (TypeB *)nodeA : (TypeB *)nodeB);
```

在以上的过程中，我们判定了不同的碰撞事件，并使用标识符精确转型了对象指针，有了这些条件，就可以进行下一步的逻辑处理过程。

碰撞处理部分

碰撞处理是指在完成碰撞的判定，碰撞的分类后，我们应当对不同的碰撞事件作出具体的响应，这也是整个碰撞逻辑处理最核心也是最紧要的部分，影响到整个游戏的实现效果。

(1) 敌机与玩家战机碰撞逻辑设计

当发生碰撞时

1. 判定敌机被撞毁，也就是说，停止敌机的当前行动，并从当前的层将敌机精灵删除。
2. 调用特效模块创建一个爆炸特效。
3. 执行玩家战机的 Hurt 方法。
4. 玩家战机存活判定，若存活则逻辑结束。
5. 玩家战机被摧毁，停止战机当前行动，停止事件监听，将战机从本机移除。
6. 调用特效模块创建爆炸特效。
7. 创建一个延时动画等待爆炸特效完成。
8. 创建场景切换，传递当前分数。
9. 场景切换。

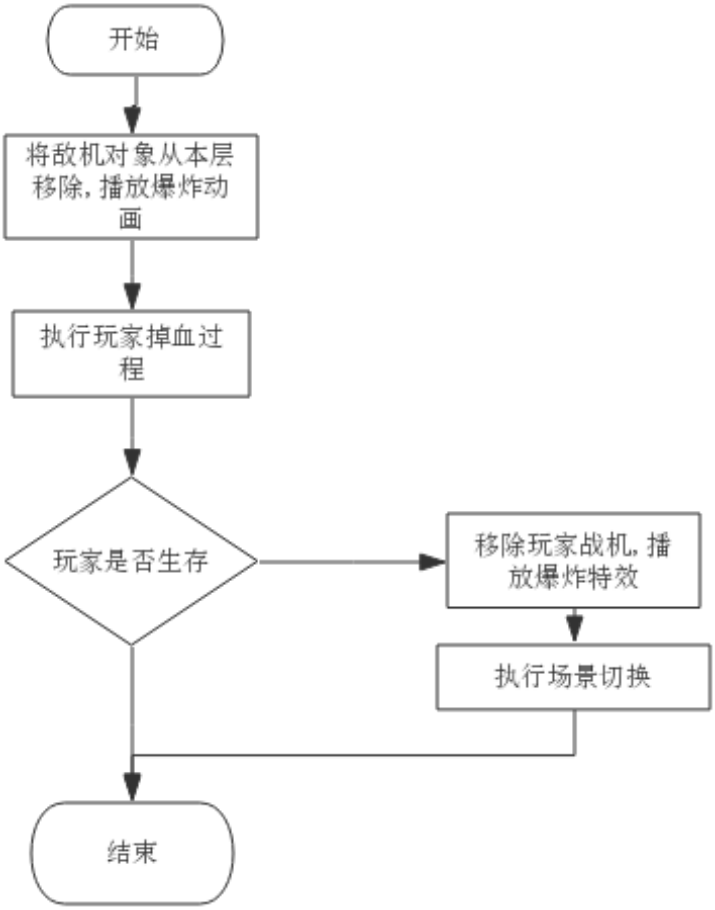

图 3-8 玩家敌机碰撞

(2) 敌机子弹与玩家战机碰撞逻辑设计

1. 停止敌机子弹的当前行动，并从当前的层将敌机子弹精灵删除。
2. 调用特效模块创建一个打击特效。
3. 执行玩家战机的 Hurt 方法。
4. 玩家战机存活判定，若存活则逻辑结束。
5. 玩家战机被摧毁，停止战机当前行动，停止事件监听，将战机从本机移除。
6. 调用特效模块创建爆炸特效。
7. 创建一个延时动画等待爆炸特效完成。
8. 创建场景切换，传递当前分数。
9. 场景切换。

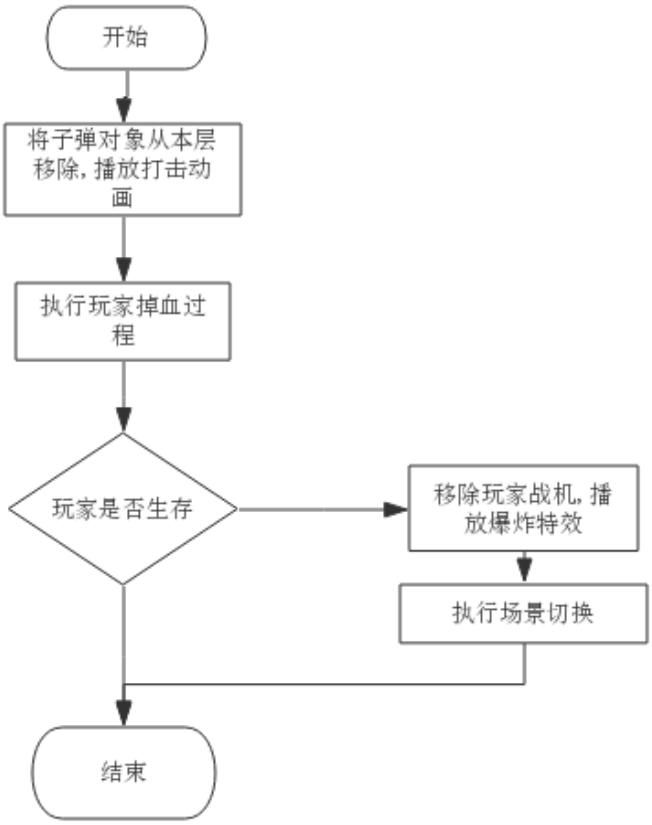

图 3-9 玩家子弹碰撞

(3) 敌机与玩家战机子弹碰撞逻辑设计

1. 停止子弹的当前行动，并从当前的层将敌机精灵删除。
2. 调用特效模块创建一个打击特效。
3. 执行敌机的 Hurt 方法。
4. 敌机存活判定，若存活则逻辑结束。
5. 敌机被摧毁，停止战机当前行动，停止事件监听，将战机从本机移除。
6. 调用特效模块创建爆炸特效。
7. 创建一个延时动画等待爆炸特效完成。
8. 增加分数

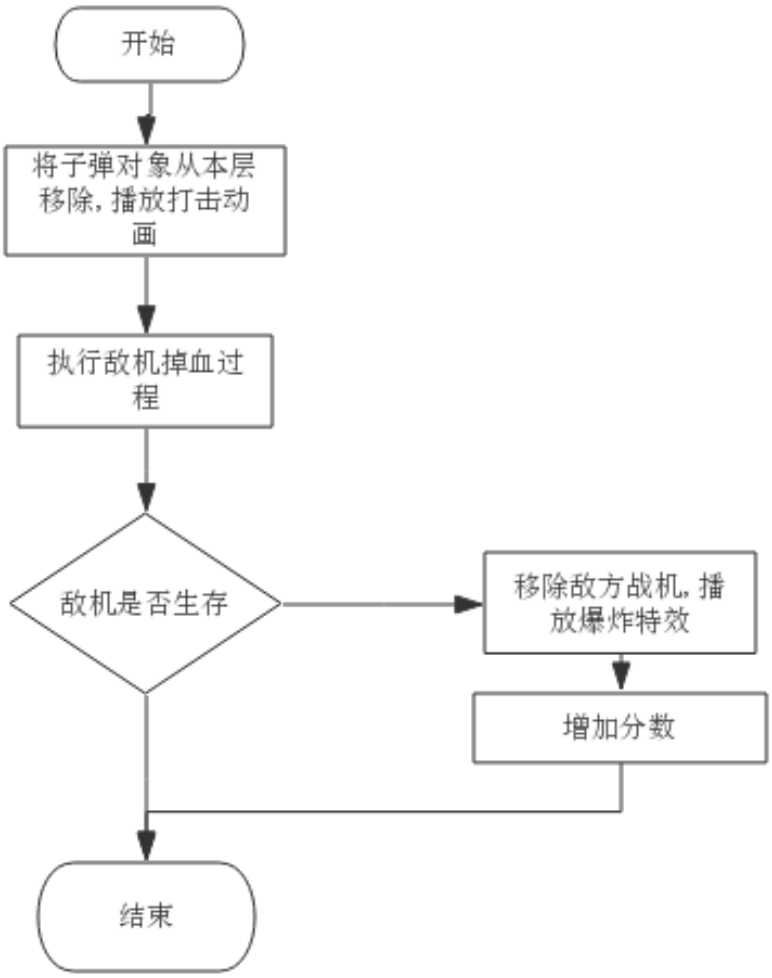

图 3-10 敌机子弹碰撞

3.3.6 触摸逻辑模块设计

触摸逻辑是控制用户与手机进行交互的模块，在本游戏中，就是控制玩家战机精灵进行躲避，攻击等一系列的操作。在原先的版本中，我们采用的是使用 Lambda 表达式将逻辑的实现写在场景代码中，在新版本的设计中，为了实现渲染代码与逻辑代码的分离，在层代码中仅仅只进行逻辑事件监听的回调函数注册我将触摸逻辑专门使用一个单独的模块来进行实现。

在事件分发部分中，我们已经了解了进行触摸监听，我们需要重写这四个方法：

onTouchesBegan，onTouchesMoved，onTouchesEnded，onTouchesCancelled。

其中，结束方法与取消方法在我们的游戏中意义不大，我并没有进行重写，仅仅重写了前两个方法，分别是：

1. 检测触摸是否发生。

2. 根据触摸对象进行响应。

检测触摸是否发生

这个功能主要由事件的 onTouchBegan 属性进行判定，这是一个返回值为布尔类型的回调函数，其作用是，当触摸事件发生时，判断触摸的对象是否是当前所监听的对象。作用原理是，获取当前的触摸坐标，和获取当前的监听对象，将监听对象的位置转换到与触摸坐标的统一坐标系，根据当前监听对象的的大小，创建一个范围盒，最后判断这个范围盒中是否包含触摸点，如果包含，则返回真，否则返回为假。

逻辑流程图的展示如下所示:

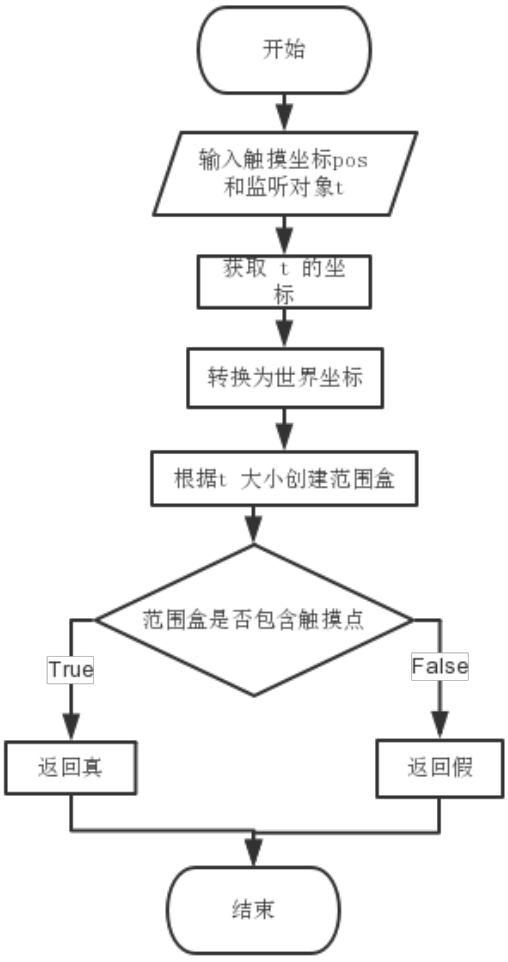

图 3-11 触摸逻辑

根据触摸对象进行响应

这个功能主要是由触摸事件的 onTouchMoved 属性进行实现，当 onTouchBegan 的返回值为真时，这个回调函数就会被调用。这个方法的实现需要两个方面，因为触摸事件为连续触摸，连

续触摸并非是一条连续的直线，而是由离散的触摸点构成的点集合，点的密集程度由屏幕的采样率决定，鉴于这样的情况，我们所做的第一步就是获取当前触摸点与上一触摸点的距离，这个距离记为 delta，是一个矢量。

获取距离变化之后，我们需要进行当前监听坐标的位置重设，将当前监听对象的位置重新设置为原先位置加上变化矢量。

3.3.7 特效设计

特效是整个游戏当中比较富有创意性的工作，并且也是难度比较大的部分，因为特效涉及到素材，设计，和如何实现的问题。首先要找到合适的素材，这些素材往往比较粗糙，并且比较简单。根据这些素材，开发者需要在脑海中构建想要做成的动画或者光效。这部分往往要考验开发者的艺术修养和创意。第三步才是根据脑海中的构想进行实现，这也是比较困难的工作，因为往往人们并不知道一个好的特效需要使用怎样的效果才能制作出来，参考别人的作品是很好的方式，但是凭借肉眼是难以辨认出复杂的光效设计。因此，制作光效还需要不断的进行尝试，修改。

在我的游戏当中，要设计和实现三种光效:

1. 子弹击中飞机后产生的打击特效
2. 飞机爆炸后的爆炸特效
3. 登陆界面的闪光光效

打击特效设计

我们使用一张静态的打击图片进行打击特效的处理，打击特效有两个过程，第一是变大，其精灵帧会有一个扩张的过程，然后是淡出，营造火花慢慢消失的效果。

同时为了使得打击特效的火花与背景进行更好的融合，我是用了颜色混合，渲染引擎会自动根据渲染函数对多层的精灵帧进行颜色混合，使得风格更加统一。

爆炸特效处理

爆炸特效是一种比较难以模拟的特效，因此不能使用单张静态的图片的方式进行模拟，但是我们可以使用连续的静态图片连续播放的方式，将其作为一个帧动画进行播放。

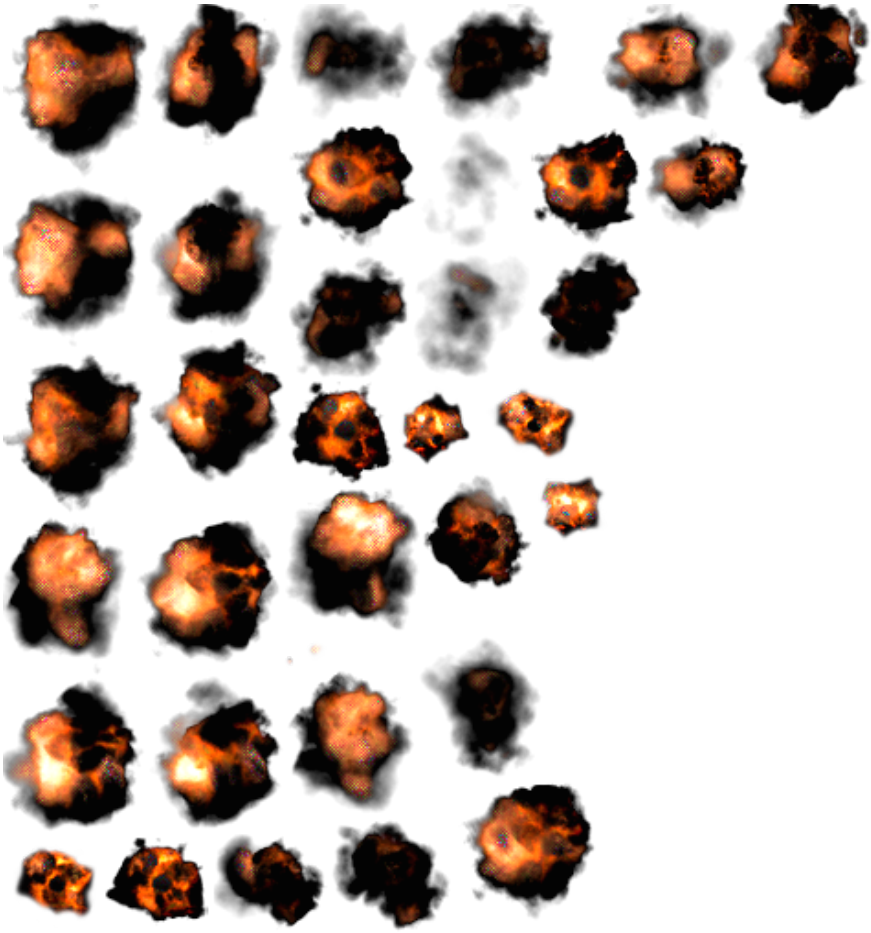

图 3-12 爆炸素材

资源中共有 35 张静态的爆炸图，在游戏的加载阶段，这些静态图片被动画引擎保存为帧动画，使用时我们直接从缓存中调用，这样避免了大量重复加载的资源浪费。

闪光光效设计

我们使用一张静态的图片，并对这个图片加上各种动作进行处理，使其成为一个比较精致的闪光特效。

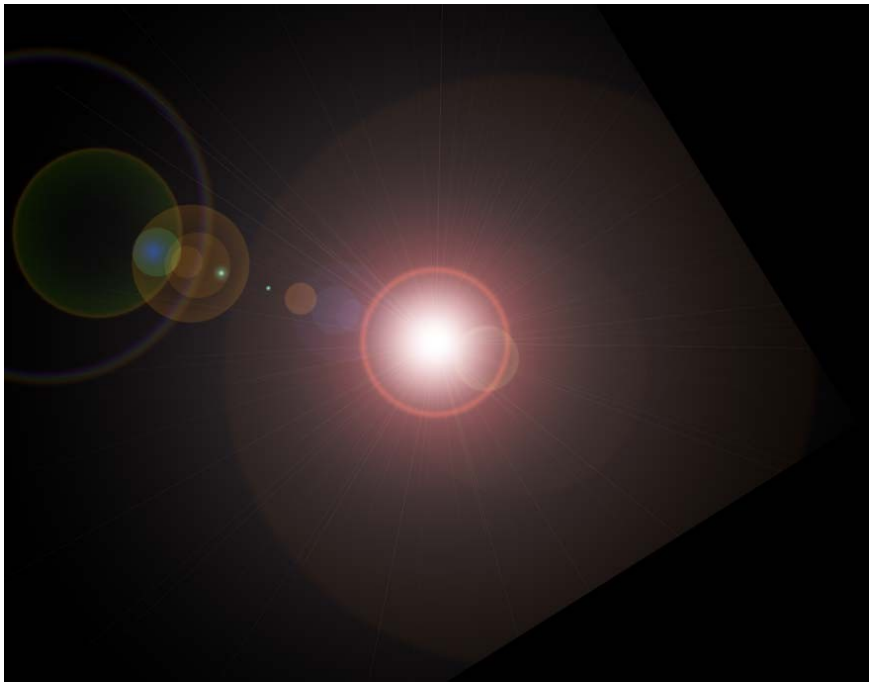

图 3-13 闪光素材

缓冲效果，是一种处理特效的常用手段，顾名思义，它可以将一个动作的执行过程变为动态的。比如，一个精灵从 A 点移动到 B 点， AB 两点之间的距离为 X，移动的持续时间为 t。则精灵的移动速度为

$$Velocity = X / t$$

这个速度值将会在整个移动期间保持恒定，也就是一个匀速运动，如果用曲线进行表示，如图

图 3-14 匀速曲线

但是在实际生活中，我们的移动往往不是保持恒定。因此使用缓冲效果可以将其改变，我们保持精灵的移动距离 X 与移动时间 t 不变，但是在移动的图中动态改变速度 V，也就是说保持图像的横坐标和积分不变，但是改变曲线的形式。用积分式可以表示为

$$X=\int_a^b vdt$$

比如下图是一个 Sin 减缓函数。

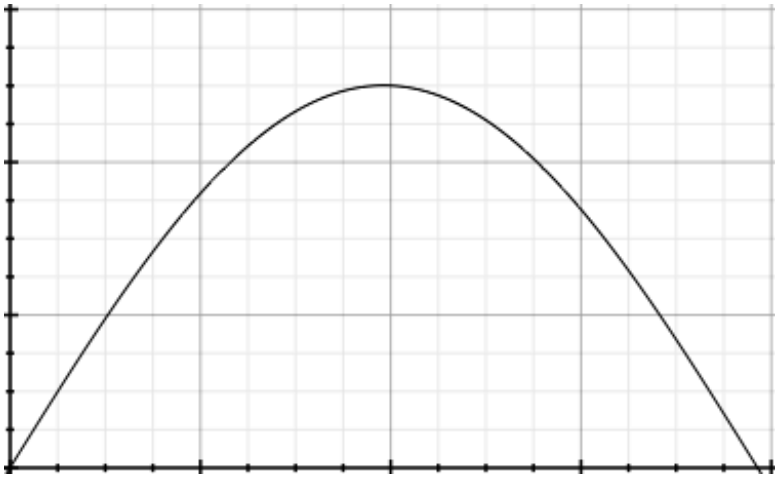

图 3-15 变速曲线

为了更加清晰的表示，我们可以将公式表达为更加通用的形式。

$$X=\int_a^b v(t)dt$$

除了移动动作， cocos2d-x 支持多种的动作，这些动作都可以使用函数进行减缓，使得可以保持更加精致的效果。因此我们可以将距离 X 更改为更加通用的参数。

光效设计

在我们的设计中，闪光光效的开始位置被设置到屏幕外，首先进行了一个淡入，玩家此时可以看到一个从中间散开的光晕，然后进行淡出，光晕消失，图片慢慢变大，然后从屏幕外移动进屏幕，最后执行旋转操作。

在我们的闪光特效中，我们使用了 Sine 移动减缓，Sine 增大减缓，组件减缓。使用以上的动作组合，最终形成了最后的闪光光效。

3.3.8 资源管理模块的设计与实现

使用 cocos2d-x 引擎制作游戏的过程当中，不可避免的要使用到多种资源文件，比如用于表示精灵的纹理文件，

用于制作帧动画的图片文件，背景音乐文件。这些文件作为程序的一部分被打包到游戏当中，需要的时候可以加载到内存中。而加载的方式有多种多样，可以从文件中直接加载，也可以采取将资源文件全部加载到资源缓存中，前者具有较好的内存占用，但是因为加载次数太多导致性能的下降，后者采用一次性加载，使用的时候直接从内存中调用，这样减少了性能的开销，但是同样的，加载资源可能在一段时间内不会被使用，导致了一部分内存的浪费，且在大型游戏中，是不可能将所有的资源一次性加载完毕的。因此，在游戏的设计中，往往会采取折中的方式，近期可能遇到的，马上就要使用的资源一次性加载，而不会使用到的暂时不加载。

在我们的游戏设计中，因为资源文件比较少，我们采用一次性将所有的资源全部加载到内存中，但仍然保留部分不常用的(比如按钮图片仅需要加载一次)直接采用文件加载的方式。

因为在整个游戏的编码过程中，同一个文件可能会在工程的不同地方所使用，这样就需要将此文件的路径抄写多次，这样既造成了代码杂乱无章，亦不利于下一步的资源替换和升级，仅仅改动一个资源文件就需要改动多出，而且这种资源缺失而造成的程序 BUG 是非常难以发现和准确定位的，因此，将资源文件出错放到编译期是一个非常好的解决办法。

在我的设计当中，资源管理模块记录所有需要加载的资源文件绝对路径，而其他使用资源的代码在引用 Resource 头文件后，可以使用其字符串常量，这样如果做出资源文件的更改，就只需要更改本模块的内容，且如果删去了部分资源路径，在编译期就可以找到工程中所有的引用。

本模块的设计主要涉及以下几个方面:

1. 使用字符串常量来记录资源文件的绝对路径。记录在 Resource 命名空间内，其他的代码文件通过引用该文件使用其值。

2. 将以 plist 文件格式的图片文件添加到帧缓存， plist 文件是一种 png 图片打包后的文件格式，可以将一组相关的 png 文件打包成一个 png 文件。但是采用这种形式的文件无法直接将图片加载，只能根据 plist 的配置文件将打包后的组合文件拆分后，分别加入到帧缓存当中。

3. 制作帧动画，并加载到帧动画缓存。帧动画是游戏的一个重要组成部分，一个帧动画往往是使用一组相关的静态图片连续播放制作而成，这些图片往往不单独使用，因此在资源加载阶段就将帧动画制作完成，并加载到动画缓存，可以节省大量的资源操作。

加载方法的设计

上文已经提到，我们需要提前将一些资源文件加载到内存中，以便于重复使用，或快速创建对象。在这个方面 cocos2d-x 提供了帧缓存，动画缓存和声音缓存来处理我们的需求。这三个缓存的管理对象为单例模式，在整个程序的运行期间只能存在这一个对象，如果一个单例对象已经被初始化，那么对这个类的重复实例化将全部返回这个已经创建好的的对象，只有这个类从未被实例化过，即没有一个此类的对象的情况下，才会进行创建。也就是说，在整个游戏的运行过程中，所有的代码将共享一个对象，这就实现了在这个场景中进行加载，而在所有的场景中都可以使用。

加载方法需要实现的功能有:

1. 加载全部的 plist 文件。
2. 创建飞行动画，这个动画主要是用于表现玩家飞机的动态效果，共有三帧。
3. 创建爆炸动画，这个动画主要用于表现飞机摧毁，共有 35 帧。
4. 将动画加载到动画缓存，并使用名字记录。

3.4 本章小结

在本章中，对游戏开发的每一个模块都做了具体详细的介绍。也记录了在开发过程中使用的一些思路和方法。

第四章 系统测试

软件测试是整个游戏设计的最终部分，是非常重要的一个步骤。因为项目并非商业项目，并且与市面上成熟的大型工程相比，我的游戏设计显得较小，因此没有必要使用自动化测试工具，或者专门的调试工具，大部分使用手动测试，性能上使用 IDE 内置的检测工具进行检测。在这种情况下，我将自己的游戏测试分为以下几个部分。

1. 单元测试，我将会对几个比较重要的模块进行单元测试，这些模块主要是游戏的逻辑处理，在视觉上可能无法观察到这些模块是否正常的进行工作，因此将这些模块进行单独的测试是非常有必要的。

2. 综合测试，在这一部分，主要是进行对游戏的综合，全面的使用，观察各个部分是否正常工作，通过具体的上手体验，找到出现 BUG 和运行失败的情况。

3. 性能测试，在这一部分，我们将对于游戏的各种参数，比如 CPU 占用，内存占用等参数进行测试，观测是否有性能瓶颈和内存泄露等情况存在。这个部分使用 IDE 内置功能进行。

4. 真机测试，在这一部分的主要功能是将游戏发布到移动端，观察游戏在实际设备上运行效果。

4.1 单元测试

因为游戏本身的特性，大部分的功能模块可以直接在玩游戏的过程中，使用肉眼进行观测，因此，大部分的游戏功能是不需要单独测试的，但是对于一些比较复杂的游戏逻辑，可能在处理上有一些问题，因此在这个部分，我们主要对以下两个模块进行测试:

1. 触摸逻辑模块

2. 碰撞逻辑模块

触摸逻辑是用户交互的一部分，我们对其测试主要通过两部分，一是测试是否进行了触摸点过滤，即是否非法相应了触摸点，二是能否正确的移动。

4.1.1 触摸逻辑模块结果展示(部分)

因为无法提供动作展示，因此在这里只打印移动数据。


```
Touch True : 137.070831, 91.654739
Touch False : 197.674225, 187.091415
```

图 4-1 触摸测试

4.1.2 碰撞逻辑模块测试（部分）

碰撞逻辑模块的测试比较繁琐，且非常影响游戏体验，但是测试的过程无法直观的呈现，因此在这里只展示一部分的测试用例和测试结果。

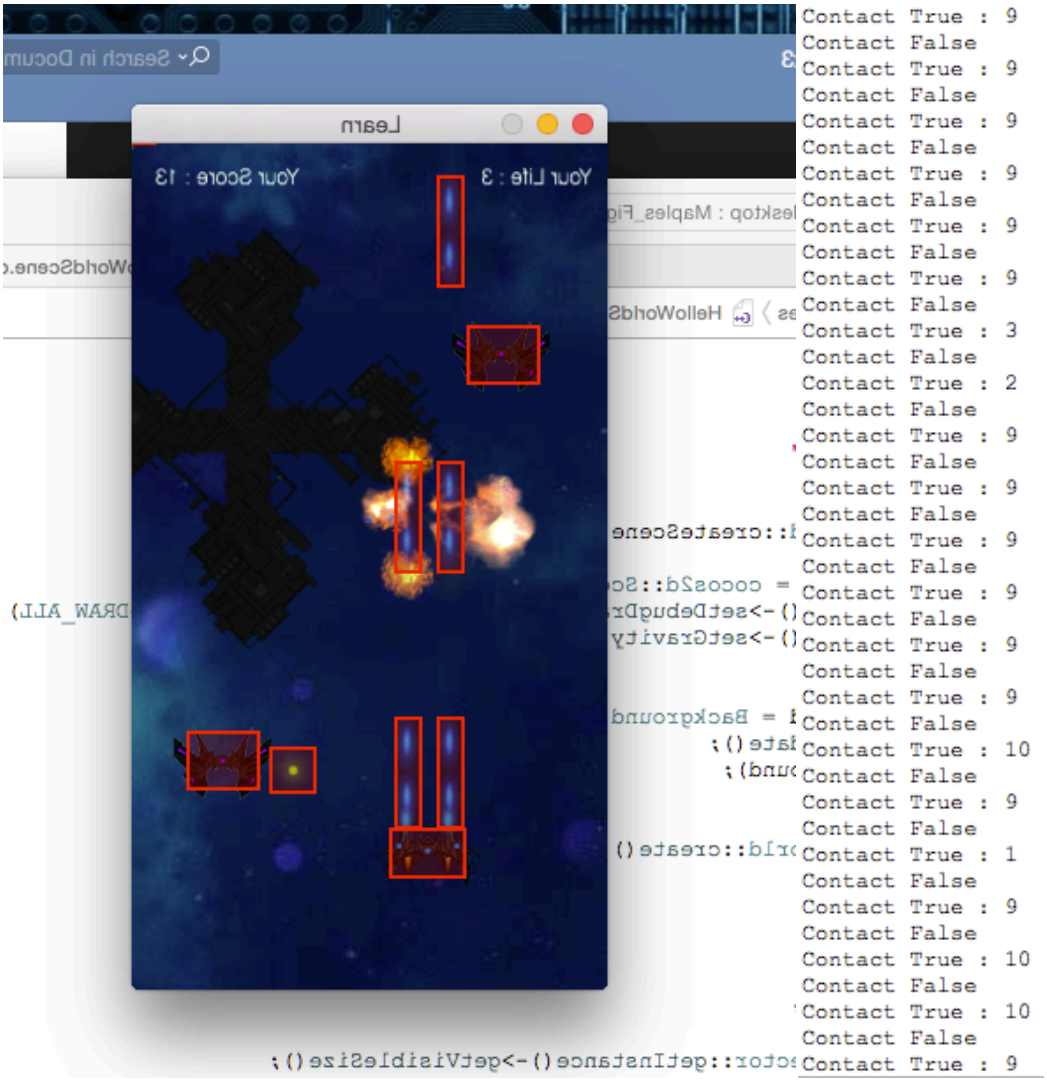

图 4-2 碰撞测试

4.2 综合测试

综合测试的是进行各个模块进行整合后的功能性测试，在文章里无法全部将测试展示出来，因此在这里仅仅选取有代表性的几张测试图进行展示。

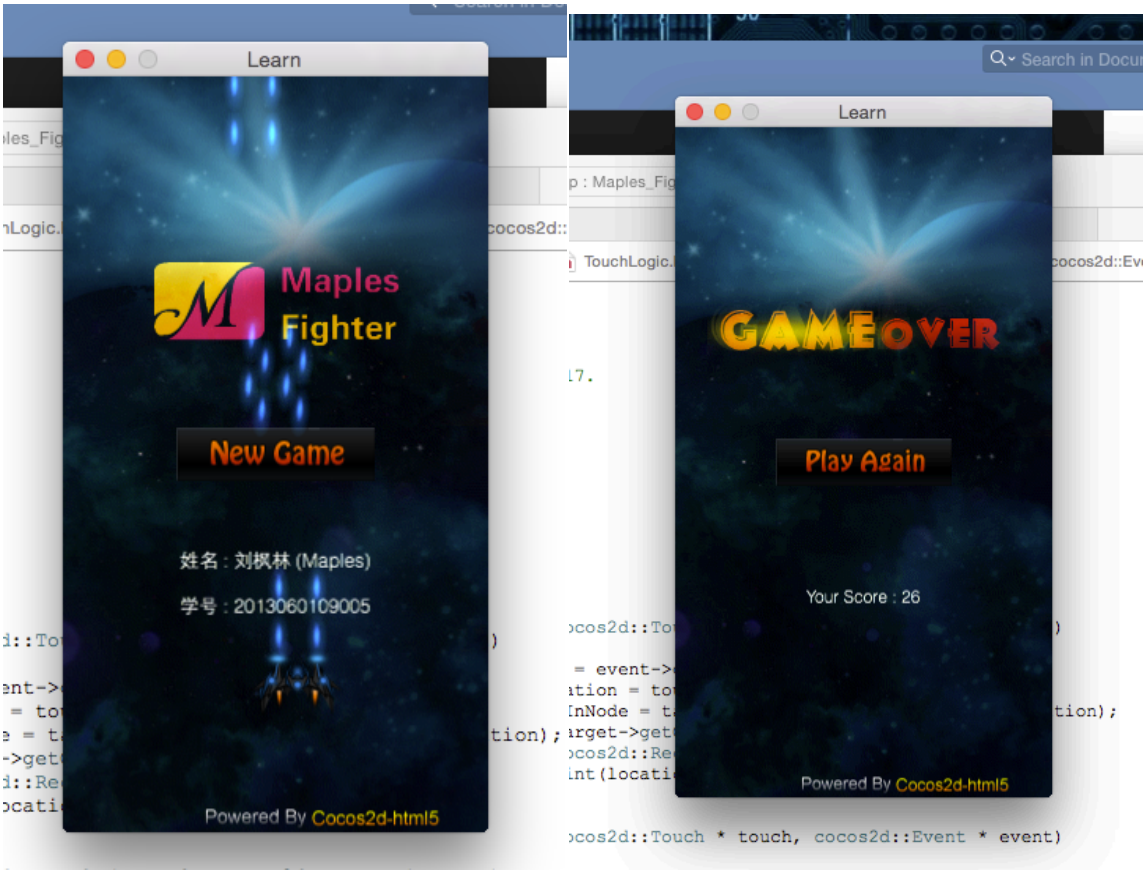

图 4-3 场景展示（一）

图 4-4 场景展示（二）

4.3 性能测试

测试平台

MacBook Pro (13-inch, Mid 2012)

Processor 2.5 GHz Intel Core i5

Memory 4 GB 1600 MHz DDR3

CPU 测试结果，游戏开发仅仅使用了单线程，导致在多核 CPU 的条件下 CPU 的利用率不够高，但是本身其占用的计算资源也比较少。

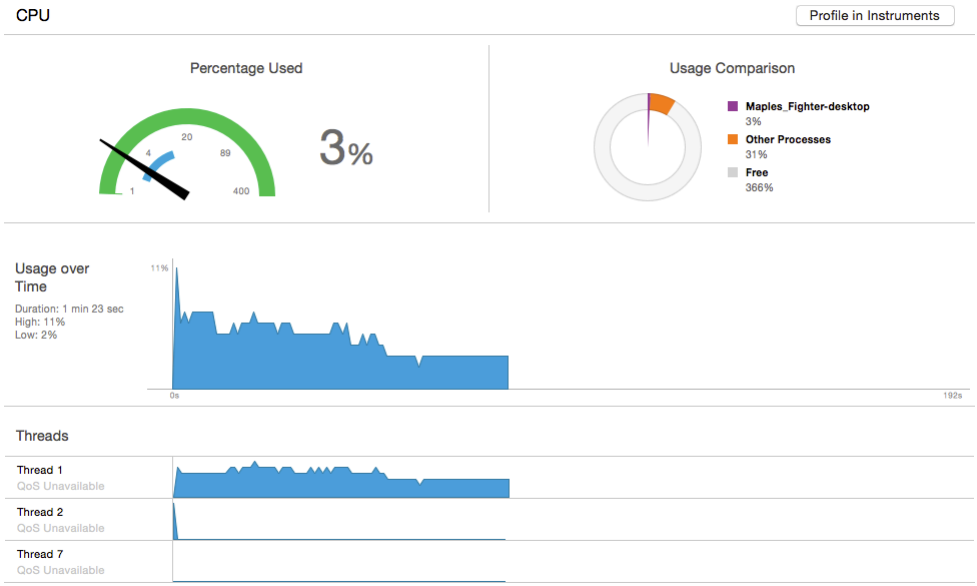

图 4-5 CPU 测试

内存测试结果，在进入游戏主场景后会有升高，结束之后的内存减少运行一段时间后没有发现明显的内存泄露。

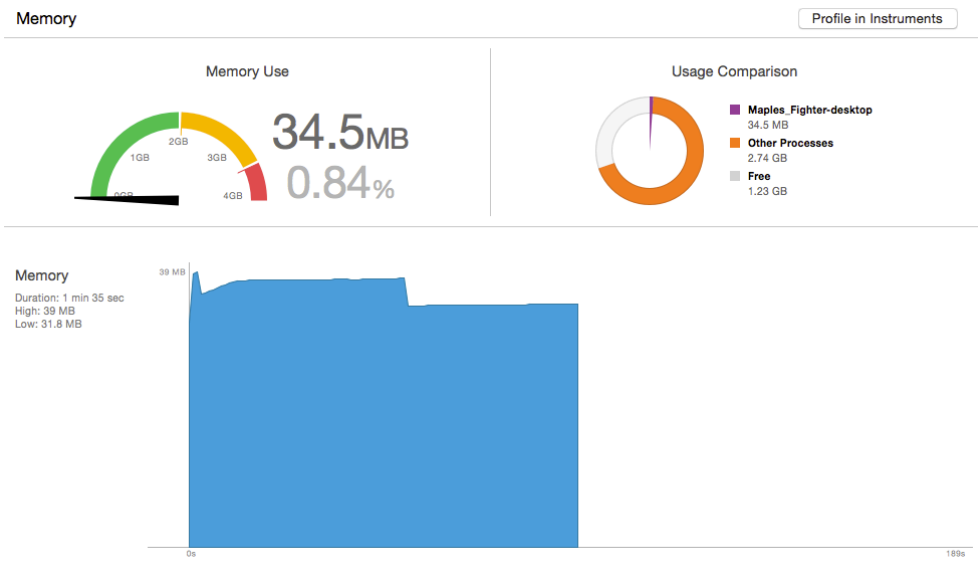

图 4-6 内存测试

功耗测试结果，CPU 唤醒次数基本可以忽略不计。

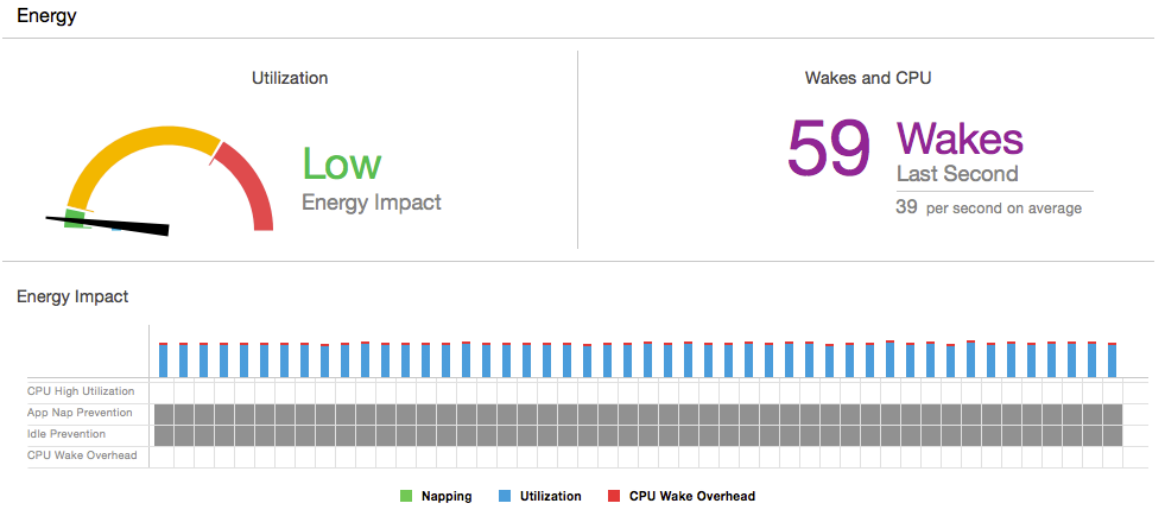

图 4-7 功耗测试

磁盘读写测试结果，基本可以忽略不计。

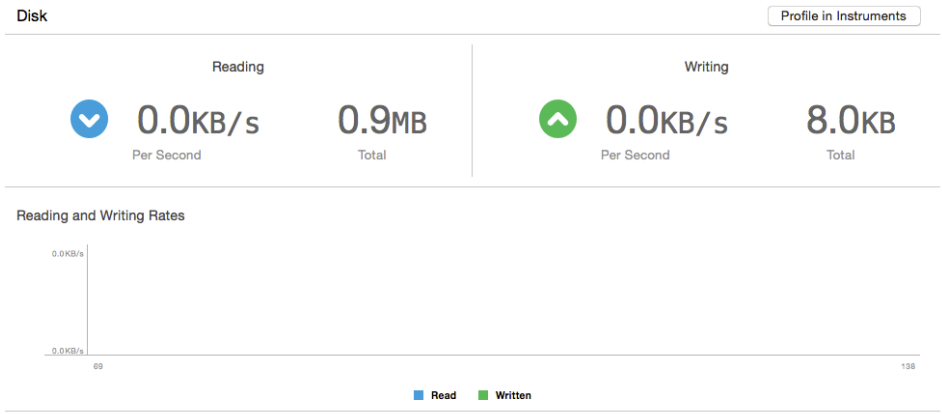

图 4-8 读写测试

4.4 真机测试

真机测试是指将我们的游戏实际的放在移动手机中运行，使得用户可以随时随地进行游戏。

在从 PC 端向移动端移植的过程当中，我们可能遇到一些问题，比如移动端的屏幕与我们设计的屏幕比例可能不同，导致在手机运行时，各个精灵的位置会失调。

cocos2d-x 引擎是一个跨平台引擎，也就是说只要可以，我们可以发布到所有的平台，在这里我们选取 MI 4LTE 设备，其基本信息如下:

Android Version: 6.0.1 MMB29M

CPU : Quad-core Max 2.5GHz

RAM : 3GB

Pixel Resolution : 1920 x 1080

测试展示

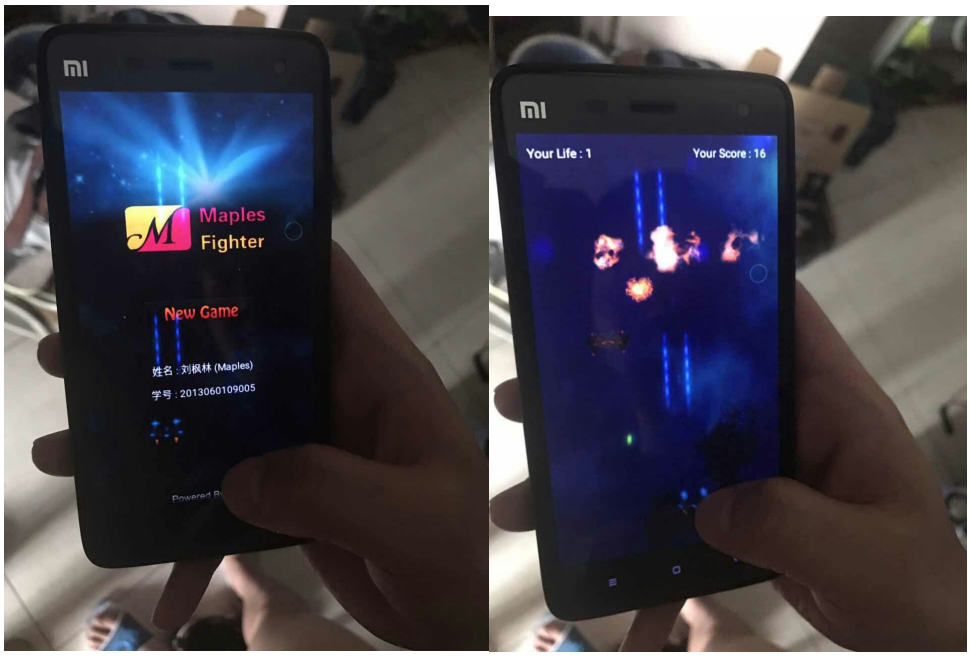

图 4-9 真机测试（一）

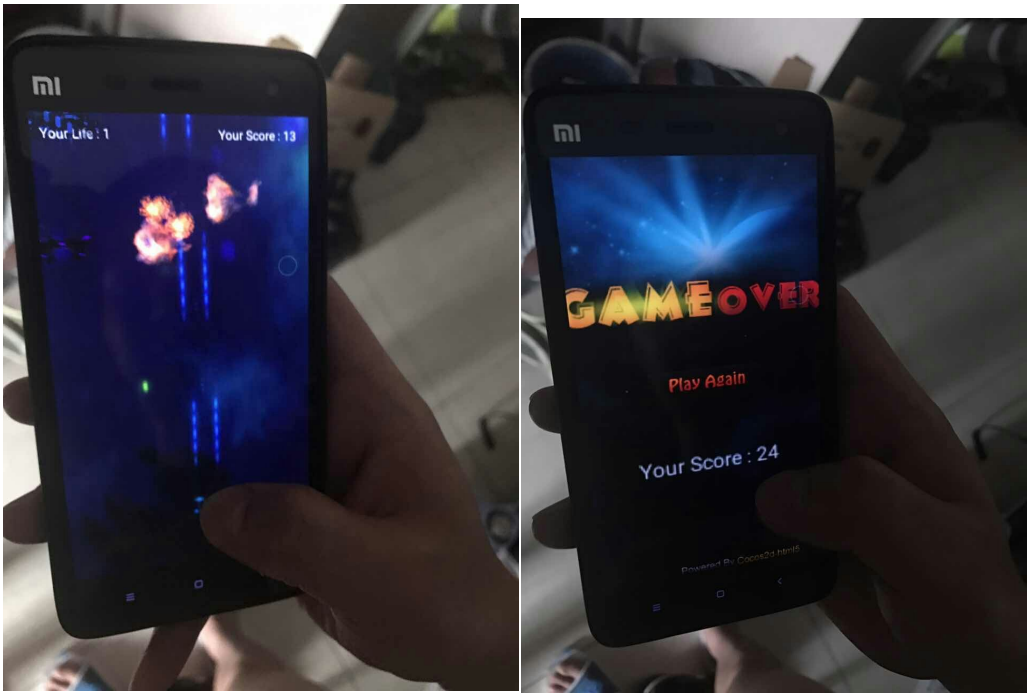

图 4-10 真机测试（二）

4.5 本章小结

在本章中，完成了项目开发的最后一步，到此，项目的开发过程的介绍已经完成。

第五章 结论

5.1 系统功能、指标分析

根据系统测试，我们可以得出结论，游戏的开发与设计基本上已经完成，并且根据测试的情况，没有太大的问题。在真机测试用，游戏运行状况良好，可玩性比较高。

整个游戏的内存占用和 CPU 占用率不高，追求性能上的优化也不是本次毕业设计目标。但是在资源加载的速度上仍然欠缺，在加载时可能会造成屏幕的闪动，影响游戏体验。

5.2 本课题有待进一步解决的问题及方向

本次毕业设计的制作与开发，仍然存在着很多不足之处，第一是对资源管理知识的欠缺，导致游戏的资源分配较差，同时加载的方式虽然有利于开发，但是非常影响游戏体验。

同时，使用物理引擎很大程度上影响了游戏的效率，导致游戏在运行过程中进行许多不必要的计算，而且物理引擎的应用并不多，没有使用到其中比较复杂和高级的模块，造成了资源的浪费。

再者，在游戏的开发过程中，仅仅使用了单线程的模型，这种模型开发比较快，而且方便调试，但是如果运行在拥有多核处理器的设备中，只利用其中一个核心的处理能力会造成非常大的资源浪费。因此，在下一步的解决中，可以考虑将不同的功能，比如渲染、计算分配到不同的线程来执行，这样可以有效的解决资源浪费的问题。

鉴于以上的情况，我们的游戏仍然需要进行进一步的改进和处理。游戏的开发是没有止境的，即使是普通的战机游戏，也有腾讯开发的战机类这种制作比较精美的大型产品。

5.3 本人收获及体会

在这次毕业设计的制作过程中，我强烈感觉到自己知识的贫乏，和对以往学习过的知识掌握不够。从最初的茫然，到慢慢的进入状态，再到对思路逐渐的清晰，整个过程难以用语言来表达。紧张而又充实的毕业设计终于落下了帷幕。回想这段日子的经历和感受，我感慨万千，在这次毕业设计的过程中，我拥有了无数难忘的回忆和收获。当选题定下来的时候，我当时便立刻着手资料的收集工作中，将搜索到的资料全部储存起来，尽量使我的资料完整、精确、数量多，这有利于毕业设计的开发。

在本次开发的过程中，我最大的收获是学习了游戏引擎的框架，认真学习一个开源的游戏引擎，不仅仅对于自己的编码水平有巨大的帮助，同时可以学习到引擎中渲染和内存管理的有关知识，对自己进一步学习和了解图形学等科研方向也有入门作用。

在开发的过程中，我深刻的认识到自己在各个方面还有不足，对于编程语言和操作系统的有关知识还是了解的不够深入，在开发过程中因为这样造成了一系列匪夷所思的问题，但是这个过程也是一个查漏补缺的过程，对于自己薄弱的、欠缺的地方，可以及时的进行补充学习。

同时，进行软件调试也是一个很好的学习部分。利用软件调试，找到可能出现的问题，或者精准的发现问题，或者是对问题进行复现，找到出错的具体原因。这些都是在开发过程中的一些宝贵经验。

5.4 本章小结

在这一章，对于整个开发过程中的收获做出总结，同时总结了整个项目的特点和不足之处。

致谢

历时将近两个月的时间终于将这篇论文写完，在论文的写作过程中遇到了无数的困难和障碍，都在同学和老师的帮助下度过了。尤其要强烈感谢我的毕业设计指导老师-邵杰老师，他对我进行了无私的指导和帮助，不厌其烦的帮助进行论文的修改和改进。另外，在校图书馆查找资料的时候，图书馆的老师也给我提供了很多方面的支持与帮助。在此向帮助和指导过我的各位老师表示最衷心的感谢!感谢这篇论文所涉及到的各位工程师。

参考文献

[1] 李华明. iOS 游戏编程之从零开始-cocos2d-x 与 cocos2d 引擎游戏开发[M].北京:清华大学出版社. 2013.21-21

[2] 满硕泉. cocos2D-x 权威指南[M].北京:机械工业出版社. 2014.19-19

[3] 刘剑卓. cocos2D-X 游戏开发技术精解[M].北京:人民邮电出版社. 2013.281-289

[4] 果冻想. cocos2D-x 入门[EB/OL].http://www.cnblogs.com/zouzf/.2014.11

外文资料原文

FIGURE 1
Stair-step effect (jaggies) produced when a line is generated as a series of pixel positions.

1 Line-Drawing Algorithms

A straight-line segment in a scene is defined by the coordinate positions for the endpoints of the segment. To display the line on a raster monitor, the graphics system must first project the endpoints to integer screen coordinates and determine the nearest pixel positions along the line path between the two endpoints. Then the line color is loaded into the frame buffer at the corresponding pixel coordinates. Reading from the frame buffer, the video controller plots the screen pixels. This process digitizes the line into a set of discrete integer positions that, in general, only approximates the actual line path. A computed line position of (10.48, 20.51), for example, is converted to pixel position (10, 21). This rounding of coordinate values to integers causes all but horizontal and vertical lines to be displayed with a stair-step appearance (known as "the jaggies"), as represented in Figure 1. The characteristic stair-step shape of raster lines is particularly noticeable on systems with low resolution, and we can improve their appearance somewhat by displaying them on high-resolution systems. More effective techniques for smoothing a raster line are based on adjusting pixel intensities along the line path (see Section 15 for details).

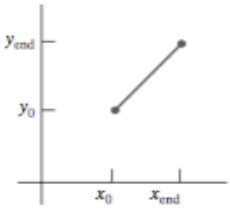

FIGURE 2
Line path between endpoint positions (x_0, y_0) and (x_{end}, y_{end}).

Line Equations

We determine pixel positions along a straight-line path from the geometric properties of the line. The Cartesian *slope-intercept equation* for a straight line is

$$y = m \cdot x + b \tag{1}$$

with m as the slope of the line and b as the y intercept. Given that the two endpoints of a line segment are specified at positions (x_0, y_0) and (x_{end}, y_{end}), as shown in Figure 2, we can determine values for the slope m and y intercept b with the following calculations:

$$m = \frac{y_{end} - y_0}{x_{end} - x_0} \tag{2}$$

$$b = y_0 - m \cdot x_0 \tag{3}$$

Algorithms for displaying straight lines are based on Equation 1 and the calculations given in Equations 2 and 3.

For any given x interval δx along a line, we can compute the corresponding y interval, δy, from Equation 2 as

$$\delta y = m \cdot \delta x \tag{4}$$

Similarly, we can obtain the x interval δx corresponding to a specified δy as

$$\delta x = \frac{\delta y}{m} \tag{5}$$

These equations form the basis for determining deflection voltages in analog displays, such as a vector-scan system, where arbitrarily small changes in deflection voltage are possible. For lines with slope magnitudes $|m| < 1$, δx can be set proportional to a small horizontal deflection voltage, and the corresponding vertical deflection is then set proportional to δy as calculated from Equation 4. For lines

whose slopes have magnitudes $|m| > 1$, δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to δx, calculated from Equation 5. For lines with $m = 1$, $\delta x = \delta y$ and the horizontal and vertical deflections voltages are equal. In each case, a smooth line with slope m is generated between the specified endpoints.

On raster systems, lines are plotted with pixels, and step sizes in the horizontal and vertical directions are constrained by pixel separations. That is, we must "sample" a line at discrete positions and determine the nearest pixel to the line at each sampled position. This scan-conversion process for straight lines is illustrated in Figure 3 with discrete sample positions along the x axis.

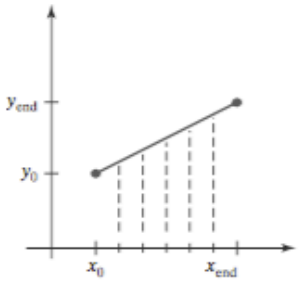

FIGURE 3
Straight-line segment with five sampling positions along the x axis between x_0 and x_{end}.

DDA Algorithm

The *digital differential analyzer* (*DDA*) is a scan-conversion line algorithm based on calculating either δy or δx, using Equation 4 or Equation 5. A line is sampled at unit intervals in one coordinate and the corresponding integer values nearest the line path are determined for the other coordinate.

We consider first a line with positive slope, as shown in Figure 2. If the slope is less than or equal to 1, we sample at unit x intervals ($\delta x = 1$) and compute successive y values as

$$y_{k+1} = y_k + m \tag{6}$$

Subscript k takes integer values starting from 0, for the first point, and increases by 1 until the final endpoint is reached. Because m can be any real number between 0.0 and 1.0, each calculated y value must be rounded to the nearest integer corresponding to a screen pixel position in the x column that we are processing.

For lines with a positive slope greater than 1.0, we reverse the roles of x and y. That is, we sample at unit y intervals ($\delta y = 1$) and calculate consecutive x values as

$$x_{k+1} = x_k + \frac{1}{m} \tag{7}$$

In this case, each computed x value is rounded to the nearest pixel position along the current y scan line.

Equations 6 and 7 are based on the assumption that lines are to be processed from the left endpoint to the right endpoint (Figure 2). If this processing is reversed, so that the starting endpoint is at the right, then either we have $\delta x = -1$ and

$$y_{k+1} = y_k - m \tag{8}$$

or (when the slope is greater than 1) we have $\delta y = -1$ with

$$x_{k+1} = x_k - \frac{1}{m} \tag{9}$$

Similar calculations are carried out using Equations 6 through 9 to determine pixel positions along a line with negative slope. Thus, if the absolute value of the slope is less than 1 and the starting endpoint is at the left, we set $\delta x = 1$ and calculate y values with Equation 6. When the starting endpoint is at the right (for the same slope), we set $\delta x = -1$ and obtain y positions using Equation 8. For a negative slope with absolute value greater than 1, we use $\delta y = -1$ and Equation 9, or we use $\delta y = 1$ and Equation 7.

This algorithm is summarized in the following procedure, which accepts as input two integer screen positions for the endpoints of a line segment. Horizontal and vertical differences between the endpoint positions are assigned to parameters `dx` and `dy`. The difference with the greater magnitude determines the value of parameter `steps`. This value is the number of pixels that must be drawn beyond the starting pixel; from it, we calculate the x and y increments needed to generate

the next pixel position at each step along the line path. We draw the starting pixel at position (x0, y0), and then draw the remaining pixels iteratively, adjusting x and y at each step to obtain the next pixel's position before drawing it. If the magnitude of dx is greater than the magnitude of dy and x0 is less than xEnd, the values for the increments in the x and y directions are 1 and m, respectively. If the greater change is in the x direction, but x0 is greater than xEnd, then the decrements -1 and $-m$ are used to generate each new point on the line. Otherwise, we use a unit increment (or decrement) in the y direction and an x increment (or decrement) of $\frac{1}{m}$.

```
#include <stdlib.h>
#include <math.h>

inline int round (const float a)  { return int (a + 0.5); }

void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
   int dx = xEnd - x0,  dy = yEnd - y0,  steps,  k;
   float xIncrement, yIncrement, x = x0, y = y0;

   if (fabs (dx) > fabs (dy))
      steps = fabs (dx);
   else
      steps = fabs (dy);
   xIncrement = float (dx) / float (steps);
   yIncrement = float (dy) / float (steps);

   setPixel (round (x), round (y));
   for (k = 0; k < steps; k++) {
      x += xIncrement;
      y += yIncrement;
      setPixel (round (x), round (y));
   }
}
```

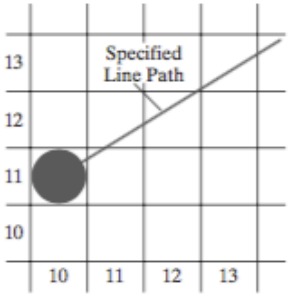

FIGURE 4
A section of a display screen where a straight-line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

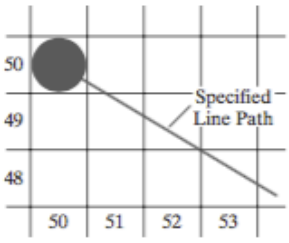

FIGURE 5
A section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

The DDA algorithm is a faster method for calculating pixel positions than one that directly implements Equation 1. It eliminates the multiplication in Equation 1 by using raster characteristics, so that appropriate increments are applied in the x or y directions to step from one pixel position to another along the line path. The accumulation of round-off error in successive additions of the floating-point increment, however, can cause the calculated pixel positions to drift away from the true line path for long line segments. Furthermore, the rounding operations and floating-point arithmetic in this procedure are still time-consuming. We can improve the performance of the DDA algorithm by separating the increments m and $\frac{1}{m}$ into integer and fractional parts so that all calculations are reduced to integer operations. A method for calculating $\frac{1}{m}$ increments in integer steps is discussed in Section 10. In the next section, we consider a more general scan-line approach that can be applied to both lines and curves.

Bresenham's Line Algorithm

In this section, we introduce an accurate and efficient raster line-generating algorithm, developed by Bresenham, that uses only incremental integer calculations. In addition, Bresenham's line algorithm can be adapted to display circles and other curves. Figures 4 and 5 illustrate sections of a display screen where

外文资料译文

画线算法

场景中的直线段由其两端点的坐标位置来定义。要在光栅显示器上显示一条线段，图形系统必须将两端点投影到整数屏幕坐标，并确定离两端点的直线路径最近的像素位置。接下来将颜色值装入帧缓存相应的像素坐标处。视频控制器从帧缓存读出写入的颜色并绘制屏幕像素。这一过程将一条线段数字化为一组离散的整数位置。一般而言，这些位置是实际线段路径的近似。例如，计算出的线段上的位置（10.48,20.51）转换为像素位置（10,21）。坐标值舍人到整数，引起除水平和垂直以外所有线段的阶梯效应（锯齿形），如图

光栅线段特有的阶梯现象在低分辨率系统中特别容易看出来，而在高分辨率系统中可以得到改善。更有效的平滑光栅线段的技术是基于调整线段路径上的像素强度（细节参照 15 节）。

直线方程

根据直线的几何特征可确定直线路径的像素位置。直线的笛卡儿斜率截距方程为

$$y = m \cdot x + b \qquad (1)$$

其中，m 为直线的斜率，b 为 y 轴截距。给定图所示线段的两个端点(x_0, y_0)和(x_{end}, y_{end})，可以计算斜率 m 和 y 轴截距 b：

$$m = \frac{y_{end} - y_0}{x_{end} - x_0} \qquad (2)$$

$$b = y_0 - m \cdot x_0 \qquad (3)$$

显示直线段的算法以直线方程 1 及式 2，3 中计算方法为基础。

对于任何沿直线给定的 x 增量 δx，可以从式 2 中计算出对应的 y 增量 δy。

$$\delta y = m \cdot \delta x \qquad (4)$$

同样，可以得出对应于指定的 x 增增量 δx

$$\delta x = \frac{\delta y}{m} \qquad (5)$$

这些方程形成了模拟设备（如向量扫描系统）中确定偏转电压的基础，其中有可能造成微小的偏转电压变化。对于具有斜率绝对值|m|<1 的直线，可以设置一个较小的水平偏转电压 δx，对应的垂直偏转电压则可以使用式 4 计算出来的 δy 来设定；而对于斜率值|m|>1 的直线，则设置一个比较小的垂直偏转电压 δy，对应的水平偏转电压则由式 5 中计算出来的 δx 来设定；对于斜率 m=1

的直线，δx=δy，因此水平偏转和垂直偏转电压相等。在每一种情况下，都可以在指定的端点之间生成一条斜率为 m 的平滑直线段。

在光栅系统中，通过像素绘制线段，水平和垂直方向的步长受到像素的间距的限制。也就是必须在离散位置上对线段进行取样，并且在每个取样位置上确定距离线段最近的像素。图 3 给出了线段的扫描转换过程，及相对 x 轴的离散取样点位置。

DDA 算法

数字微分分析仪(digital differential analyzer，DDA)方法是一种线段扫描转换算法，基于便用式 4 或式 5 来计算 δx 或 δy，在一个坐标轴上以单位间隔对线段取样，从而确定另一个坐标轴上最靠近线路径的对应整数值。

首先考虑如图 2 所示的具有正斜率的线段。例如，如果斜率小于等于 1，则以单位 x 间隔(δx=1)取样，并逐个计算每一个 y 值，

$$y_{k+1} = y_k + m \tag{6}$$

下标 k 取整数值，从第一个点 1 开始递增直至最后的端点。由于 m 可以是 0 与 1 之间的任意实数，所以计算出的 y 值必须取整。

对于具有大于 1 的正斜率的线段，则交换 x 和 y 的位置。也就是以单位 y 间隔(δy =1)取样顺序计算每个 x 值

$$x_{k+1} = x_k + \frac{1}{m} \tag{7}$$

此时，每一个计算出的 x 值要沿 y 扫描线舍人到最近的像素位置。

式 6 和式 7 基于从左端点到右端点处理线段的假设（参见图 2 ）。假如这个过程中的处理方向相反，即起始端点在右侧，那么 δx= -1,并且

$$y_{k+1} = y_k - m \tag{8}$$

或者（当斜率大于 1 时）是 δy = -1，并且

$$x_{k+1} = x_k - \frac{1}{m} \tag{9}$$

式 6 到式 9 也可以用来计算具有负斜率的线段的像素位置。假如斜率的绝对值小于 1，并且起始端点在左侧，可设置 δx=1 并用式 6 计算 y 值。当起始端点在右侧（具有相同的斜率）时，我们可设置 δx= -1 并且由式 8 得到 y 的位置。同样，负斜率的绝对值大于 1 时可以使用 δy=-1 和式 9 或者 δy=1 和式 7 进行计算。

这个算法可以概括为下面的过程：输人线段两个端点的像素位置。端点位置间水平和垂直的差值赋给参数 dx 和 dy。绝对值大的参数确定参数 steps 的值。该值也是在即将画出的这条线段上

的像素数目；按照这个数值，沿线段路径计算每一步的下一个像素位置。先绘制起始位置 （x_0，y_0）的像素，然后调整每一步的 x 和 y，获得并逐一绘制余下的像素。假如 dx 的绝对值大于 dy 的绝对值，且 x0 小于 xEnd，那么 x 和 y 方向的增量值分别为 1 和 m。假如 x 方向的变化较大，但 x0 大于 xEnd，那么就采用减量-1 和-m 来生成线段上的每个点。在其他情况下，y 方向使用单位增量（或减量），x 方向使用 1/m 的增量(或减量)。

```
#include <stdlib.h>
#include <math.h>

inline int round (const float a)
{
   return int (a + 0.5);
}

void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
   int dx=xEnd-x0, dy=yEnd-y0, steps, k;
   float xIncrement, yIncrement, x = x0, y = y0;
   if (fabs (dx) > fabs (dy))
      steps = fabs (dx);
   else
      steps = fabs (dy);

   xIncrement = float (dx) / float (steps);
   yIncrement = float (dy) / float (steps);
   setPixel (round (x), round (y));

   for (k = 0; k < steps; k++)
   {
      x += xIncrement;
      y += yIncrement;
      setPixel (round (x), round (y));
   }
}
```

DDA 方法计算像素位置要比直接使用直线方程 1 计算的速度更快。它利用光撖特性消 除了直线方程 1 中的乘法，而在 x 或 y 方向使用合适的增量，从而沿线路径逐步得到各像素的位置。但在浮点增量的连续叠加中，取整误差的积累使得对于较长线段所计算的像素位置偏离实际线段。而且该过程中的取整操作和浮点运算仍然十分耗时。我们可以通过将增量 m 和 1/m 分离成整数和

小数部分，从而使所有的计算都简化为整数操作来改善 DDA 算法的性能。在 10 节将讨论用整数步长计算 1/m 增量的方法。在下一节中,我们考虑既能用于直线又能用于曲线的 更通用的扫描线程序。