

从零开始 **Android** 游戏编程(第二版)

作者：邢野 时间：2009-9-25

MSN:xingye(at)sohu.com

目录

前言	3
第一版前言	5
第一章 搭建开发环境	6
第二章 创建第一个程序 Hello Tank	10
第三章 显示文字和图片	17
第四章 响应用户事件	30
第五章 小结——扫雷游戏的实现	36
第六章 SurfaceView 动画	39
第七章 精灵、帧动画与碰撞检测	46
第八章 地图的设计和实现	62
第九章 游戏程序的生命周期	75
第十章 游戏循环的设计	86
第十一章 演员 (Actor)、视口 (ViewWindow)，演出开始	88
第十二章 音乐与音效	95

前言

没想到重新打开这篇文档已经是一年之后了。

去年三月，我停止了这一系列文章的写作。六月，离开了工作了五年的公司。作为公司的创始人和业务主管，我不能容忍它发展的如此缓慢，而合伙人却很享受这种慢节奏的生活。九月底，去了西藏，一如期待中的蓝天白云，雄伟的布达拉宫，美丽的雅鲁藏布江和轻微的高原反应。开车的刘师傅是个很懂生活的人，当他用一个急转弯将羊卓雍错推入我们眼帘的时候，每个人都惊呆了。原来照片上的颜色不是 PS 出来的！世界上真的有这么美丽的湖！从高原下来到成都、长沙、武汉。回到了久违的母校，见到了阔别多年的队长和同学。才发现不光是高原的美景能让人震撼，久别重逢一样能冲击人的心灵。当那些青春的记忆一瞬间被撩起，不知是不是温暖，不知是不是凄凉。人，也笑了，酒，也醉了。

“南巡”归来，心中一下少了很多杂念，或许是江山如画让人更觉人生苦短吧。既然自己对移动开发这么感兴趣，何不干脆寄情于此，忘掉什么 web，忘掉什么 ERP。于是 iPad, MacBook 摆上了书桌，多年不用的 C 语言重新捡起。arm、toolchain、xcode、跨平台、交叉编译。陌生，熟悉，可憎，可爱。突然发现，原来自己还是喜欢学习，一把年纪了，还能觉得吸收知识是一个快乐的过程。

过程虽然曲折，前途却有光明。看到同一个程序在 windows, mac, ios, android 都可以运行的时候，自己都有点“马德里不思议”的感觉。

两个来月的学习、尝试终于有所收获。恰在这时，又一封关于这一教程的邮件出现在了邮箱。如同前面所有的邮件一样，xiangyang 也想知道这个系列教程还有没有续集。很遗憾，它没有。自从我下定决心制作一个跨平台的框架之后，就放弃了对 java SDK 的探索。我喜欢 java 的干净快捷，但是我想更集中精力。当取得了阶段性成果之后，我很想让自己紧绷的神经放松一下。于是我对 xiangyang 说“用一周的时间完成这个教程，以减少读者的遗憾”，这也就是大家在后面将看到的。

一些变化：

从现在开始，不再使用 OPhone SDK。Google 的新工具非常好，可以管理多个 SDK 版本，使用起来非常方便。而相比之下，OPhone 老湿就太不给力了，还在吃 Cupcake 吧，别吃了，再吃噎到了。

另外，我重新整理了随书源码。让大家使用起来更方便。

最后，很多人希望我能有个 blog 以方便交流，所以本文将通过我的 csdn 博客重新发布。其实我以前在 javaeye 上发过，可没想到 javaeye 上竟然有那么多脑残，看到 OPhone 就像看到他母亲的裸照一样不能容忍，纷纷想拿五毛钱买回去珍藏！？你瞧不起它并不代表你就比它优秀，真的。

邢野

2011-1-10

http://item.taobao.com/auction/item_detail.htm?item_num_id=9077643779

第一版前言

什么是 OPhone

OPhone 是基于 Linux、面向移动互联网的终端基础软件及系统解决方案。

OPhone SDK 是专为 OPhone 平台设计的软件开发套件，它包括 OPhone API，OPhone 模拟器，开发工具，示例代码和帮助文档(摘自 OPhone 官方网站：<http://www.ophonesdn.com/>)。

简而言之，OPhone 是一个移动终端的操作系统，移动终端包括手机、MID、NetBook 等等。

与其他领域的编程一样，OPhone 编程并没有什么神秘之处，只需简单的学习就可以掌握大部分的概念。剩下的就是尽情发挥你的想象力了。

写作本文的目的

为了普及 OPhone 编程的基本知识，并通过复刻一个坦克大战游戏让读者了解 2D 游戏编程的简单思路。文中的程序结构和实现方法并非最优，希望能起到一个抛砖引玉的作用，让更多的人加入到 OPhone 开发的行列中来。

谁适合阅读本文

虽然本文叫做“从零开始 OPhone 编程”，但并不能面对那些对编程一无所知的读者。实际上，本文要求读者了解 java 语言的基本知识，最好会使用 eclipse。在文章的每个章节都标有难度，有能力的读者完全可以跳过相对容易的章节直接阅读自己感兴趣的内容。

本文的时效性

本文只适合当前版本的 OPhone SDK (v1.0)，本文的代码、图片、链接可能会因时间推移而失效。

第一章 搭建开发环境

难度：容易

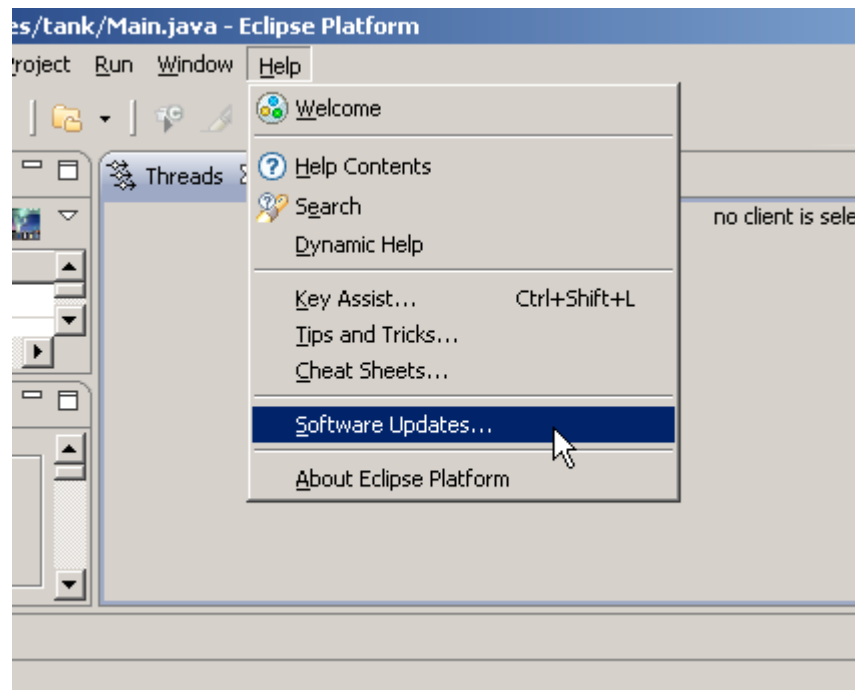
工欲善其技，必先利其器。我们要做的第一件事就是搭建 **Android** 开发环境。本文只介绍 **Windows** 下的安装方法，**Linux** 下的安装方法请参考官方网站的介绍。

与 **PC** 编程略有不同的是，**Android** 的程序需要在模拟器中运行。因此，我们需要一个集成开发环境，一个 **SDK** 和一个模拟器。因为 **Android** 编程使用 **java** 语言，所以我们还需要 **JDK**，最好使用安装版本(<http://java.sun.com/javase/downloads/index.jsp>)选用 **JDK 6 Update 16 Windows** 版即可。

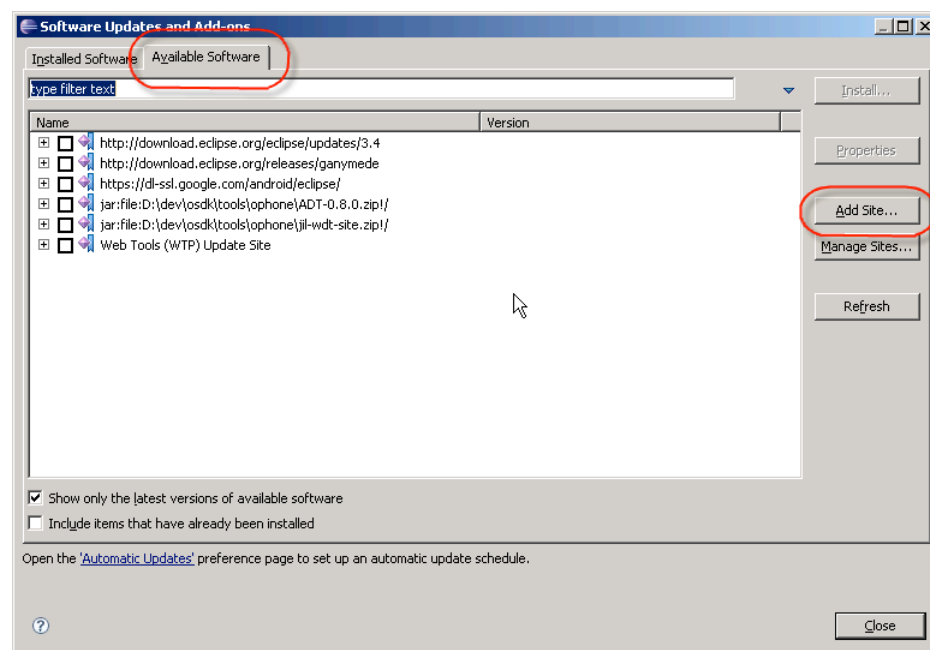
集成开发环境我们选用 **eclipse**，可以使用 **eclipse3.3** 到 **3.5** 的任意版本（<http://download.eclipse.org/eclipse/downloads/>）最好下载 **JDT** 集成版。然后我们可以从 **Android** 官方网站(<http://www.android.com>)下载 **Android SDK**（当然，如果你不能翻墙，可以到国内的网站下载），**SDK** 全部安装完毕之后，还需要安装 **eclipse** 插件。插件是用来扩展 **eclipse** 功能的。开发 **Android** 用的插件叫 **ADT**（**Android Developer Tools**），它可以帮助我们完成创建项目，向模拟器部署并运行程序，调试程序等工作。关于 **ADT** 的功能，在后面使用中我们会逐渐熟悉。

安装 **ADT** 的方法如下（以 **eclipse3.4** 为例）：

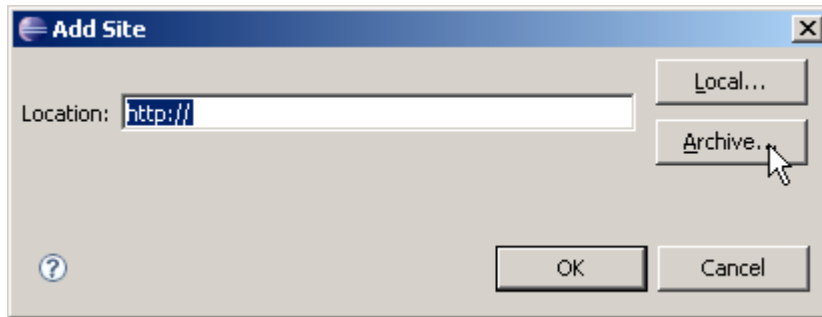
启动 **eclipse**，选择菜单中的 **Help -> Software Updates**



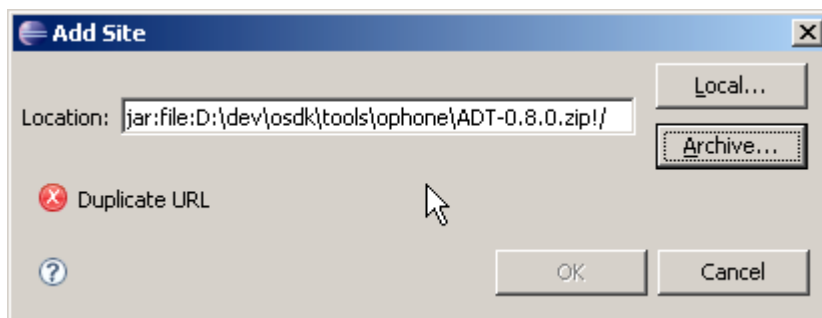
点击 Add Site



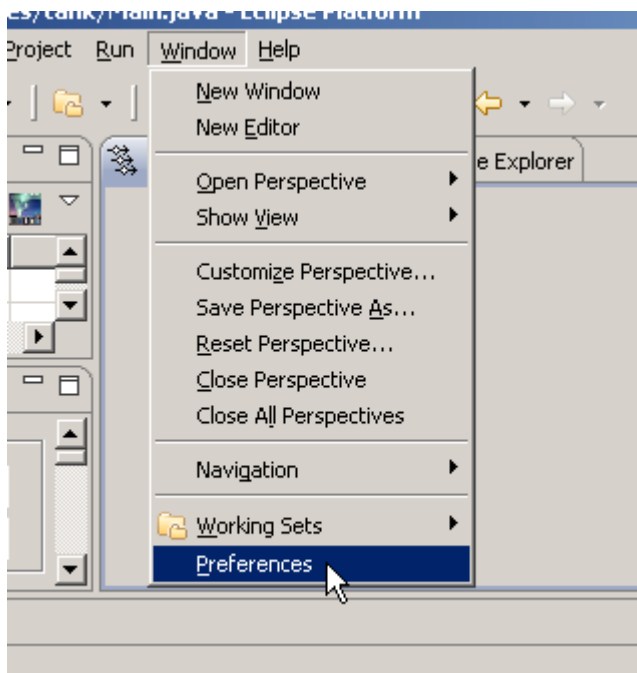
点击 Archive...



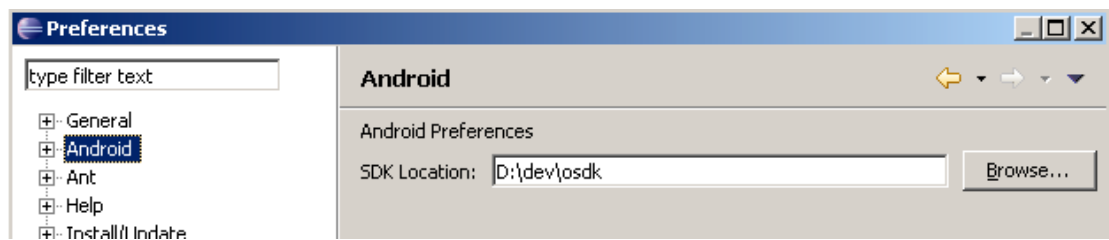
找到 OPhone SDK 安装目录下 tools\ophone\ ADT-0.8.0.zip(因为我已经安装好了 ADT，所以出现了重复 URL 的提示)，点击 OK 即可开始安装



ADT 安装完毕后还要简单配置一下，打开菜单中的 Window -> Preferences



找到 Android 项，通过 Browse 按钮指定 Android SDK 的安装位置



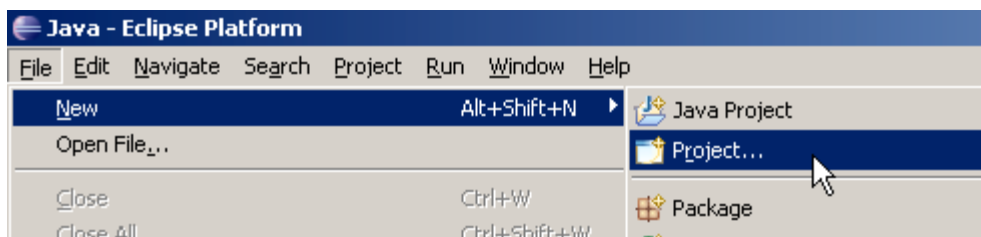
至此为止，Android 的安装环境就全部搭建完毕了。下一章节，我们会遇见经典的 helloworld，下章见！

第二章 创建第一个程序 Hello Tank

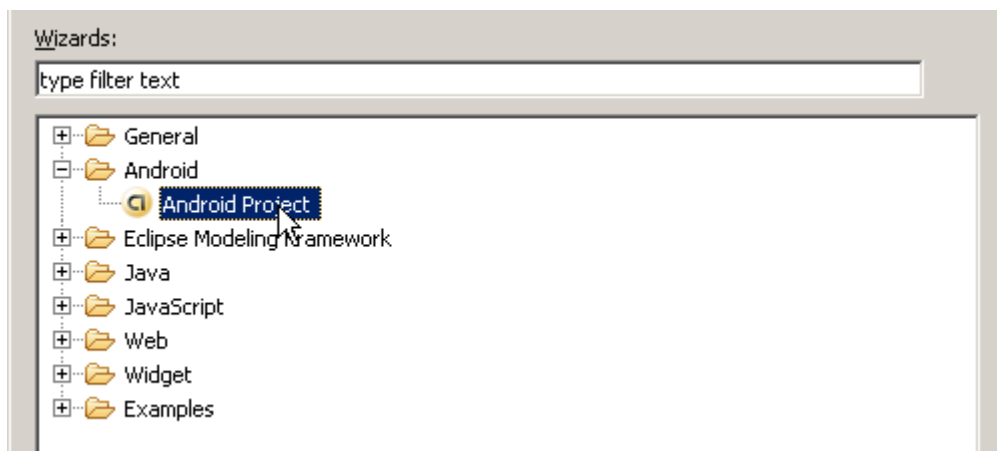
难度：容易

现在开始，我们要真正写作 **Android** 程序了。虽然前面安装过程那么复杂，但是写起程序来却是非常简单。而且为了让大家有一个直观的认识，本文不会叙述大段的原理，而是在编码的过程中渗透对原理、概念的讲解。

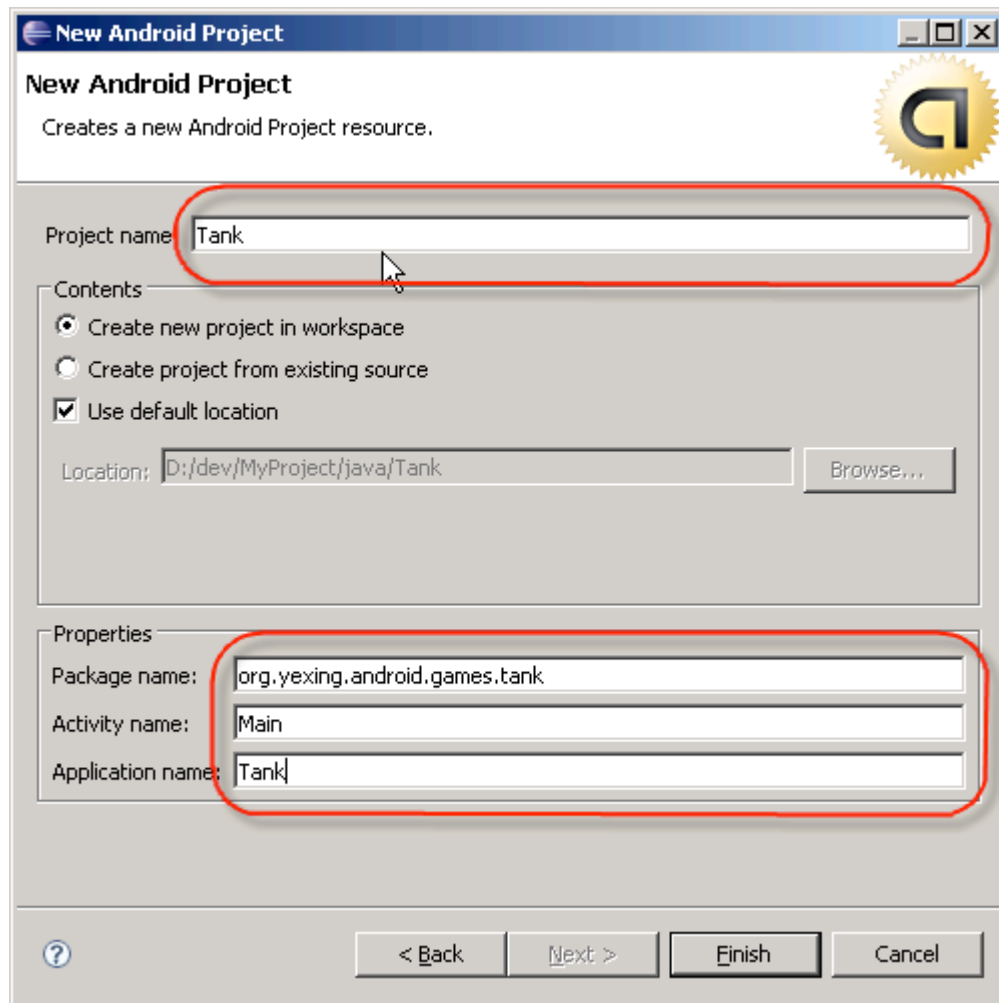
让我们打开 **eclipse**，选择菜单中的 **File -> New -> Project...**



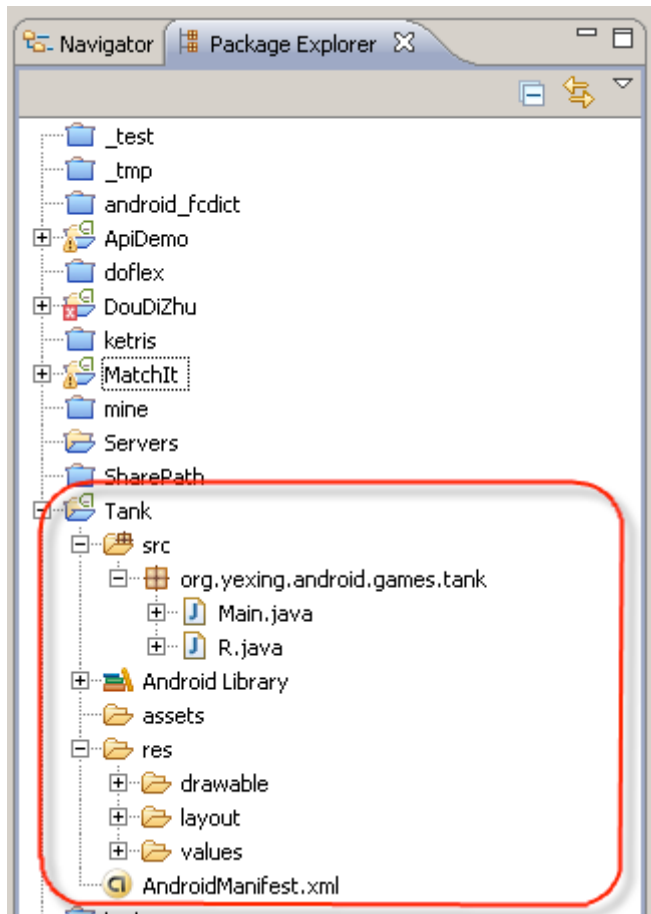
选择 **Android -> Android Project**



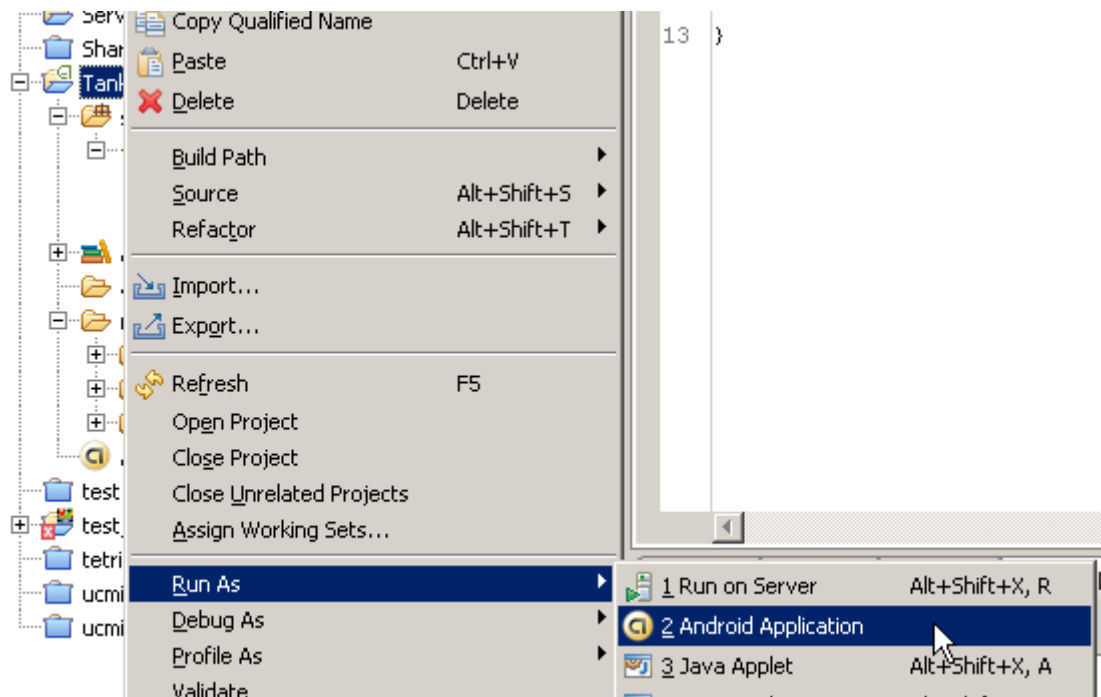
下面需要我们输入项目的一些信息，因为我们要复刻经典游戏坦克大战，所以我们的程序就取名 **Tank**



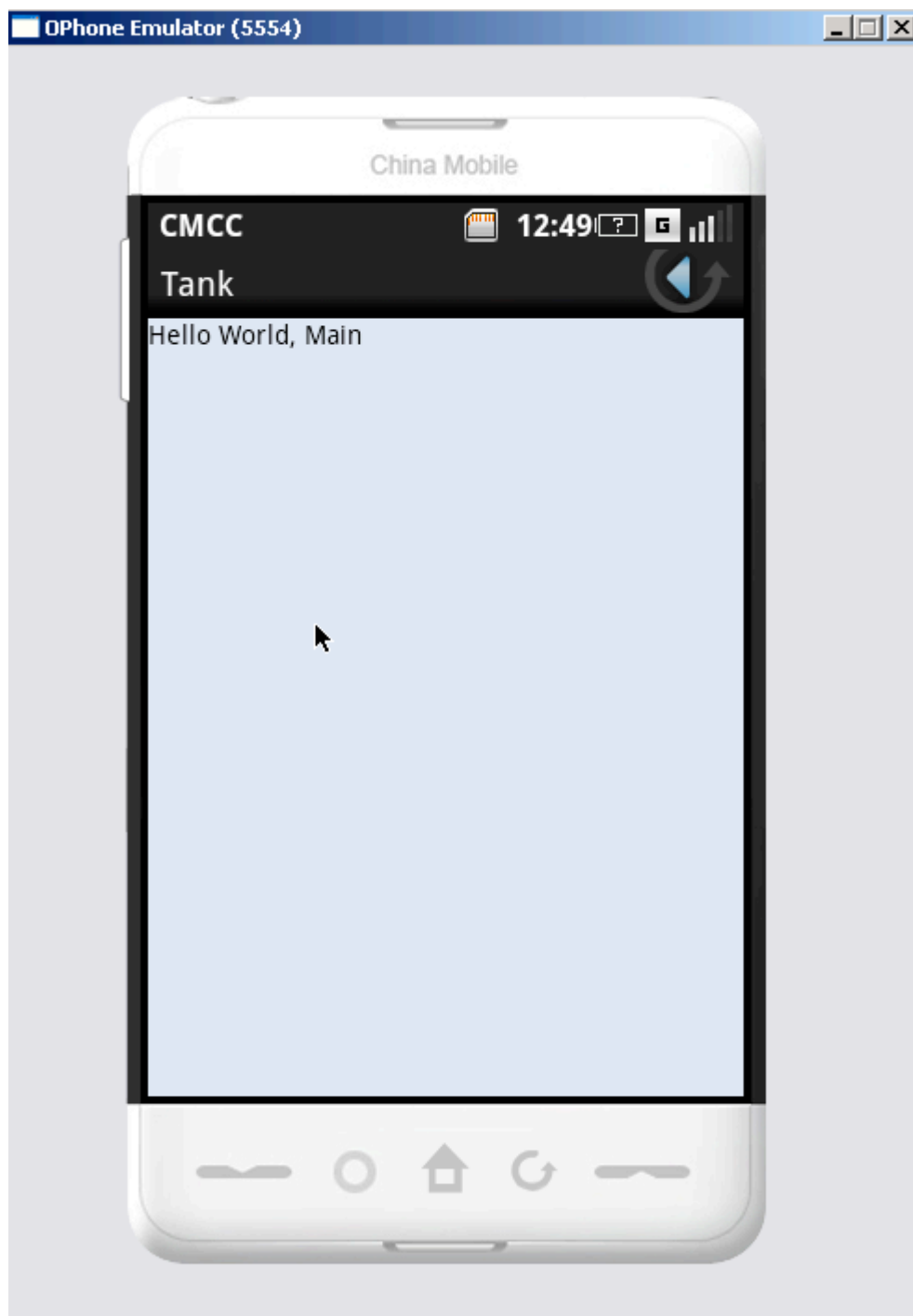
这样，一个 Android 项目就创建完成了，我们可以在 eclipse 的 Package Explorer 看到我们的项目



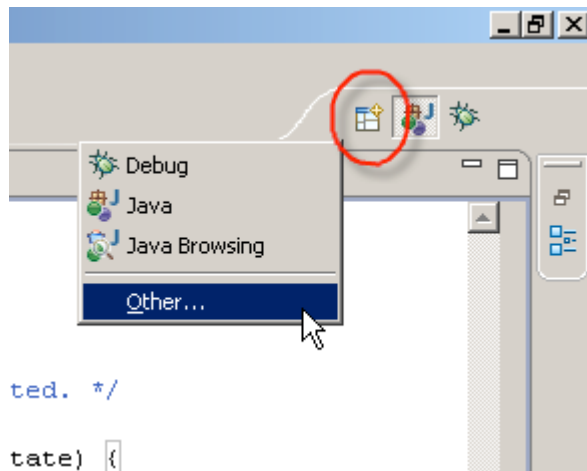
托 ADT 的福，虽然我们只输入了几个名字，但这个项目实际上已经可以运行了。右击项目名称，选择 Run As -> Android Application



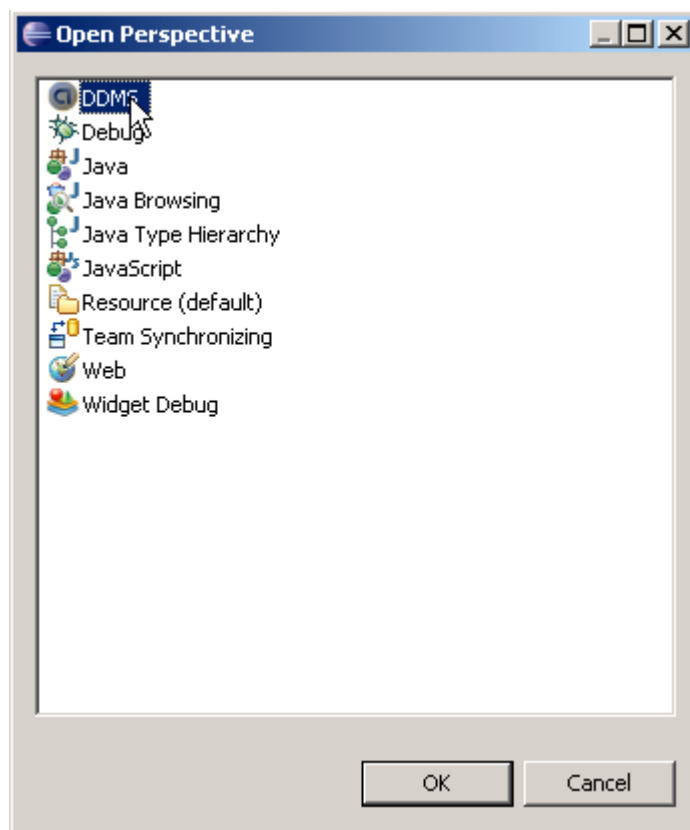
不出意外的话，你会看到一个手机模拟器被启动，而我们刚刚建立的程序会被运行起来



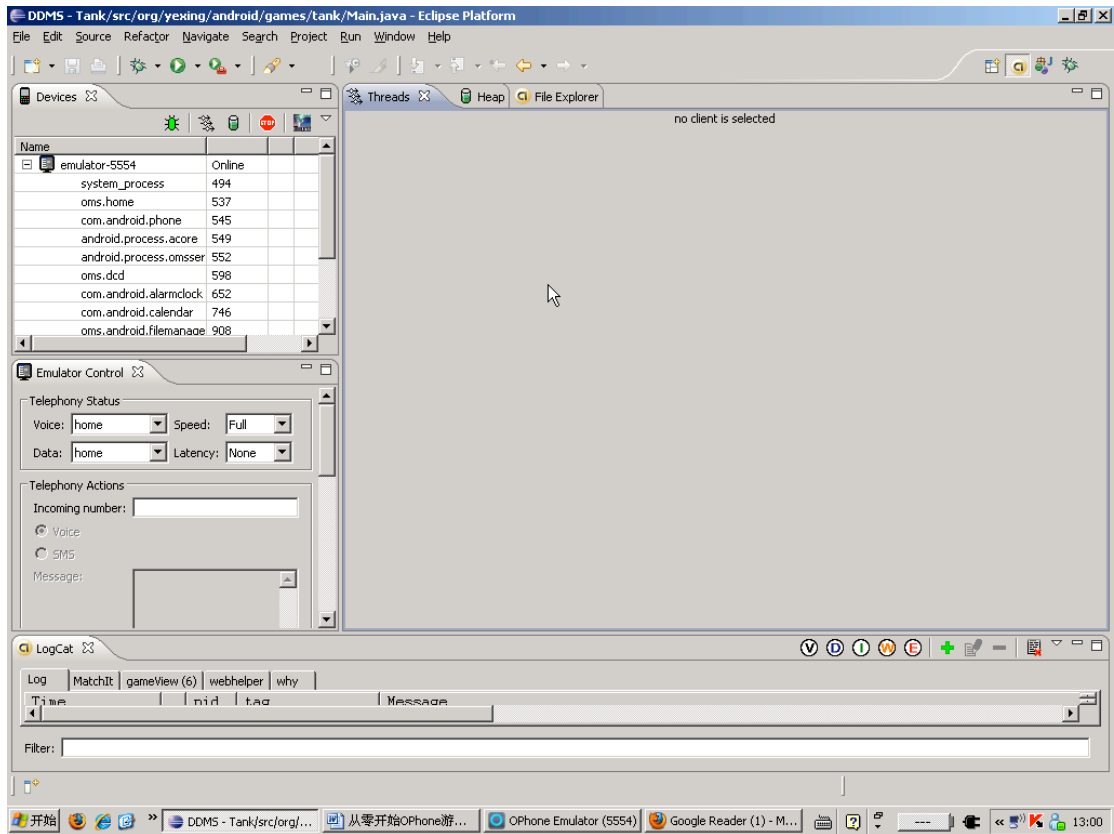
如果你发现模拟器启动了，而程序并没有被运行，可能需要手工启动程序。这里我们用到一个重要的工具 DDMS (Davlik Debug Manager)。运行 DDMS 快捷方法是点击 eclipse 右上角的 Open Perspective，如果在弹出的列表中没有 DDMS，那么点击 Others



选择 DDMS

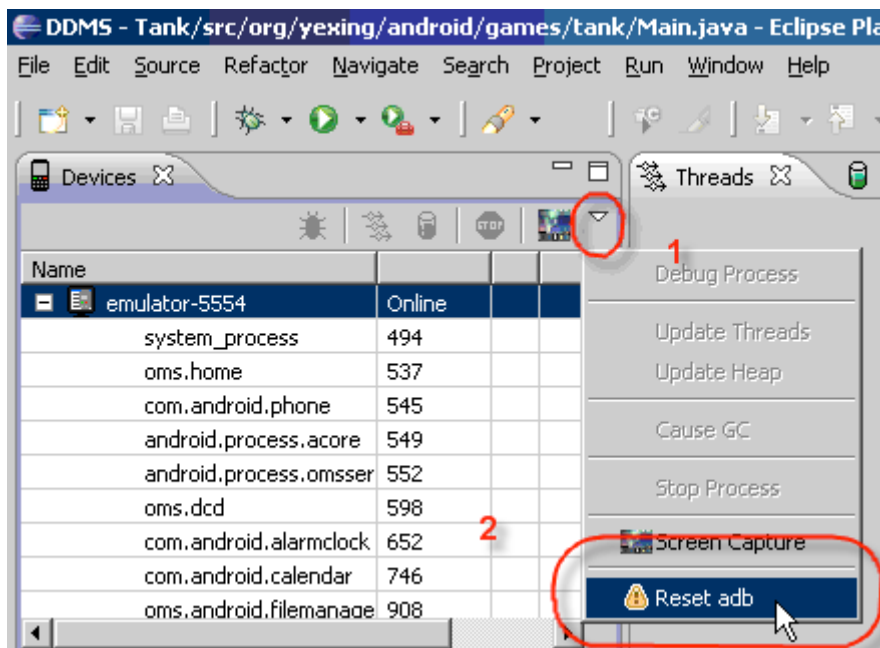


这样我们就打开了 DDMS 界面，这个工具我们会经常用到。



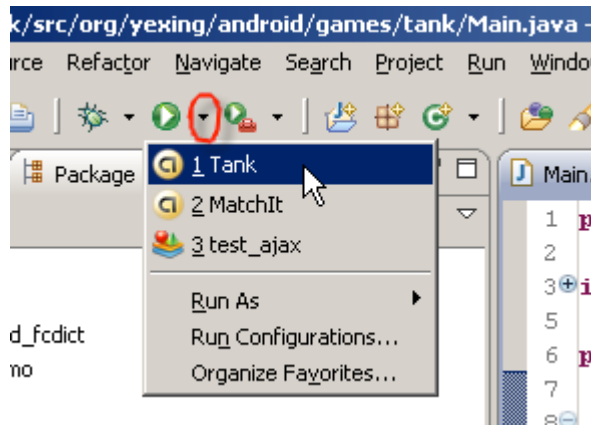
刚刚说到模拟器启动了而程序并没有被运行，很可能是在模拟器启动过程中 DDMS 失去了与模拟器的链接。解决方法很简单：

点击 Devices 标签下的工具栏，选择 Reset adb



然后右击项目名称，Run As -> Android Application。

除了右击运行项目，还可以通过工具栏上的运行按钮启动程序



在运行按钮左边的是 Debug 按钮，这两个我们以后也会经常用到。

现在我们已经有了第一个可以运行的 **Android**，虽然你可能对 **ADT** 生成的一堆文件感到一头雾水，也不知道程序界面上那一句“**Hello World, Main**”是从哪里来的，但是没关系，随着本文的深入你会逐渐熟悉 **Android** 项目的目录结构，程序设计的原则和方法，以及调试和部署的方法。现在读者可以自己熟悉一下模拟器的操作，让我们下章再见。

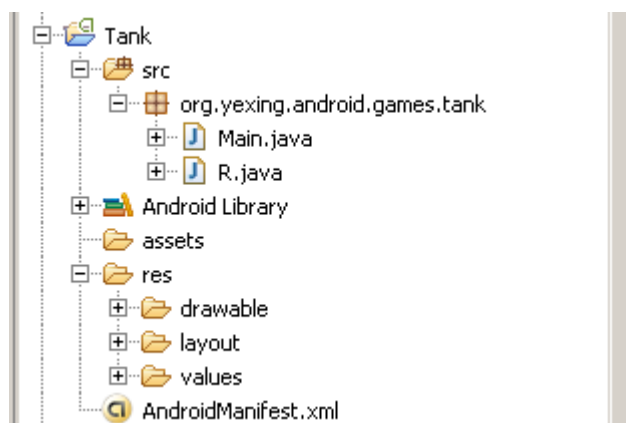
第三章 显示文字和图片

难度：容易

从本章开始，读者就要编写代码了。按照作者的原则——少一些理论，多一些实践，代码中可能会有跳跃的地方。但是请大家不要着急，随着学习的深入，你很快就会了解其中的奥秘。不过在开始之前，我们还是要先来理顺一下思路，看看完成一个坦克大战游戏需要哪些工作：首先，我们需要一个基本的程序，这个程序能够在 **Android** 上运行；这个程序要能够显示图形包括地图，主角和 **NPC** 等等；程序能够接受用户的输入，控制主角移动；程序要能够控制 **NPC** 和子弹的移动；程序还能对各种事件做出判断，比如击中敌人，获得物品，胜利或者失败。

现在我们就从基本程序开始，一步一步实现它。

首先，让我们看一下刚刚生成的文件目录



在源文件目录下，只有 **Main.java** 和 **R.java** 两个文件，刚刚被我们命名成 **Main.java** 的文件就是程序的入口文件。而 **R.java** 是由插件来维护的资源定义文件，我们先不管它。

Main.java 内容如下：

```
package org.yexing.android.games.tank;

import android.app.Activity;

import android.os.Bundle;

public class Main extends Activity {
```

```

    /** Called when the activity is first created. */

    @Override

    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);

    }

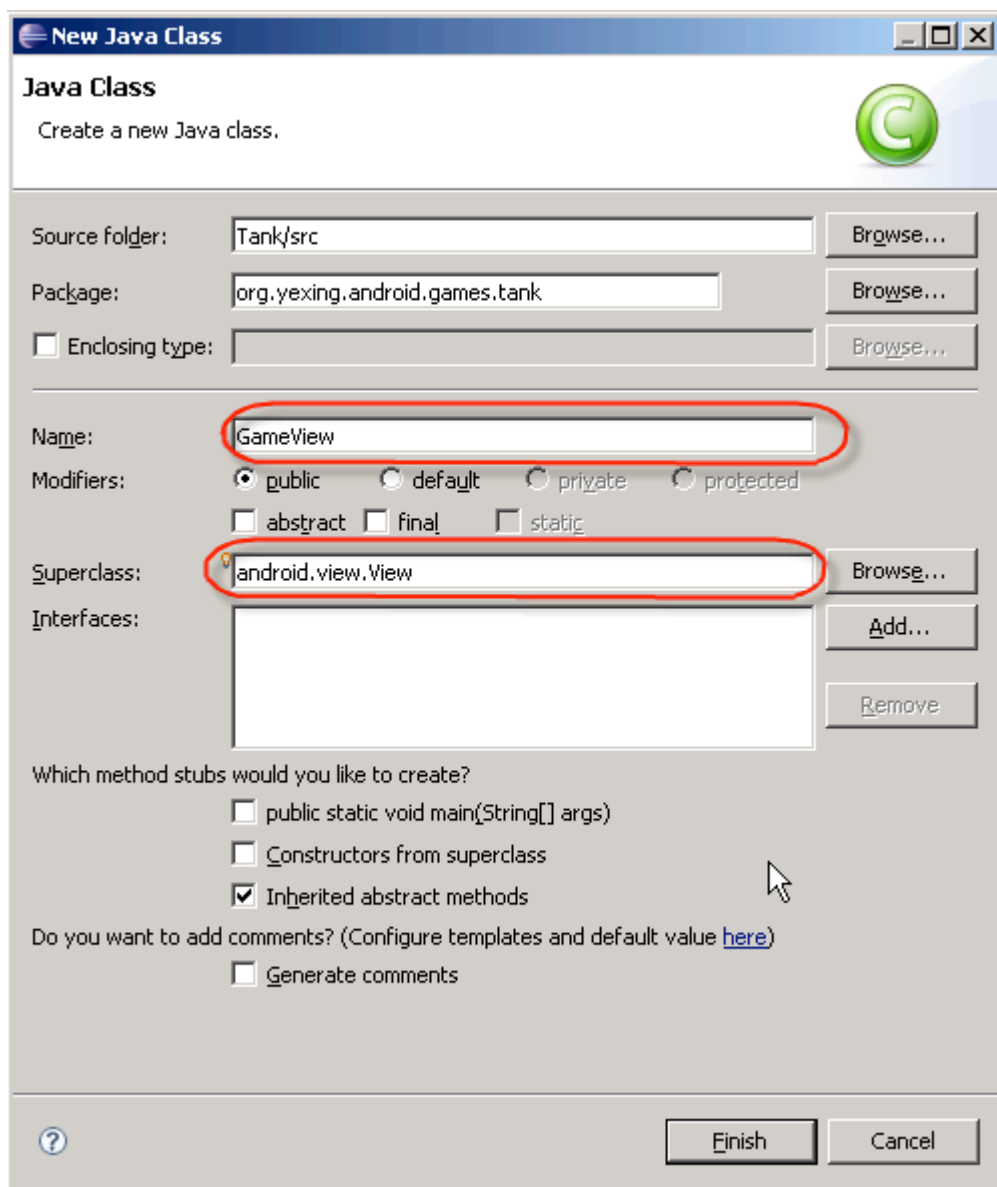
}

```

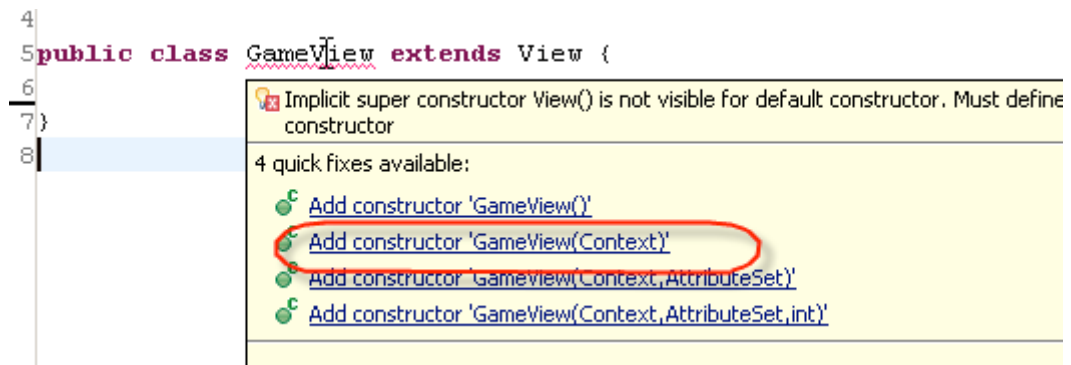
很幸运，Main.java 的代码非常之少，而且还有一段注释，以致我们很容易知道函数 onCreate 的作用，需要解释的只是 setContentView()。先不要管注释中提到的 Activity 和 setContentView 的参数 R.layout.main，我们使用 setContentView 的另一种形式：setContentView(View view)。setContentView 的作用是设定当前使用的视图即 View（依此理解，可以有很多个 View，需要用哪个就可以把他作为 setContentView 的参数显示出来）。View 是一个非常重要的组件，它可以用来显示文字，图片，也可以接收客户的操作，比如触摸屏，键盘等等，而我们的游戏中正是需要绘图和交互，看来 View 很符合我们的需要（但是请注意，使用 View 并不是我们的最终方案，原因会在后面说明。此处介绍 View 是为了讲解基础的图形和用户控制）。

下面我们就要订制一个属于自己的 View，可以通过继承自系统提供的 View，并重载相关的函数来实现。创建类的方法如下：

右击包名 New -> Class



点击 **Finish**，一个 **View** 类就创建好了。这里是第一次创建类，以后就不会有图片演示了，请大家记住的这个方法。**GameView** 创建好了，但是代码还有一些错误，这里介绍一下 **eclipse** 的使用技巧，将鼠标悬停在有错误的位置，或者将光标停在有错误的行，然后按 **Ctrl+1** 键，就会出现修改建议，大部分时候，使用修改建议都可以改正我们的错误，如图



可以看出来，刚刚的错误是因为没有创建构造函数，选择修改建议的第二项，增加一个构造函数

```
public GameView(Context context) {  
    super(context);  
    // TODO Auto-generated constructor stub  
}
```

我们的 View 就创建好了。

回到 Main.java，刚刚说了，只要将 View 作为 setContentView 的参数，这个 View 就可以被显示出来：

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(new GameView(this));  
}
```

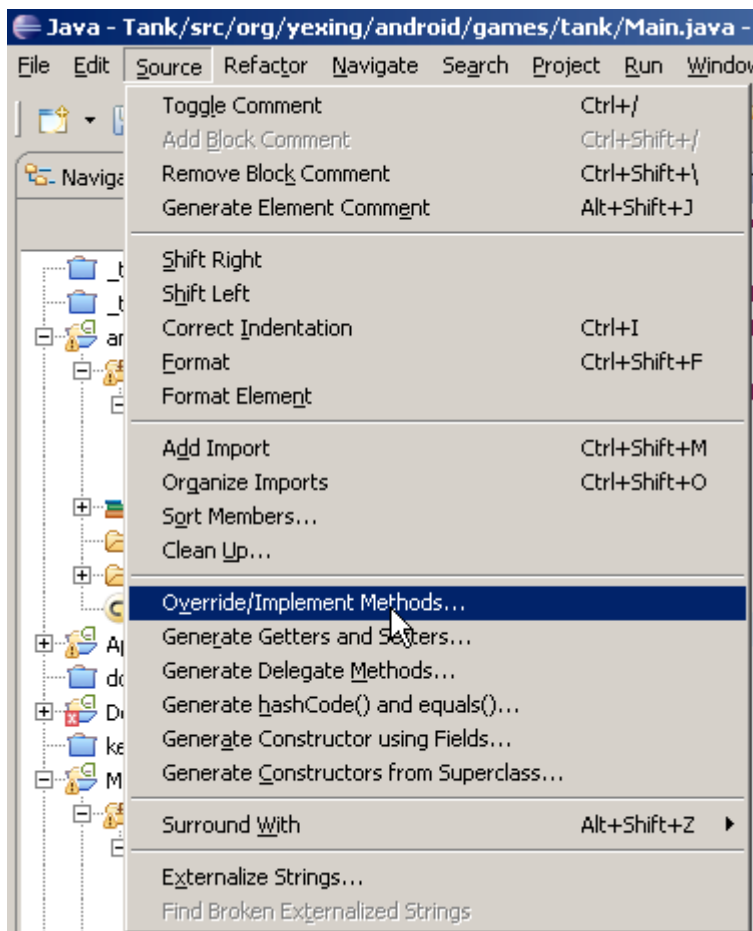
现在让我们运行模拟器，看看程序变成什么样子了（启动模拟器的方法见第二章）。



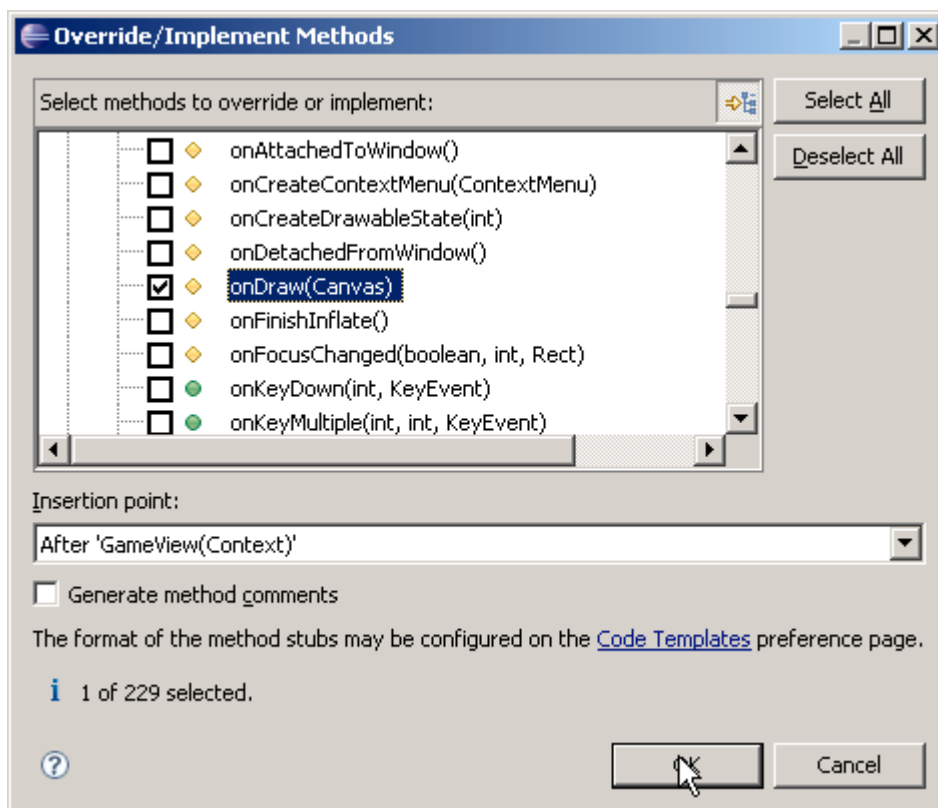
不要意外，屏幕上就是一片空白，因为我们创建了一个 **View**，但是没有让它显示任何内容。

下面我们就会在 **View** 上显示一段文字和一张图片。

让 **View** 显示内容也很简单，只需要重载 **View** 的 **onDraw** 函数，把相应的语句写入 **onDraw** 中即可。打开 **GameView.java**，点击菜单 **Source -> Override/Implement Method...**



选中 onDraw 点击 OK



下面这段代码就会被加入到程序当中，所有与显示有关的代码都会在这里面完成

```
@Override

protected void onDraw(Canvas canvas) {

    // TODO Auto-generated method stub

    super.onDraw(canvas);

}
```

这里我们遇到了又一个非常重要的类 **Canvas**，**Canvas** 一般翻译成画布，所有的绘图操作都是通过 **Canvas** 中的函数来完成的，比如显示文字的函数 **Canvas.drawText()**，显示位图的函数 **Canvas.drawBitmap()**，以及各种绘制图形的函数如 **Canvas.drawRect()**，**Canvas.drawArc()**等等。下面让我们显示一段文字在屏幕上：

```
protected void onDraw(Canvas canvas) {

    // TODO Auto-generated method stub

    super.onDraw(canvas);

    canvas.drawText("坦克大战", 50, 50, new Paint());

}
```



坦克大战四个字已经出现在了屏幕上。让我们来详细看一下这条语句：


```
canvas.drawText("坦克大战", 50, 50, new Paint());
```

第一个参数是要显示的文字，第二、第三个参数是文字在屏幕上的坐标，说到坐标得多讲两句。在 2D 编程中，屏幕坐标的原点是屏幕的左上角，横向向右增大，纵向向下增大，如上图所示。最后一个参数是 **Paint**，通常翻译成画笔，它决定了文字或图形的颜色，字体，线条粗细等等，后面用到相应属性的时候会详细介绍。那么这条语句就是在屏幕上（50，50）的位置用缺省的画笔写出“坦克大战”四个字。另外如果 **eclipse** 提示代码错误，不要忘了用 **Ctrl+1**。

有了文字，下面就是图像了。显示图像比显示文字略微复杂一些，首先我们要准备一张位图，图片必须是 **png** 格式的，文件名只能是小写字母，数字和下划线。



battlecity.png

然后将这张图片 **copy** 到工程的 **res/drawable** 目录下。可以直接在 **eclipse** 的目录树中粘贴。



显示位图的函数是 **Canvas.drawBitmap()**，**drawBitmap** 有很多种形态，我们先看其中最简单的一种

```
canvas.drawBitmap(bitmap, left, top, paint)
```

乍一看似乎和 `drawText` 差不多，4 个参数有三个都相同，但这第一个参数 `bitmap` 要比文本复杂得多。首先，他是一个 `Bitmap` 类实例，因为我们现在还不需要这个类的其他功能，所以不过多介绍 `Bitmap`，只考虑它是怎么来的。得到 `Bitmap` 实例的方法也有很多种，这里只介绍其中的一种

```
BitmapFactory.decodeResource(res, id);
```

此方法可以返回一个 `bitmap` 实例，但是这个函数还需要两个参数 `res` 和 `id`。`res` 是 `Resources` 实例，而 `id` 是一个整数，下面让我们分别了解这两个参数。`res` 的地位跟 `bitmap` 差不多，只需要作为参数被使用，因此，只要得到实例就可以了，获得 `Resources` 实例的方法如下：

```
res = context.getResources();
```

天哪，事情越来越复杂了，因为这段代码里面有多了一个陌生面孔 `context`。`context` 是 `Context` 实例，`Context` 通常翻译做上下文，这个名称似乎有点晦涩，他究竟是什么呢？让我们回头看看写好的程序

```
public GameView(Context context) {  
    super(context);  
    // TODO Auto-generated constructor stub  
}
```

这时候我们有一个 `context` 实例，继续溯源而上，在 `Main.java` 中

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(new GameView(this));  
}
```

原来，`context` 指向 `Main` 类。好了，我们终于找到 `res` 的源头了。还有另外一个分支第二个

参数 id。

```
BitmapFactory.decodeResource(res, id);
```

id是一个整形，它到底是谁的id呢？我们还是得往前面找，还记得我们第一次见到函数 setContentView时什么样子么

```
setContentView(R.layout.main);
```

对，他的参数是R.layout.main，后来被我们替换成了GameView实例。R.layout.main就是一个整数。它被定义在文件R.java中，我们前面讲过R.java是由插件维护的资源定义文件。说到这里大家应该猜到了吧。让我们打开R.java文件

```
public final class R {  
  
    public static final class attr {  
  
    }  
  
    public static final class drawable {  
  
        public static final int battlecity=0x7f020000;  
  
        public static final int icon=0x7f020001;  
  
    }  
  
    public static final class layout {  
  
        public static final int main=0x7f030000;  
  
    }  
  
    public static final class string {  
  
        public static final int app_name=0x7f040001;  
  
        public static final int hello=0x7f040000;  
  
    }  
  
}
```

果然，位图文件 battlecity.png 在这里面也被分配了一个 id: R.drawable.battlecity，没错，

就是它了，这就是我们要找的 `id`。至此为止，我们终于可以使用 `drawBitmap` 了。

对于一次创建，多次使用的资源，我们把他放到构造函数里面。增加了图形显示的 `GameView` 如下：

```
public class GameView extends View {

    Bitmap bmp;

    public GameView(Context context) {

        super(context);

        // TODO Auto-generated constructor stub

        Resources res = context.getResources();

        bmp = BitmapFactory.decodeResource(res,
R.drawable.battlecity);

    }

    @Override

    protected void onDraw(Canvas canvas) {

        // TODO Auto-generated method stub

        super.onDraw(canvas);

        canvas.drawText("坦克大战", 0, 50, new Paint());

        canvas.drawBitmap(bmp, 0, 100, new Paint());

    }

}
```

运行效果



第四章 响应用户事件

上一章介绍了如何显示文字和图片，一般来说，下一步就该讲到动画了。可是我们前面说了，使用 **View** 不是最终的选择，要实现动画还需要很多复杂的代码。相对来说，学习如何响应用户事件要简单些。

本章前半部分讲解按键事件的响应，但是这也不是最终方案，因为实际上的手机可能没有硬键盘，需要使用虚拟键盘，所以后半部分我们会讲解虚拟键盘的设计和实现。

同绘图一样，**View** 也是通过回调函数来响应用户事件的。键盘事件的回调函数有多个，以对应不同的事件，我们暂时只用到 **onKeyDown**，对应按键被按下的事件，其他函数以后用到再介绍。让我们重载 **onKeyDown**（重载一个函数的方法前面章节有介绍）：

```
@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

    // TODO Auto-generated method stub

    return super.onKeyDown(keyCode, event);

}
```

onKeyDown 有两个参数：**keyCode** 和 **event**，通过 **keyCode** 能判断是哪个键被按下，**event** 比较复杂，包含了这次按键更多的信息，我们暂时先不考虑它。

现在我们要通过按键控制主角向四个方向移动。所谓移动，就是将主角的图像在不同的位置显示出来，也就是改变函数 **drawBitmap** 中的第二、第三个参数。比如用户按下右方向键，我们就把横坐标增加，这样下次显示出来的时候，主角就会往右一点。为了节约时间，我们就把刚刚显示的图片 **BattleCity** 作为主角好了。首先定义两个全局变量 **x** 和 **y**，然后在 **onKeyDown** 中改变 **x**、**y** 的值，然后重绘 **View**。因为代码没有什么难度，所以不做讲解了。

```
public class GameView extends View {

    int x=0, y=0;
```

```

.....

@Override

protected void onDraw(Canvas canvas) {

    .....

    canvas.drawBitmap(bmp, x, y, new Paint());

}

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

    // TODO Auto-generated method stub

    switch(keyCode) {

        case KeyEvent.KEYCODE_DPAD_UP:

            y -= 10;

            break;

        case KeyEvent.KEYCODE_DPAD_DOWN:

            y += 10;

            break;

        case KeyEvent.KEYCODE_DPAD_LEFT:

            x -= 10;

            break;

        case KeyEvent.KEYCODE_DPAD_RIGHT:

            x += 10;

            break;

    }

    postInvalidate(); //通知系统重绘 View

    return super.onKeyDown(keyCode, event);

}

}

```

完成后我们肯定很想测试一下，但是此时你会发现，按键根本没有任何反应。这就是我们要特殊指出的地方。**View** 被显示时，缺省情况下没有获得焦点，就是说，按键动作没有发送

给 View，所以需要在构造函数中增加一句

```
public GameView(Context context) {  
    .....  
    setFocusable(true);  
}
```

再运行程序，看看图片是否按照我们的指令运动起来了。

前面说过，很多手机没有硬键盘，所以我们需要一个软键盘的解决方案。软键盘就是在屏幕上显示一个键盘，然后响应用户的触摸屏操作，模拟成键盘操作。对于坦克大战，我们只需要在屏幕上显示一个模拟的游戏手柄（显示图片的方法大家没有忘记吧，显示位置可以根据模拟器自行调整）：



在用户触摸模拟手柄上的方向键和开火键时进行相应的操作。我们拿方向键做演示，步骤如下：首先确定四个方向键在屏幕上的区域（上图的红色方框），然后在触摸屏事件的响应函

数中判断事件是否发生在方向键区域中，最后如果事件发生在区域中进行相应的操作。

下面，我们引入一个非常有用的类 **Rect** (**RectF** 与 **Rect** 基本相同，不过以 **float** 作为坐标参数)，**rect** 是 **rectangle** 的简写，顾名思义，这个类代表了一个矩形。**Rect** 通过矩形 4 个边来定义这个矩形的范围。他们分别是 **left**, **right**, **top**, **bottom**，如图所示：



转化为屏幕坐标，**top** 是矩形坐上角的纵坐标，**left** 是矩形坐上角的横坐标，**right** 是矩形右下角的横坐标，**bottom** 是右下角的纵坐标。有了 **Rect** 我们就可以方便的表示虚拟手柄各个键的位置。同时 **Rect** 还提供了一些很有用的函数，其中 **Rect.contains(x, y)**能够判断点(x, y)是否在矩形框中，正好是我们需要的。

现在我们可以开始编码了，首先为虚拟键盘的方向键创建 **Rect**（可以用绘图工具测量坐标）：

```
Rect rKeyUp = new Rect(56, 290, 86, 320);  
Rect rKeyDown = new Rect(56, 350, 86, 380);  
Rect rKeyLeft = new Rect(26, 320, 56, 350);  
Rect rKeyRight = new Rect(86, 320, 116, 350);
```

然后重载触摸屏响应函数：

```
@Override  
  
public boolean onTouchEvent(MotionEvent arg0) {  
    // TODO Auto-generated method stub  
    return super.onTouchEvent(arg0);  
}
```

下面我们要做的是，首先判断触摸屏操作是不是按下，如果是，取得坐标 (x, y)，然后判断坐标所在的按键，做出相应的操作

```
@Override

public boolean onTouchEvent(MotionEvent arg0) {

    // TODO Auto-generated method stub

    if (arg0.getAction() == MotionEvent.ACTION_DOWN) {

        int ax = (int) arg0.getX();

        int ay = (int) arg0.getY();

        if (rKeyUp.contains(ax, ay)) {

            y -= 10;

        } else if (rKeyDown.contains(ax, ay)) {

            y += 10;

        } else if (rKeyLeft.contains(ax, ay)) {

            x -= 10;

        } else if (rKeyRight.contains(ax, ay)) {

            x += 10;

        }

        postInvalidate(); //不要忘记刷新屏幕

    }

    return super.onTouchEvent(arg0);

}
```

现在让我们运行一下，每次用鼠标点击模拟手柄的方向键，图片就会移动



至此为止，我们介绍了两种响应用户事件的手段，但是要真正完成对一个游戏的控制，还需要更多的工作，后面还有深入的讲解。

第五章 小结——扫雷游戏的实现

目前，我们学习了如何建立 **Android** 编程环境，如何显示文字和图片，如何响应用户事件。

作为总结，我们要运用这些知识实现一个扫雷游戏。



先说游戏规则：扫雷，就是在一个分成若干小格的矩形区域中发现隐藏的地雷，找到它，但是不能触发它。每次翻开一个小格，如果下面是地雷，游戏就失败了。如果不是地雷，而它的周围 8 个格中有地雷，那么就会显示周围的地雷数。如果周围 8 个格中没有地雷，那就是空白的。如果你认为某一格是地雷可以用红旗标记它，正确标记了所有的地雷或者翻开了所有不是地雷的格就取得胜利。

再说用户操作：在 **Windows** 中用户可以有三种操作，左键单击，右键单击，左右键同时单击。左键可以翻开一个小格，可以触雷，如果点到了一个空白格，跟它相联的所有空白格都会被打开。右键可以标记一个格，不会触雷，标记方式有两种，第一次单击用红旗标记，确信此处有雷，再次单击红旗变成问号，表示可能有雷。左右键同时单击只在此种情况有效，即一个格周围有雷，并且所有的雷都已被用红旗标记出来。此时单击此格会打开周围所有未标记的格，此操作会触雷，就是说如果标记错了游戏就会失败，所以一定要小心使用。因为

在手机上没有右键，所以我们必须设计替代方案，一种方法是设计一个开关图标，点击打开开关后，所有的操作都被认为是右键和左右键同时单击，另外还可以借助手机上的菜单键，按住菜单键等同打开开关，松开等同关闭开关。为了方便游戏，我们可以同时实现两个方案。

最后让我们分析一下程序的大体思路，从最直观的用户界面开始，我们需要根据游戏的显示区域创建游戏地图，一个 m 行 n 列的二维整形数组，数组中的一个值对应界面上的一个格，不同的数值可以表示不同的状态，比如 **0** 表示空白格，**1** 表示此格周围有一个雷，**2** 表示有两个雷等等。因为雷的位置是随机的，所以这张地图需要在每次游戏开始的时候被初始化。另外因为这张地图不能直接显示给用户看，所以我们还需要另一个同样大小的地图把它盖起来。并且也用不同的数值来表示一个格是否被翻开，或者被标记等等。这样每次刷新屏幕我们会根据地图上的数值在屏幕上对应的位置显示不同的图片，就是我们看到的游戏界面。这种使用多层地图的技术在游戏中非常常见。这两个地图可以在同一个二维数组中，也可以分开两个数组，为了便于理解，我们使用两个数组分别表示。另外，界面上还要显示当前剩余的地雷数量和游戏时间。剩余地雷数是地雷总数减去用户标记的地雷数，可以是负值。游戏时间是每次新游戏开始时启动的一个计时器，我们知道，如果要时间连贯显示，就必须不停的自动刷新屏幕，但是回顾前面四章的内容，并没有讲解如何循环刷新屏幕，所以在这里会介绍一种使用 **Handler** 刷新屏幕的方法，这种方法比较简单，但是效率并不高，所以后面我们还会介绍另外一种更有效的方法。

接着来看用户事件的响应，虽然用户也可以用键盘来操作，但那样就失去了扫雷游戏追求速度的快感，所以我们只设计使用触摸屏的方案。当用户点击屏幕时，首先判断点击在哪一格，再根据用户点击的方法，以及被点击格的状态，判断用户操作的结果，改变地图上相应格的数值，刷新后用户操作的结果就会在屏幕上反映出来。

另外我们还需要一些游戏状态的标志，比如游戏胜利，失败或是正在游戏中。如果游戏胜利，还需要存储记录，由于存储操作前面没有讲到，我们暂时放弃这个功能。

为了缩减篇幅，下面使用源代码直接讲解，源程序的 **eclipse** 工程文件已经随本文一起提供下载，这里就算是一个源程序导读（在源程序的 **res/raw** 目录下有一首 **mp3** 很好听哦）。

首先看 **Main.java**

.....

```
@Override

protected void onPause() {

    /*

    * 在程序被挂起或者退出的时候改变游戏状态以结束游戏循环

    */

    gameView.gameState = GameView.STATE_LOST;

    super.onPause();

}

.....
```

这里我们对 `onPause` 做一个说明，也是对 `Android` 程序生命周期的一个简单介绍。详细内容会随着文章的深入慢慢讲解。大家知道在 `pc` 中，多个程序是可以同时运行的，即多进程。在手机中，程序依然可以多进程运行，但是还有一些不同：首先是屏幕，当一个应用启动后，他要独占屏幕。这样一些有打断功能程序运行起来后，当前的程序就不可见了，最简单的例子就是来电。当你正在游戏，突然有电话打进来，来电程序会占领屏幕，你的游戏转到后台运行（这时 `onPause` 事件就会被触发）；另一个不同的地方是，如果一个程序被转到后台太长时间而没有再次被激活，那么系统会结束这个程序。在结束之前会触发相应的事件（`onSaveInstanceState`，后面会介绍）。而与之对应的，当程序开始，触发 `onCreate` 事件时，我们需要检测当前程序是第一次运行还是被在后台销毁后重新运行，如果是重新运行，我们需要装载程序销毁前的状态信息。

这里我们用 `onPause` 的方法并不正确，因为程序挂起的时候不应该让游戏结束，但是为了简化代码，我们暂时先这样做，后面会加以改进。

程序的主要内容在 `GameView.java` 中，随本章提供的源代码中有详细的注释，请大家自行阅读源码。

本章示例程序 <http://u.115.com/file/f1e65e86c1>

第六章 SurfaceView 动画

难度：中等

前面介绍的内容，还是比较简单的，应用这些知识，可以完成一些非实时游戏，比如井字棋等，或者一些画面刷新不是很频繁、实时性不强的游戏，比如我们前面做的扫雷。但是我们的目标是坦克大战，对操作的实时性要求比较高，更有很多的 NPC 需要处理，绘图的工作量也很大，所以我们要用一个新的视图类 **SurfaceView** 代替 **View** 来完成显示工作。**SurfaceView** 与 **View** 有一些不同，但是我们只用其中的一个特性：在主线程之外的线程中向屏幕上绘图。这样就可以避免在画图任务繁重的时候造成主线程阻塞，从而提高程序的反应速度。

首先让我们重新定义一个 **GameView** 类，让他继承自 **SurfaceView**，并且要实现 **SurfaceHolder.Callback** 接口。为什么要实现 **Callback** 接口呢？因为使用 **SurfaceView** 有一个原则，所有的绘图工作必须得在 **Surface** 被创建之后才能开始（**Surface**—表面，这个概念在图形编程中常常被提到。基本上我们可以把它当作显存的一个映射，写入到 **Surface** 的内容可以被直接复制到显存从而显示出来，这使得显示速度会非常快），而在 **Surface** 被销毁之前必须结束。所以 **Callback** 中的 **surfaceCreated** 和 **surfaceDestroyed** 就成了绘图处理代码的边界。我们直接让 **GameView** 类实现 **Callback** 接口，使程序更简洁一些。

GameView 被创建，并补充了构造函数之后就是这个样子（创建类和添加构造函数的方法前面有介绍哦）

```
package org.yexing.android.games.tank;

import android.content.Context;

import android.view.SurfaceHolder;

import android.view.SurfaceView;

import android.view.SurfaceHolder.Callback;
```

```

public class GameView extends SurfaceView implements Callback {

    public GameView(Context context) {

        super(context);

        // TODO Auto-generated constructor stub

    }

    public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2,
int arg3) {

        // TODO Auto-generated method stub

    }

    public void surfaceCreated(SurfaceHolder arg0) {

        // TODO Auto-generated method stub

    }

    public void surfaceDestroyed(SurfaceHolder arg0) {

        // TODO Auto-generated method stub

    }

}

```

这里我们有看到了一个新的类 **SurfaceHolder**，我们权且把它当作一个 **Surface** 的控制器，用它来操作 **Surface**。因为我们现在还不需要直接操作 **Surface**，所以我们不做深入讲解。而唯一要使用的是 **SurfaceHolder.addCallback**，即为 **SurfaceHolder** 添加回调函数。原因前面我已经说明了，方法如下：

```

public GameView(Context context) {

```

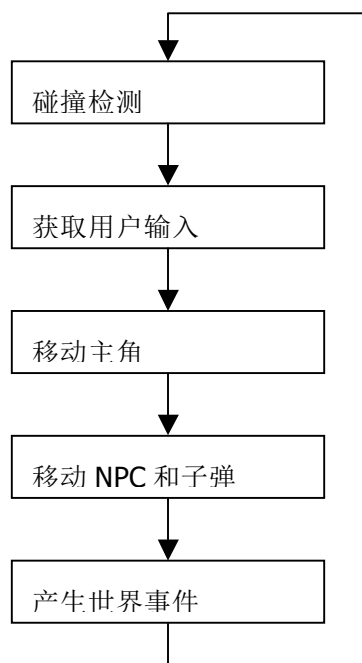


```
super(context);  
  
// TODO Auto-generated constructor stub  
  
getHolder().addCallback(this);  
  
}
```

现在我们可以运行一下，跟第一次使用 **View** 一样，界面上什么也没有。因为我们还没有编写绘图的代码嘛。

前面说过，我们之所以使用 **SurfaceView** 代替 **View**，是因为 **SurfaceView** 可以在主线程之外的线程中进行绘图操作，从而提高界面的反应速度。下面我们要做的就是创建一个用来绘图的线程。不过在这之前我们可以先了解一些关于游戏循环的知识：

我们知道，一般的应用程序是用户驱动的，就是用户操作了，程序再来响应。而我们的游戏呢，不管用户有没有操作，都会有一些变化，最明显的就是 **npc** 会移动、发生世界事件等。因此，我们可以说，游戏程序在一个无限循环当中，我们就把它叫做游戏循环。那么在游戏循环中要做哪些工作呢？让我们用一个流程图来说明游戏循环的过程：



这只是我们假设的流程，不同的游戏肯定会都有些变化。而且细节上会有更多的差别。

了解了游戏循环，下面的工作就是建立一个线程，线程中包含一个游戏循环，在游戏循环中更新游戏的各种数据，并根据这些数据将游戏画面绘制在 **Surface** 上最终显示给玩家。

创建线程的方法很简单，我们不需要知道 **Thread** 的很多高级特性。只需要知道，在线程中完成具体的工作需要重载 **run()** 函数。线程通过 **start()** 函数启动。然后就会执行 **run()** 函数中的内容，**run()** 函数执行结束后线程就会终止。因此我们将游戏循环放在 **run()** 函数中。通过 **start()** 启动循环，并通过适当的方式结束循环进而结束整个线程。还要注意一点，所有对 **Surface** 的操作都必须要保证同步，因此我们会使用 **Synchronized** 关键字，同步 **SurfaceHolder**。

增加了 **GameThread** 后的代码如下：

```
public class GameView extends SurfaceView implements Callback {

    public static final String tag = "GameView";

    //声明GameThread类实例
    GameThread gameThread;

    public GameView(Context context) {

        super(context);

        // TODO Auto-generated constructor stub

        //获取SurfaceHolder
        SurfaceHolder surfaceHolder = getHolder();

        //添加回调对象
        surfaceHolder.addCallback(this);

        //创建GameThread类实例
        gameThread = new GameThread(surfaceHolder);

    }

    public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2,
```

```

int arg3) {

    // TODO Auto-generated method stub

    Log.v(tag, "surfaceChanged");

}

public void surfaceCreated(SurfaceHolder arg0) {

    // TODO Auto-generated method stub

    Log.v(tag, "surfaceCreated");

    //启动gameThread

    gameThread.start();

}

public void surfaceDestroyed(SurfaceHolder arg0) {

    // TODO Auto-generated method stub

    Log.v(tag, "surfaceDestroyed");

    //通过结束run()函数的方法结束gameThread，详见GameThread类的定义

    gameThread.run = false;

}

/**
 * GameThread的定义
 * @author xingye
 *
 */

class GameThread extends Thread {

    SurfaceHolder surfaceHolder;

    //run()函数中控制循环的参数。

    boolean run = true;

```

```

public GameThread(SurfaceHolder surfaceHolder) {

    this.surfaceHolder = surfaceHolder;

}

@Override

public void run() {

    // TODO Auto-generated method stub

    int i = 0;

    while(run) {

        Log.v(tag, "GameThread");

        Canvas c = null;

        try {

            synchronized (surfaceHolder) {

                //我们在屏幕上显示一个计数器，每隔1秒钟刷新一次

                c = surfaceHolder.lockCanvas();

                c.drawARGB(255, 255, 255, 255);

                c.drawText("" + i++, 100, 100, new Paint());

                Thread.sleep(1000);

            }

        } catch (Exception e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        } finally {

            if (c != null) {

                surfaceHolder.unlockCanvasAndPost(c);

            }

        }

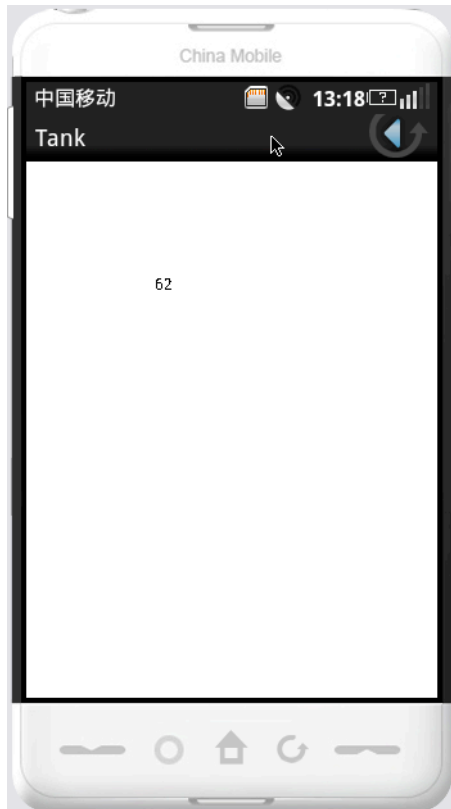
    }

}

```

```
}  
  
}
```

运行程序看一下效果



第七章 精灵、帧动画与碰撞检测

经过前几章的学习，大家对使用位图、接受用户控制应该已经有了初步的概念，也可以运用这些知识完成简单的小游戏。这一章中，我们会为游戏中最重要的部分——图形处理建立一个基本的框架，这还不是游戏引擎，但是其中很多方法可以为读者以后创建自己的游戏引擎提供借鉴。这一章的涉及的内容比较多，既有 2D 游戏的基础理论，又有复杂的代码。尤其是代码部分，如果详细讲解，恐怕会占用很大的篇幅。所以我们只对关键的函数进行讲解，以方便读者今后灵活运用这些代码（所有的源代码都与本章节内容一同提供下载）。

这个框架是完全依照 MIDP 中 `javax.microedition.lcdui.game` 包设计的：

Classes

GameCanvas

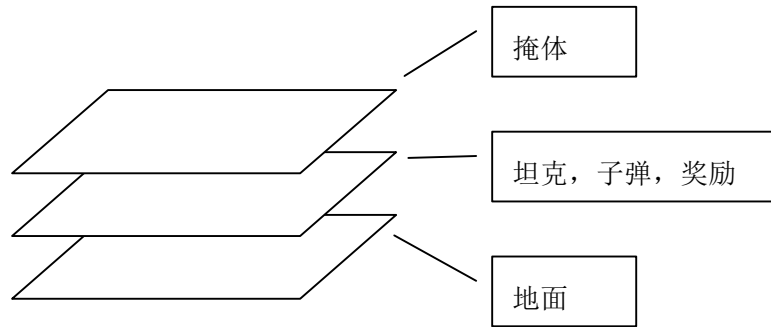
Layer

LayerManager

Sprite

TiledLayer

`game` 包中有 5 个类，其中 **Layer**（层）是一个抽象类，对图形显示作了基本的定义。以我们的目标游戏《坦克大战》为例，在游戏中有这样一些图形元素：我方和敌方的坦克、坦克发出的子弹、地面、墙体、水域掩体等。这些元素虽然外观不同，但是本质上却非常相似：都是在特定位置以特定尺寸显示一个或一组位图，有些位图位置还会变动，**Layer** 就定义了位置，尺寸，显示等相关的功能。之所以叫做 **Layer**，与游戏中**分层地图**的概念有关，先让我们了解一下什么是分层地图：还是说坦克大战，当我们的坦克行驶在普通地面上时，坦克的图像肯定是覆盖了地面的图像，这样我们能看到坦克。当坦克行驶到掩体时，我们会发现，掩体的图像覆盖了坦克的图像，如图所示：



实际上，我们在程序中，只要首先显示地面的图像，然后显示坦克的图像，最后显示掩体的图像（掩体图片是镂空的），就能达到这种效果，这就是分层地图。通常我们把最下面的叫做地面层，中间的叫做物件层，最上面的叫做天空层。关于地图我们就讲这么多，这里只介绍图形意义上的分层，是为了帮助大家理解 **Layer** 一词的意义。关于地图的详细内容我们在第十章会深入讲解。

首先让我们看一下抽象类 **Layer** 的定义：

```
package org.yexing.android.games.common;

import android.graphics.Canvas;

public abstract class Layer {

    int x = 0; // Layer的横坐标

    int y = 0; // Layer的纵坐标

    int width = 0; // Layer的宽度

    int height = 0; // Layer的高度

    boolean visible = true; // Layer是否可见

    Layer(int width, int height) {

        setWidthImpl(width);

        setHeightImpl(height);

    }
```

```
/**
 * 设定Layer的显示位置
 *
 * @param x
 *         横坐标
 * @param y
 *         纵坐标
 */
public void setPosition(int x, int y) {

    this.x = x;

    this.y = y;
}
```

```
/**
 * 相对于当前的位置移动Layer
 *
 * @param dx
 *         横坐标变化量
 * @param dy
 *         纵坐标变化量
 */
public void move(int dx, int dy) {

    x += dx;

    y += dy;
}
```

```
/**
 * 取得Layer的横坐标
 *
```



```
    * @return 横坐标值
    */

    public final int getX() {

        return x;
    }

    /**
     * 取得Layer的纵坐标
     *
     * @return 纵坐标值
     */

    public final int getY() {

        return y;
    }

    /**
     * 取得Layer的宽度
     *
     * @return 宽度值
     */

    public final int getWidth() {

        return width;
    }

    /**
     * 取得Layer的高度
     *
     * @return 高度值
     */

    public final int getHeight() {
```

```

        return height;
    }

    /**
     * 设置Layer是否可见
     *
     * @param visible
     *         true Layer可见, false Layer不可见
     */
    public void setVisible(boolean visible) {
        this.visible = visible;
    }

    /**
     * 检测Layer是否可见
     *
     * @return true Layer可见, false Layer不可见
     */
    public final boolean isVisible() {
        return visible;
    }

    /**
     * 绘制Layer, 必须被重载
     *
     * @param c
     */
    public abstract void paint(Canvas c);

    /**

```

```

    * 设置Layer的宽度
    *
    * @param width
    */
    void setWidthImpl(int width) {
        if (width < 0) {
            throw new IllegalArgumentException();
        }
        this.width = width;
    }

    /**
    * 设置Layer的高度
    *
    * @param height
    */
    void setHeightImpl(int height) {
        if (height < 0) {
            throw new IllegalArgumentException();
        }
        this.height = height;
    }
}

```

Layer 的代码不多，根据函数名称就可以知道它的功能，主要是 Layer 尺寸、位置的设定和获取。其中最重要的方法 `paint` 是虚方法，Layer 图像就是通过这个方法显示出来的。因此继承自 Layer 的所有类都要实现这个方法。

Sprite（精灵）继承自 Layer，同时又增加了帧动画，图形变换和碰撞检测的功能。Sprite 是我们这一章重点介绍的内容。首先让我们了解一下**精灵**的概念。Sprite 这个词在 2D 游戏

中非常常见，一般指游戏中具有独立外观和属性的个体元素。如主角、NPC、宝箱、子弹等等，这些都是精灵。

下面就让我们来创建 **Sprite** 类并使其继承自 **Layer**。创建完毕时，IDE 会提示必须实现 **paint** 方法。但是这时候我们会发现，**paint** 方法要显示那些图形呢？没有。因此我们需要为 **Sprite** 增加一个 **Bitmap** 类型变量，用来存放 **paint** 要显示的图形。同时，我们要创建一个构造函数用来初始化这个 **Bitmap** 变量。

```
public Sprite(Bitmap image) {  
    super(image.getWidth(), image.getHeight());  
  
    initializeFrames(image, image.getWidth(), image.getHeight(),  
false);  
  
    initCollisionRectBounds();  
  
    this.setTransformImpl(TRANS_NONE);  
}
```

虽然这个构造函数只有几行，却涉及到不少的知识。**super** 不用说了，**initializeFrames** 是做什么的呢？这就要提到**帧动画**的概念了。什么是帧动画呢？如下图，我们看到一组星星的图片（4 张 16x16 的位图）



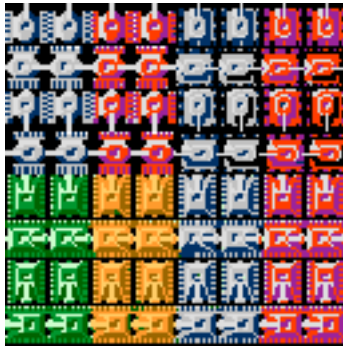
当我们在同一个位置以一定的时间间隔连续显示这几幅图片的时候就变成了这个样子



我们看到，星星在发光，这就是帧动画。即取得一个连续画面中的几个关键帧，在一定的时间间隔下连续的显示这些帧从而形成动画，**initializeFrames** 的功能就是初始化这些关键帧。那么又为什么要初始化呢？通常情况下，我们为了节省空间也为了便于管理，会将一组动画的多个帧保存在同一个图片文件中（如上图的星星）。这样一来每次显示的时候就不能显示整张图片，而只能显示这个图片的一部分。因此，我们要计算每一帧在整张图片上的位置以便正确显示。就让我们来看看 **initializeFrames** 的定义

```
private void initializeFrames(Bitmap image, int fWidth, int fHeight,  
    boolean maintainCurFrame)
```

`initializeFrames` 作了这样的工作，首先取得一个位图，然后根据用户设置的单一帧的宽度和高度计算这个位图中包括多少帧。如刚刚我们看到的星星的图片(64x16 像素)，当单一帧的宽度和高度与图片相同的时候，就只有一帧。但是当一帧的宽度和高度均为 16 像素时，整个图片就可以分为 4 帧了。这时候，函数会计算每一帧的顶点坐标，如第一帧的顶点是 (0, 0)，第二帧的顶点是 (16, 0)，并依次类推。这个函数还可以处理更复杂的情况，如下图：



函数会将计算好的各个帧的顶点横纵坐标分别保存在两个数组中 (`frameCoordsX[]`, `frameCoordsY[]`)，下次我们使用帧的序号访问各个帧时 (在 `paint` 中) 就可以很快找到它的所对应的位图区域了。

在这个 `Sprite` 的构造函数中, `initializeFrames` 使用了 `image.getWidth()`和 `image.getHeight()` 作为一帧的高度和宽度，所以这个精灵注定只能有一帧。`Sprite` 的构造函数还有其他样式，如

```
public Sprite(Bitmap image, int frameWidth, int frameHeight)
```

这时候我们就可以指定帧的宽度和高度，定义具有多个帧的 `Sprite` 了。

说完了为帧动画做初始化工作的 `initializeFrames`，让我们来看下一个函数 `initCollisionRectBounds`。

```
private void initCollisionRectBounds() {  
  
    collisionRectX = 0;  
  
    collisionRectY = 0;
```

```

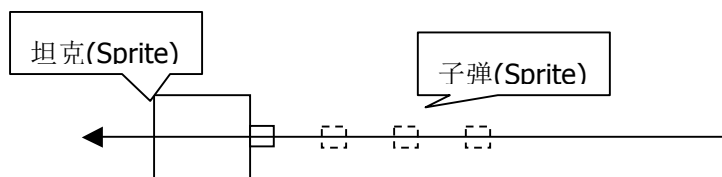
        collisionRectWidth = this.width;

        collisionRectHeight = this.height;

    }

```

这个函数的代码不多，名字翻译过来就是“初始化碰撞矩形边缘”。由此引出另外一个游戏中的重要概念——碰撞检测。在我们的目标游戏坦克大战中，碰撞检测可是少不了的，我们的子弹击中敌方坦克就是一次碰撞，只有进行了碰撞检测才能够触发这次击中事件，不然我们就没法消灭敌人的坦克了。2D 游戏中的碰撞检测有几种，最简单的是矩形碰撞检测，复杂一些的有多边形检测和像素检测等。这里我们只介绍一下矩形检测。如图：



我们取坦克和子弹的矩形外框，当这两个矩形重叠的时候就认为是碰撞了。函数 `initCollisionRectBounds` 的功能就是设定 `Sprite` 的矩形外框。同前面初始化帧的原理一样，我们需要这个函数在多个帧组合成的图片中确定一帧的大小。

现在我们还剩下构造函数中最后一行代码了：

```

this.setTransformImpl(TRANS_NONE);

```

这行代码设置了 `Sprite` 图形的变换方式。变换一共有 8 种，定义如下：

```

public static final int TRANS_NONE = 0;

public static final int TRANS_ROT90 = 5;

public static final int TRANS_ROT180 = 3;

public static final int TRANS_ROT270 = 6;

public static final int TRANS_MIRROR = 2;

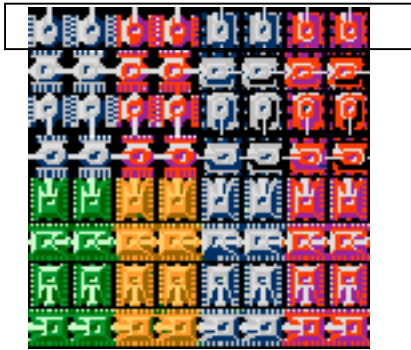
public static final int TRANS_MIRROR_ROT90 = 7;

public static final int TRANS_MIRROR_ROT180 = 1;

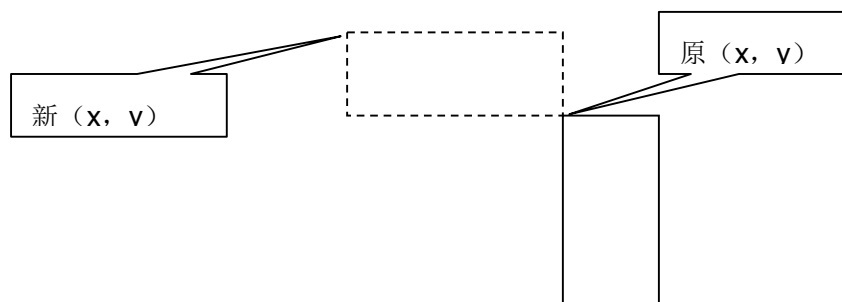
public static final int TRANS_MIRROR_ROT270 = 4;

```

所谓变换，就是对图形进行旋转和镜像等操作，这就相当于增加了图形资源。如图：



因为要显示向 4 个方向行驶的坦克，每个坦克都需要 4 组图片。如果我们使用了旋转变换功能，每个坦克只需要一组图片就够了，其他的图片完全可以由旋转获得。需要指出的是，旋转是围绕参照点（reference pixel）进行的，如果没有使用函数 `setRefPixelPosition` 设定参照点，缺省情况下参照点就是(0,0)。因此如果我们使用参数 `TRANS_ROT90` 旋转的话，图形应该是这样



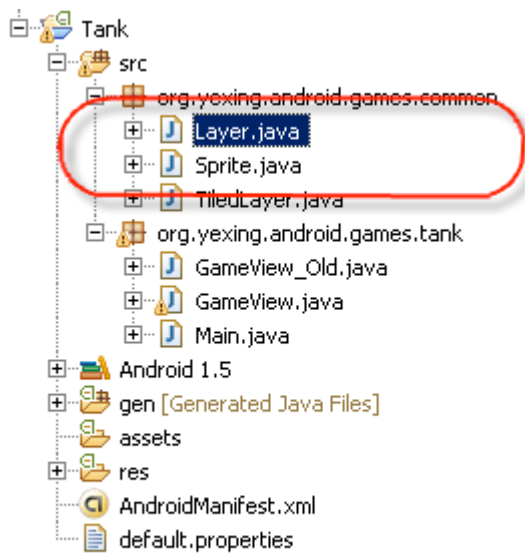
这时候有一点必须要特别提示一下，旋转之后，`getX` 和 `getY` 的返回值将发生变化。

讲到这里，这一章的理论实在是够多了，我们必须要总结一下：

首先这一章讲的是图形显示。我们依照 `j2me` 中的 `games` 包建立了两个类 `Layer` 和 `Sprite`。重点介绍了 `Sprite`（精灵）类相关的知识，包括帧动画、碰撞检测和旋转变换。下面我们来看一组 `Sprite` 的应用实例：

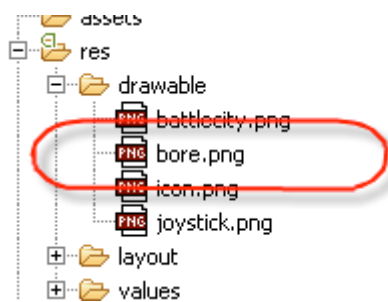
第一个例子，在屏幕上显示一个带有帧动画的 `Sprite`。

让我们拿出前面做过的 `Tank` 的源代码，将 `Layer.java` 和 `Sprite.java` 添加到源代码中。



（其中的 TiledLayer 我们在讲解地图的时候再详细介绍）

将图片 bore.png 拷贝到图形资源目录下



打开 GameView.java，在其中添加一个 Sprite 类型变量 s，并在构造函数中初始化 s

```
//取得系统资源

Resources res = context.getResources();

//获取位图

Bitmap bmpBore = BitmapFactory.decodeResource(res,
R.drawable.bore);

//创建一个Sprite对象，使用位图bmpBore，帧的宽度和高度都为16像素

s = new Sprite(bmpBore, 16, 16);

//显示在150, 150位置

s.setPosition(150, 150);
```

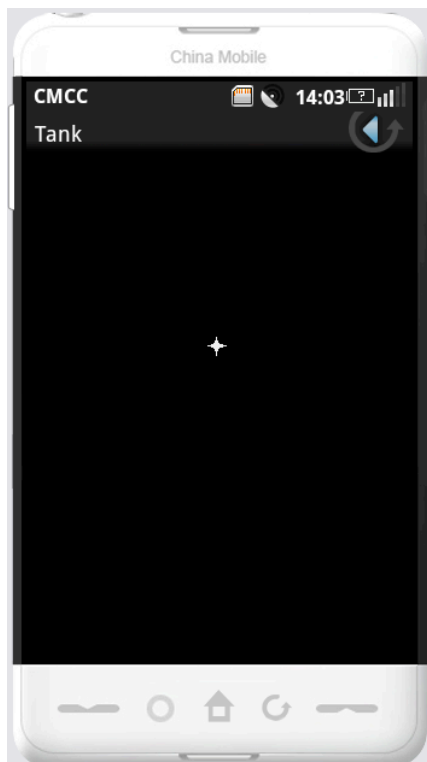


```
//显示第0帧，第1帧和第3帧  
s.setFrameSequence(new int[]{0,1,3});
```

在 `GameThread` 的 `run` 函数中显示这个 `Sprite`

```
synchronized (surfaceHolder) {  
    c = surfaceHolder.lockCanvas();  
    //显示精灵  
    s.paint(c);  
    //显示下一帧  
    s.nextFrame();  
    Thread.sleep(200);  
}
```

运行程序，我们可以看到一个小星星在屏幕上闪烁



第二个例子，`Sprite` 的旋转变换。

依旧使用上一个程序，因为星星旋转起来根本没法分辨，所以这次我们使用系统为程序提供的图标文件 `icon.png`。

首先我们定义一个 **Sprite** 数组

```
Sprite[] ss = new Sprite[8];
```

并在构造函数中初始化这个数组

```
        Bitmap bmpIcon = BitmapFactory.decodeResource(res,  
R.drawable.icon);  
  
        //创建Sprite, 并设置为不同的旋转方式  
  
        for(int i=0; i<8; i++){  
  
            ss[i] = new Sprite(bmpIcon);  
  
            ss[i].setTransform(i);  
  
        }
```

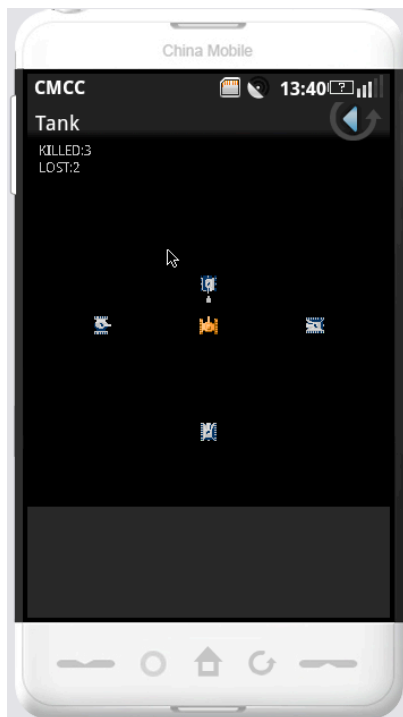
然后在 **run** 函数中显示出来

```
        for(int i=0; i<8; i++) {  
  
            ss[i].setPosition(100, i*40);  
  
            ss[i].paint(c);  
  
        }  
  
        Thread.sleep(200);
```




运行一下看看效果



最后让我们来做一下碰撞检测的例子，这个例子要稍稍复杂一些。先说一下设计思路：



这是一个小游戏，在一个 320x320 的区域中，我方坦克在最中间。从上下左右 4 个方向有敌人坦克向中间进攻。我方坦克可以向四个方向发射子弹，使用方向键改变方向，使用空格

键发射，但同时最多只能发射两颗子弹。敌人坦克有三种类型，其中是普通坦克，的速度是普通坦克的一倍，速度与普通坦克相同，但是需要击中两次才能被摧毁。

首先我们需要引入几个图形文件



分别为子弹、敌方坦克、爆炸效果和我方坦克。

在 `GameView` 中声明变量

```
// 背景色

Paint p = new Paint();

// 文字

Paint pntText = new Paint();

Sprite player; // 我方坦克

.....
```

在 `GameView` 的构造函数中做一些初始化工作

```
// 初始化我方坦克

player = new Sprite(BitmapFactory.decodeResource(res,

        R.drawable.player1), 16, 16);

player.setFrameSequence(new int[] { 0, 1 });

.....
```

增加按键响应事件

```
public boolean onKeyDown(int keyCode, KeyEvent event)
```

最后就是在 **GameThread** 的 **run** 函数中完成游戏逻辑了。具体内容请看本章的代码，代码中有详细的注释。

本章示例程序 <http://u.115.com/file/f1fd539783>

第八章 地图的设计和实现

这本来是第十章，前面计划还有两章的内容，一是跟第四章一样，完成一个 **Asteroid** 游戏作为小结，总结一下前面讲过的 **Sprite** 的用法，并演示 **NPC** 和子弹的处理方法。但是，在写第七章的最后一个例子的时候，把本来简单的碰撞检测的例子扩充了一下，加入了 **NPC** 和子弹，基本和 **Asteroid** 的功能差不多了，所以就把原定的 **Asteroid** 砍掉了。另外一章是讲解程序的生命周期，内容相对简单，但考虑到上一章讲 **Sprite** 时已经引入了 **TiledLayer**，如果接着讲地图应该会更连贯些，所以把生命周期向后移了一章。

那么，就先让我们看看地图的设计和实现。

如果我们需要的地图很小又很少，完全可以将整个地图画在一张图片上。但是如果地图很多，绘制和管理地图的工作就会很麻烦，这时我们就需要用到另外一种技术——图块（**Tile**）。所谓 **Tile**，就是将地图中的公共元素提取出来，然后在显示的时候组合这些元素形成完整的地图，这就是本章要介绍的主要内容。

如下图是组成坦克大战地图的所有元素（16x16 像素）：



接下来让我们看一幅游戏中的场景：



可以看到，整个游戏场景就是由上面那些 **Tile** 构成的。这样，摆在我们面前的任务就很简单了：将地图依照 **Tile** 的大小分成若干格，将对应的 **Tile** 填到格子中。

在前面讲 **Sprite** 的时候，我们知道可以将组合在一张图片中的关键帧编号，以后就可以通过编号来使用这个帧（参看上一章帧动画的相关内容），这种方法也同样适用于 **Tile**。而 **2D** 地图很容易让我们想到二维数组。也就是说我们可以将 **Tile** 的编号放到二维数组中，这样我们就可以通过历遍数组元素，像显示 **Sprite** 那样将整张地图一块一块的显示出来。

以上面的那张游戏截图为例，一共 **13** 行 **13** 列，我们可以定义一个 **13x13** 的二维数组（因为地图上有空白区域，所以我们将 **Tile** 的编号从 **1** 开始，用 **0** 表示空白）。

让我们找到 **GameView_Old.java**，增加成员变量 **map[][]**：

```
int[][] map = {  
    {0,0,0,2,0,0,0,2,0,0,0,0,0},  
    {0,1,0,2,0,0,0,1,0,1,0,1,0},  
    {0,1,0,0,0,0,1,1,0,1,2,1,0},  
    {0,0,0,1,0,0,0,0,0,2,0,0,0},  
    {3,0,0,1,0,0,2,0,0,1,3,1,2},  
    {3,3,0,0,0,1,0,0,2,0,3,0,0},  
    {0,1,1,1,3,3,3,2,0,0,3,1,0},  
    {0,0,0,2,3,1,0,1,0,1,0,1,0},  
    {2,1,0,2,0,1,0,1,0,0,0,1,0},  
    {0,1,0,1,0,1,1,1,0,1,2,1,0},  
    {0,1,0,1,0,1,1,1,0,0,0,0,0},  
    {0,1,0,0,0,1,1,1,0,1,0,1,0},  
    {0,1,0,1,0,1,6,1,0,1,1,1,0},  
};
```

在资源中增加 **tile.png**



在构造函数中初始化 `bitmap` 对象

```
res = context.getResources();  
  
bmp = BitmapFactory.decodeResource(res, R.drawable.tile);
```

在 `onDraw` 中绘图

```
//用来显示图块的Rect对象  
  
Rect src = new Rect(0, 0, 0, 16);  
  
Rect dst = new Rect();  
  
for(int i=0; i<13; i++) {  
    for(int j=0; j<13; j++) {  
        //根据Tile的编号得到对应的位置  
  
        src.left = (map[i][j]-1) * 16;  
  
        src.right = src.left + 16;  
  
        //根据地图上的编号计算对应的屏幕位置  
  
        dst.left = j * 16;  
  
        dst.right = dst.left + 16;  
  
        dst.top = i * 16;  
  
        dst.bottom = dst.top + 16;  
  
        canvas.drawBitmap(bmp, src, dst, paint);
```



```

    }

}

```

最后不要忘了修改 Main.java 中的 setContentView

```

    GameView_Old gameView;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        gameView = new GameView_Old(this);

        setContentView(gameView);
    }

```

好了，运行一下程序看看结果：



对比一下原图，除了基地四周的砖块之外两者并无二致，可以说我们的 **Tile** 地图实践基本成功。

现在我们知道了使用 **Tile** 显示地图的原理，实际上，我们不需要每次都都很麻烦的写那么多代码，还记得前面说过的 **TiledLayer** 么？其中早已封装了上述操作。不仅如此，**TiledLayer** 还能显示动态的地图呢。下面就让我们看一看 **TiledLayer** 的基本用法。其实，它与 **Sprite** 的用法非常相似（我们需要将 **TiledLayer.java** 加入到项目中，请使用本章附带程序的 **TiledLayer.java** 文件，上一章的 **TiledLayer** 并不能正确工作）：

这次让我们使用 **GameView**，把刚刚的数组 **map** 拷贝到 **GameView** 中，并声明一个 **TiledLayer** 类型变量

```
//背景  
  
TiledLayer backGround;
```

在构造函数中初始化 **TiledLayer**

```
// 背景图  
  
backGround = new TiledLayer(13, 13,  
BitmapFactory.decodeResource(res,  
R.drawable.tile), 16, 16);
```

TiledLayer 的构造函数有 5 个参数，分别是地图的行列数（是以 **Tile** 为单位的），包含 **Tile** 的 **bitmap** 对象，**Tile** 的宽度和高度。

通过 **setCell** 方法将定义在数组中的 **Tile** 编号传递给 **TiledLayer**。

```
for(int i=0; i<13; i++) {  
    for(int j=0; j<13; j++) {  
        backGround.setCell(i, j, map[i][j]);  
    }  
}
```

最后，只需要在 `run` 函数中调用 `paint` 方法，就可以将 `TiledLayer` 显示出来了。当然，你可以像控制 `Sprite` 那样控制 `TiledLayer` 显示的位置。

```
background.paint(c);
```

让我们看一下运行的效果



下面让我们来学习如何实现动态地图。所谓动态地图跟前面讲到的帧动画是一个道理，就是循环显示几个关键帧。让我们看一下前面 `Tiles` 的图片



我们会发现有两张水域的图片，这就是为动态地图准备的，组合起来之后应该会有如下的效果：



那么，我们如何在 **TiledLayer** 中实现动态地图呢？**TiledLayer** 为我们准备了这样几个函数：
createAnimatedTile：创建动态图块。很多人会迷惑于这个函数的名字，说是创建动态图块，可是创建在哪儿啊？创建出来怎么用呢？只有天知道。其实，这个函数的主要功能也就是为动态图块分配了一个存储结构。你不调用它还会报错，调用了其实也没什么用。
createAnimatedTile 返回一个动态图块的编号，从-1 开始依次递减，第一次调用返回-1，这样就分配了一个编号为-1 的动态图块。第二次调用会返回-2，依次类推。这个返回值一般没有用，因为我们做地图的时候肯定已经确定了动态图块的位置，这个编号早就写到了数组中了。以后你可以通过这个编号来控制相应的图块。函数有一个参数，指定一个 **Tile** 的编号，动态图块最初显示的就是这个 **Tile**。而正是通过改变这个 **Tile** 来实现动画的。请看下面代码：

```
background.createAnimatedTile(4);
```

我们初始化了一个动态图块，编号是-1，参数 4 表示当前显示 **Tile** 序列图中的第四个 **Tile**，就是第一张水域的图片。

setAnimatedTile：动态图块的内容就是使用这个函数改变的。函数的第一个参数是动态图块的编号，如刚刚的-1。第二个参数是 **Tile** 的编号。

说到这里，大家应该清楚动态地图的用法了吧：

首先在地图数组中确定需要显示动态图块的位置，填入相应的编号，例如我们将上一张地图的第三行增加三块水域

```
int[][] map = {  
    {0,0,0,2,0,0,0,2,0,0,0,0,0},  
    {0,1,0,2,0,0,0,1,0,1,0,1,0},  
    {0,1,-1,-1,-1,0,1,1,0,1,2,1,0},  
    {0,0,0,1,0,0,0,0,0,2,0,0,0},  
    {3,0,0,1,0,0,2,0,0,1,3,1,2},  
    {3,3,0,0,0,1,0,0,2,0,3,0,0},  
    {0,1,1,1,3,3,3,2,0,0,3,1,0},  
    {0,0,0,2,3,1,0,1,0,1,0,1,0},
```

```

        {2,1,0,2,0,1,0,1,0,0,0,1,0},
        {0,1,0,1,0,1,1,1,0,1,2,1,0},
        {0,1,0,1,0,1,1,1,0,0,0,0,0},
        {0,1,0,0,0,1,1,1,0,1,0,1,0},
        {0,1,0,1,0,1,6,1,0,1,1,1,0},
    };
};

```

然后在 `GameView` 的构造函数中初始化 `TiledLayer`，除了在 `setCell` 之前调用 `createAnimatedTile` 之外，没有其他区别。

最后就是在 `run` 函数中调用 `setAnimatedTile`，不断地改变图块了

```

    if (background.getAnimatedTile(-1) == 4) {
        background.setAnimatedTile(-1, 5);
    } else {
        background.setAnimatedTile(-1, 4);
    }

    background.paint(c);

```

来让我们看一下运行的效果



其实笔者觉得 `TiledLayer` 完全也可以像 `Sprite` 那样使用帧序列和 `nextFrame` 来实现动态效果，似乎更易用一些，有兴趣的读者可以自己修改 `TiledLayer` 实现这个功能。

到这里，我们已经掌握了显示地图的方法，但是，这个地图还不能真正运用到我们的游戏中。读者肯定也看到了，在 `TiledLayer` 的第一个例子中，我们的坦克可以穿墙而过，显然，这个地图还缺少最基本的功能——阻挡。

有一种简单的方案可以实现阻挡，让我们看一下 `Sprite` 类，其中有一个方法：

```
public final boolean collidesWith(TiledLayer t, boolean pixelLevel)
```

检测 `Sprite` 与 `TiledLayer` 的碰撞。这种检测是以 `Tile` 为单位的，当与 `Sprite` 重合的 `Tiles` 编号不为 0 时函数返回 `true`，否则返回 `false`。

下面让我们做一个小例子测试一下：

打开 `GameView_Old`, 增加一个 `Sprite` 类型的成员变量

```
// 主角
Sprite player;
```

在构造函数中初始化 `player`

```
// 初始化主角
player = new Sprite(BitmapFactory.decodeResource(res,
    R.drawable.player1), 16, 16);
player.setFrameSequence(new int[] { 0, 1 });
```

在 `onDraw` 中绘制 `player`

```
player.paint(c);
background.paint(c);
```

在 `onKeyDown` 中控制 `Sprite` 的运动

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    // TODO Auto-generated method stub
    switch (keyCode) {
        case KeyEvent.KEYCODE_DPAD_UP:
            x = player.getX();
            y = player.getY();
            player.move(0, -16);
            if(!player.collidesWith(background, false)) {
                y -= 16;
            }
            player.setTransform(Sprite.TRANS_NONE);
            player.setPosition(x, y);
            break;
        case KeyEvent.KEYCODE_DPAD_DOWN:
            x = player.getX();
            y = player.getY();
            player.move(0, 16);
            if(!player.collidesWith(background, false)) {
                y += 16;
            }
            player.setTransform(Sprite.TRANS_ROT180);
            player.setPosition(x, y);
            break;
        case KeyEvent.KEYCODE_DPAD_LEFT:
            x = player.getX();
```

```

        y = player.getY();
        player.move(-16, 0);
        if(!player.collidesWith(backGround, false)) {
            x -= 16;
        }
        player.setTransform(Sprite.TRANS_ROT270);
        player.setPosition(x, y);
        break;
    case KeyEvent.KEYCODE_DPAD_RIGHT:
        x = player.getX();
        y = player.getY();
        player.move(16, 0);
        if(!player.collidesWith(backGround, false)) {
            x += 16;
        }
        player.setTransform(Sprite.TRANS_ROT90);
        player.setPosition(x, y);
        break;
    }
    postInvalidate(); // 通知系统刷新屏幕
    return super.onKeyDown(keyCode, event);
}

```

可以看到，坦克只能在空白区域运动，这回不能上墙了。

但是这种方法还是比较粗糙的，很多时候不能实现我们的目的，比如坦克大战中，水和砖头是坦克不能通过的，但是掩体是可以通过的，还有子弹可以通过水域，这时候还是检测 **Tile** 的编号来的准确些。就是说，我们事先确定好那些编号的 **Tile** 可以通过，哪些不能。然后模仿 **collidesWith** 方法根据 **Tank** 的位置取得它下一步要到达的 **Tile** 的编号，并判断坦克是否被阻挡。

让我们用这个方案重写 **onKeyDown** 方法（为了简化教程，我们假设每次 **player** 和 **tile** 都是完全重合的）：

首先我们先定义一个函数用来判断 **Tank** 是否可以通过

```

// 判断坦克是否可以通过
private boolean tankPass(int x, int y) {
    // 不超过地图范围
    if (x < 0 || x > 12 * 16 || y < 0 || y > 12 * 16) {
        return false;
    }
}

```



```

    int tid = map[y / 16][x / 16];
    if (tid == 1 || tid == 2 || tid == -1)
        return false;
    return true;
}

```

然后在 onKeyDown 中运用新的方案

```

@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    // TODO Auto-generated method stub
    switch (keyCode) {
        case KeyEvent.KEYCODE_DPAD_UP:
            x = player.getX();
            y = player.getY();
            player.move(0, -16);
            if (tankPass(player.getX(), player.getY())) {
                y -= 16;
            }
            player.setTransform(Sprite.TRANS_NONE);
            player.setPosition(x, y);
            break;
        case KeyEvent.KEYCODE_DPAD_DOWN:
            x = player.getX();
            y = player.getY();
            player.move(0, 16);
            if (tankPass(player.getX(), player.getY())) {
                y += 16;
            }
            player.setTransform(Sprite.TRANS_ROT180);
            player.setPosition(x, y);
            break;
        case KeyEvent.KEYCODE_DPAD_LEFT:
            x = player.getX();
            y = player.getY();
            player.move(-16, 0);
            if (tankPass(player.getX(), player.getY())) {
                x -= 16;
            }
            player.setTransform(Sprite.TRANS_ROT270);
            player.setPosition(x, y);
            break;
        case KeyEvent.KEYCODE_DPAD_RIGHT:
            x = player.getX();
            y = player.getY();
            player.move(16, 0);

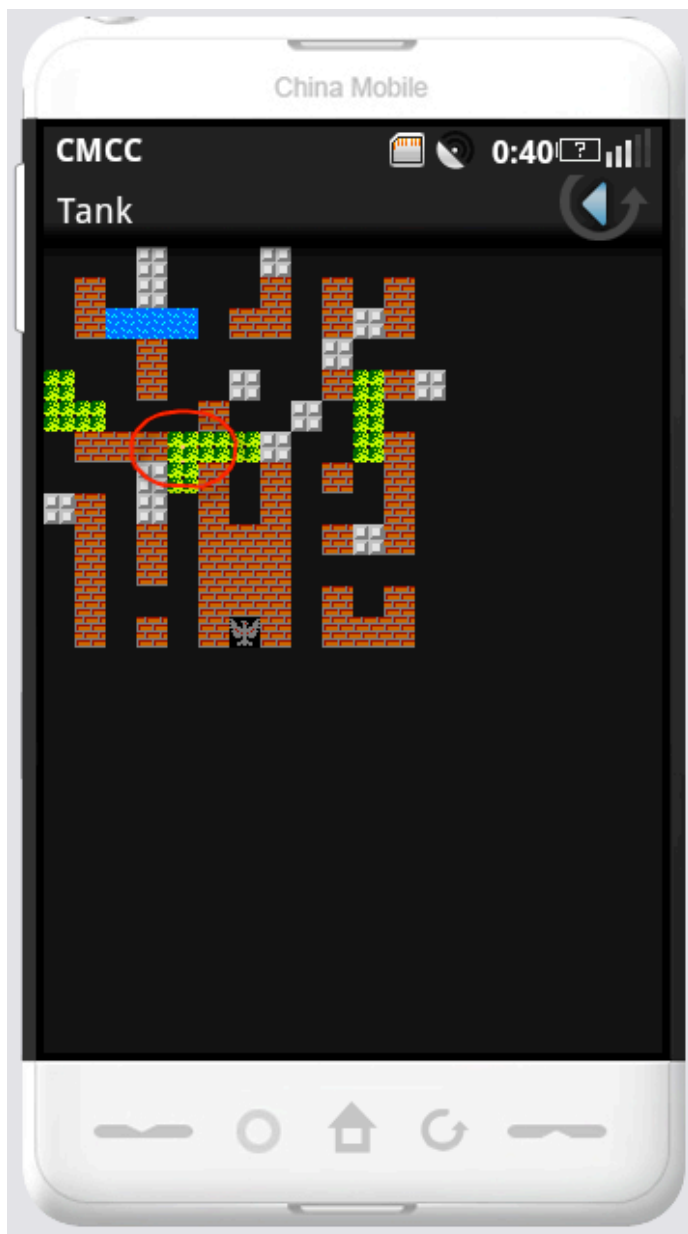
```

```

        if (tankPass(player.getX(), player.getY())) {
            x += 16;
        }
        player.setTransform(Sprite.TRANS_ROT90);
        player.setPosition(x, y);
        break;
    }
    postInvalidate(); // 通知系统刷新屏幕
    return super.onKeyDown(keyCode, event);
}

```

现在让我们运行一下看看吧，这回终于能够达到了我们想要的效果，我们的坦克正藏在掩体下面，并且它不能通过墙和水域。

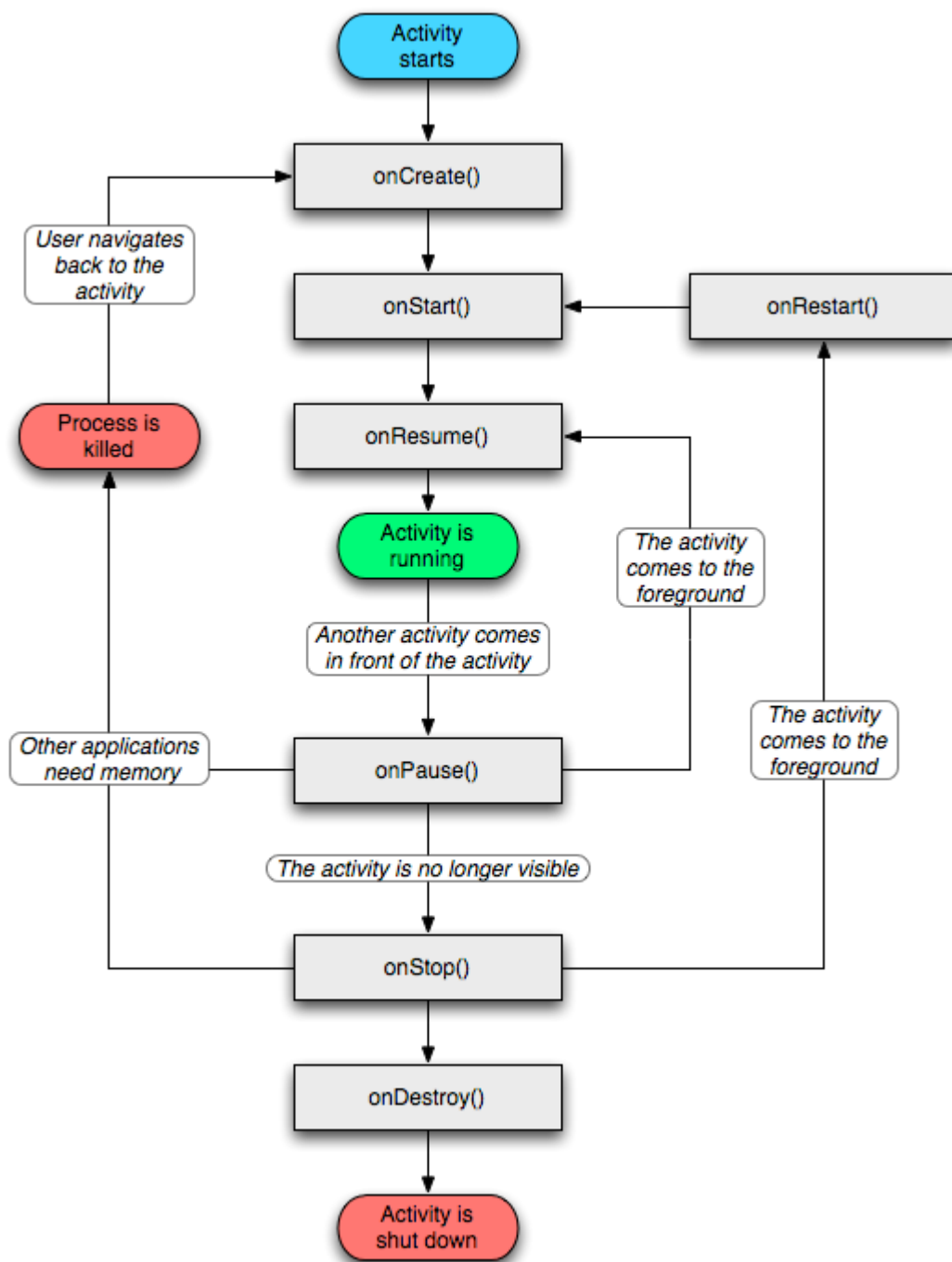


到此为止，关于地图的内容就讲解完毕了。这些内容并不复杂，首先介绍了使用 **Tile** 组成地图的原理，然后介绍了 **TiledLayer** 已经动态地图，最后演示了阻挡的实现方法。本章的大部分例子使用了 **GameView_Old**，并没有使用游戏循环，也没有涉及到地图的平滑滚动，在以后需要的时候会补充这部分知识。

本章示例程序 <http://u.115.com/file/f12827d8db>

第九章 游戏程序的生命周期

在讲解游戏程序的生命周期之前，让我们先看看普通 **Android** 应用的生命周期。关于生命周期，**SDK** 附带的文档上有详细的解释，让我们打开文档，找到 **andorid.app->Activity**，我们会看到这样一张图片



图片将整个程序的生命周期描述的非常清楚，为了加深理解，我们创建一个程序实际看一下这个过程。

创建项目 LifeCycle，sdk 就选择 1.6 吧。在 Activity 中重载如下几个函数，并增加 Log 语句：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.main);
        LogF();
    }

    @Override
    protected void onDestroy() {
        // TODO Auto-generated method stub
        super.onDestroy();
        LogF();
    }

    @Override
    protected void onPause() {
        // TODO Auto-generated method stub
        super.onPause();
        LogF();
    }

    @Override
    protected void onRestart() {
        // TODO Auto-generated method stub
        super.onRestart();
        LogF();
    }

    @Override
    protected void onResume() {
        // TODO Auto-generated method stub
        super.onResume();
        LogF();
    }

    @Override
    protected void onStart() {
        // TODO Auto-generated method stub
        super.onStart();
        LogF();
    }

    @Override
    protected void onStop() {
        // TODO Auto-generated method stub
        super.onStop();
        LogF();
    }
```

```
}
```

LogF()定义如下:

```
public static void LogF() {  
    Log.v(Thread.currentThread().getStackTrace()[3].getClassName(),  
Thread.currentThread().getStackTrace()[3].getMethodName());  
}
```

除了 onCreate 之外, 都需要手工添加, 重载函数的方法前面有哦, 一年过去了, 大家没忘吧:)

让我们在模拟器中运行这个程序。同时在 LogCat 中查看输出。前面好像没有讲到 LogCat, 但是很多代码用到了 Log, 大家都已经找到了吧。

程序启动后, 我们看到了 3 条自定义的 Log 信息:

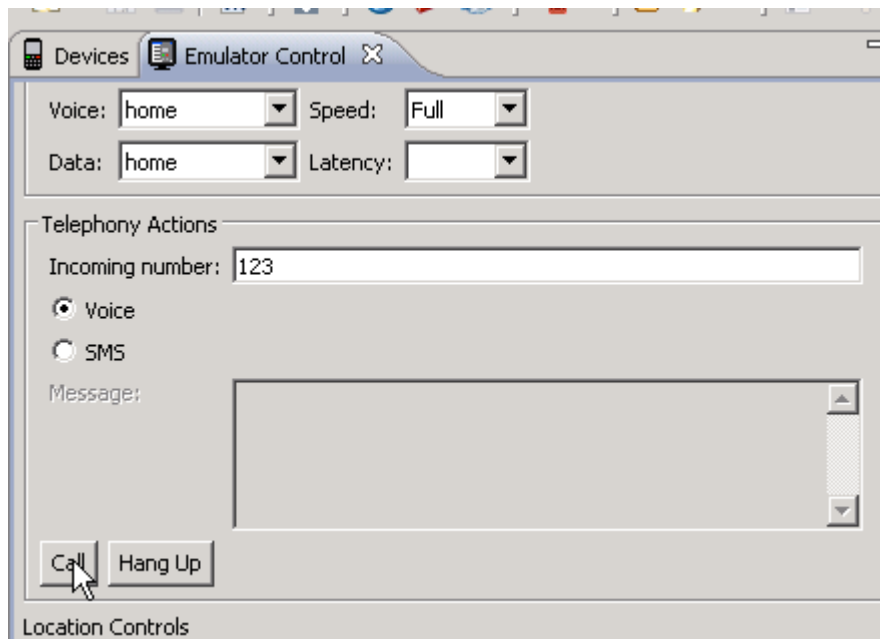
```
01-10 14:02:45.774 W 757 system.out Can't dispatch  
01-10 14:02:45.773 V 757 org.yexing.android.a... onCreate  
01-10 14:02:45.773 V 757 org.yexing.android.a... onStart  
01-10 14:02:45.781 V 757 org.yexing.android.a... onResume  
01-10 14:02:45.862 W 570 TextMessageService Starting intent
```

让我们按下返回键结束程序,

```
01-10 14:04:50.041 D 630 dalvikvm GC freed 153 KB  
01-10 14:04:50.651 V 757 org.yexing.android.a... onPause  
01-10 14:04:50.971 V 757 org.yexing.android.a... onStop  
01-10 14:04:50.981 V 757 org.yexing.android.a... onDestroy  
01-10 14:04:56.061 D 757 dalvikvm GC freed 1679 KB
```

这就是一个程序从创建到销毁的标准流程。但是作为手机应用, 我们前面提到的被抢占屏幕的情况就必须要被考虑。让我们来测试一下:

重新运行 LifeCycle, 在 DDMS 中模拟一个电话呼入



日志中出现了

```
org.yexing.android.a... onPause
InCallScreen onCreate()... this = com.an
org.yexing.android.a... onStop
```

这次没有调用 `onDestory`。

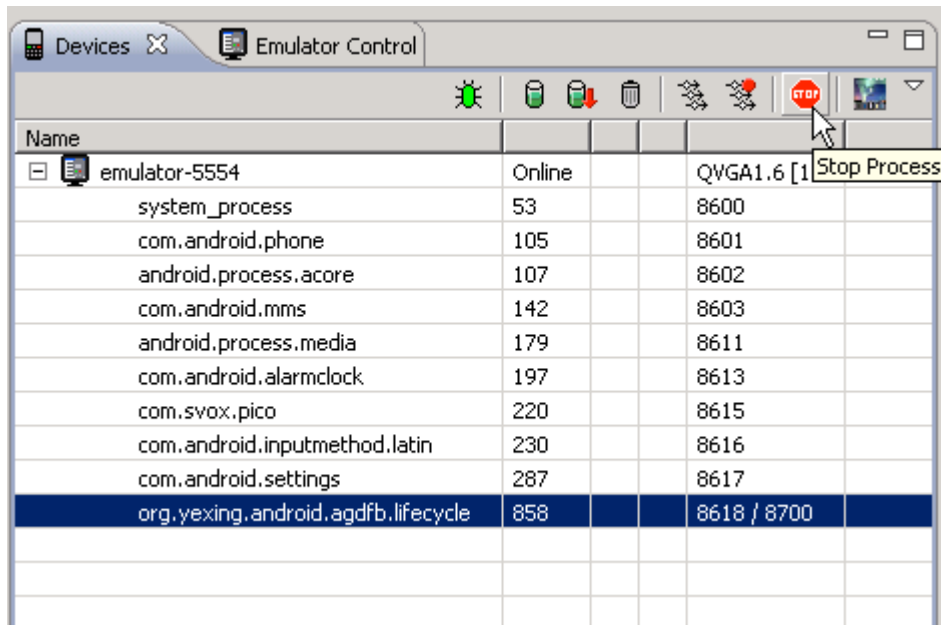
然后让我们把电话挂掉

```
1:00:00.000 org.yexing.android.a... onPause
1:00:00.000 org.yexing.android.a... onRestart
1:00:00.000 org.yexing.android.a... onStart
1:00:00.000 org.yexing.android.a... onResume
1:00:00.000 org.yexing.android.a... onCreate
```

同样，没有调用 `onCreate` 而调用了 `onRestart`。

另外还有一种情况，就是当程序被放置到后台过久，系统在一定条件下会自动将程序销毁，让我们看一下这种情况下程序的生命周期会有什么变化。

运行 `LifeCycle`，转到 `DDMS`，模拟一个来电，然后在 `Devices` 找到 `LifeCycle` 并强行停止他



我们会发现日志中并没有任何输出。

这时，让我们挂掉电话，日志中出现了如下三行

```

org.yexing.android... onCreate
org.yexing.android... onStart
org.yexing.android... onResume
ActivityManager      Displayed activity org.y

```

可以看到，程序被重新创建了，调用了 `onCreate` 而不是 `onRestart`，这与我们前面说的流程相悖，因为在这种情况下，我们应该继续程序的执行而不是重新初始化。那么如何解决这个问题呢？

方法如下，让我们重载下面这个函数：

```

@Override
protected void onSaveInstanceState(Bundle outState) {
    // TODO Auto-generated method stub
    super.onSaveInstanceState(outState);
    Log.v(this.toString(),
Thread.currentThread().getStackTrace()[2].getMethodName());
}

```

重新拨入电话，日志中出现了如下内容

```

ActivityManager      Starting activity: Intent {
org.yexing.android... onSaveInstanceState
org.yexing.android... onPause
InCallScreen         onNewIntent: intent=Intent

```

可以看到，在 `onPause` 之前执行了 `onSaveInstanceState`。那么执行了它有什么作用呢？是

不是就已经把程序状态保存了呢？还没有，保存的过程需要我们自己来编码。我们拿这个函数与其他的 `onXXX` 对比会发现，它与 `onCreate` 一样，都有一个 `Bundle` 类型的参数，而缺省的名字似乎已经透露了玄机，`onCreate` 的参数名叫 `savedInstanceState`，意为被保存的状态，正与 `onSavedInstanceState` 对应，那么名为 `outState` 意为输出状态的参数，功能就不言自明了。把需要保存的值放到 `outState` 中，在 `onCreate` 中检查 `savedInstanceState` 是否为 `null`，如果有值就取出来恢复现场。具体的用法，学习了游戏程序的生命周期之后会有实例讲解。

前面讲的是一个普通应用程序的生命周期，下面让我们进一步了解一个游戏程序的生命周期。我们的游戏同样基于 `SurfaceView`。根据前面讲过的内容，我们知道，现在程序中增加了游戏循环，它是一个单独的线程，因此在程序的生命周期中就增加了对游戏线程的操作。

在 `LifeCycle` 中增加 `GameView`，继承自 `SurfaceView`，实现 `SurfaceHolder.Callback` 和 `Runnable` 接口（前面已经讲过哦）。重载函数，并在函数中添加 `Log`。修改 `onCreate` 使其显示 `GameView`。

让我们运行程序看看 `Log` 的输出：

启动程序

```
org.yexing.android... onCreate
org.yexing.android... onStart
org.yexing.android... onResume
org.yexing.android... surfaceCreated
org.yexing.android... surfaceChanged
```

来电呼入

```

ActivityManager Starting activity: intent { act=android
org.yexing.android... onSaveInstanceState
org.yexing.android... onPause
InCallScreen onNewIntent: intent=Intent { act=android
InCallScreen internalResolveIntent: action=android.i
InCallScreen onResume()...
PhoneApp disable keyguard
InCallScreen - onResume: initial status = SUCCESS
InCallScreen syncWithPhoneState()...
PhoneApp updateWakeState: callscreen true, dial
PowerManagerService mDimDelay=-1 while trying to dim
PhoneApp updateWakeState: keepScreenOn = true (i
PhoneApp updateWakeState: callscreen true, dial
PhoneApp updateWakeState: keepScreenOn = true (i
org.yexing.android... surfaceDestroyed
MediaPlayer Couldn't open file on client side, tryi
ActivityManager checkComponentPermission() adjusting {p
ActivityManager checkComponentPermission() adjusting {p
org.yexing.android... onStop
ActivityManager checkComponentPermission() adjusting {p

```

通话结束

```

PhoneApp updateWakeState: keepScreenOn
org.yexing.android... onRestart
org.yexing.android... onStart
org.yexing.android... onResume
org.yexing.android... surfaceCreated
org.yexing.android... surfaceChanged

```

应用结束

```

org.yexing.android... onPause
org.yexing.android... surfaceDestroyed
org.yexing.android... onStop
org.yexing.android... onDestroy
deliberate CC: freed 3000 bytes at 13601

```

再追加两种前面没有讲到的情况，一是屏幕翻转（在模拟器中的快捷键是 Ctrl+F11）

```

org.yexing.android... onSaveInstanceState
org.yexing.android... onPause
ARMAsembler generated scanline__00000077:030101
org.yexing.android... onStop
org.yexing.android... onDestroy
org.yexing.android... surfaceDestroyed
StatusBar updateResources
org.yexing.android... onCreate
org.yexing.android... onStart
org.yexing.android... onResume
org.yexing.android... surfaceCreated
org.yexing.android... surfaceChanged

```

可以看到，如果程序没有设置固定的横屏或竖屏状态，每次翻转屏幕，就会将程序关闭并重新启动。

另一种情况是休眠。当我们没有强制应用不休眠时，或短暂按下电源键时，应用会进入暂停状态，屏幕上显示锁屏画面。让我们解锁屏幕，应用被唤醒，继续执行，让我们看一下此时的 Log：

```
WindowManager I'm tired mEndcallBehavior=0x2
org.yexing.androi... onSaveInstanceState
org.yexing.androi... onPause
KeyguardViewMediator wakeWhenReadyLocked(82)
KeyguardViewMediator handleWakeWhenReady(82)
KeyguardViewMediator pokeWakelock(5000)
org.yexing.androi... onResume
KeyguardViewMediator pokeWakelock(5000)
```

前面我们也讲过，我们在一个独立的线程中进行游戏循环，依照一般应用的生命周期，在适当的时候开始和结束游戏循环，保存游戏状态，就能够完美的控制游戏程序的生命周期了。

首先让我们用文字总结一下这个过程：

在 **onCreate** 中，初始化游戏状态或恢复游戏状态。注意，这里不是初始化图片，声音等数据(Data)，只是游戏内状态，比如现在是开始菜单还是在游戏当中，玩家的位置，敌人的位置等等数值(Value)；

在 **onRestart** 中恢复游戏状态；

在 **onResume** 中开始游戏循环（本应在 **onStart** 中，但是我们看到，各种情况下，**onResume** 都会在 **onStart** 之后调用，所以简单的用 **onResume** 代替了 **onStart**）；

在 **onSaveInstanceState** 中保存游戏状态；

在 **onPause** 中结束游戏循环；

在 **onDestory** 中销毁游戏数据。

另外补充几句，游戏的保存和读取实际上是不属于应用程序生命周期的，而是一种游戏内操作。但是，你也可以根据程序的生命周期来决定是否应该保存和读取游戏，比如实现自动保存，以防止程序意外终止造成的损失。

最后，就让我们用一个程序来演示本章所讲的内容。我们依然使用计时器程序来演示，因为他很直观。程序的关键点在于当程序被隐藏时，停止计时，被重新显示时继续计时。

首先在游戏进程开始后进行计时，我们首先要取得开始的时间：

```
public void run() {
```

```

start = System.currentTimeMillis();
while(run) {
    now = System.currentTimeMillis();
    .....

```

那么计时器显示的数值就是当前时间 **now** 减去开始时间 **start**，为了便于观察，我们用秒作为单位，就是 $(now - start)/1000$ 。

在程序被暂停后，我们需要停止游戏循环，可以终止线程也可以停止对数值的计算，本例中我们选择终止线程。我们为 **GameView** 增加函数 **pause**

```

public void pause() {
    run = false;
}

```

如果现在运行程序，我们会发现，程序被暂停再恢复就会重新计时，那么如何在程序恢复后继续计时呢？我们要在暂停时将当前的时间保存起来，恢复后就可以用 **now-start** 加上这个时间，就实现连续计时了。

我们设定变量 **last** 保存上次时间，那么计时器当前的值就是

```

millis = last + now - start;

```

而 **pause** 函数修改成

```

public void pause() {
    run = false;
    last = millis;
}

```

这样就解决了重新计时的问题。但是别忘了，前面我们还特别提到一种情况，就是程序会被系统销毁，并重新执行，那么这种方案就不适用了，因为程序重新执行时，**last** 会被重新初始化，保存的值也就随之丢失了。当然，聪明的读者肯定记得前面我们说过的 **onSaveInstanceState**，没错，下面就是它发挥作用的时候了。

首先我们在 **GameView** 中创建函数 **save**

```

public void save(Bundle outState) {
    outState.putLong("last", millis);
}

```

前面没有讲解 **Bundle**，下面就让我们看看他的用法。

如果你了解 **Map**，你会发现两者很相似，他们都可以用来存储 **key-value** 值对，读取方法也

一样，只是 **Bundle** 的函数设定了变量类型而已。

有了保存就有读取，让我们增加 **load** 函数

```
public void load(Bundle savedInstanceState) {  
    if(savedInstanceState != null) {  
        last = savedInstanceState.getLong("last");  
    }  
}
```

最后我们再增加一个 **resume** 函数，来启动线程，前面说过 **resume** 和 **start** 起着同样的作用

```
public void resume() {  
    run = true;  
    new Thread(this).start();  
}
```

到此为止，我们已经有了控制游戏进程的所有函数，下一步就是在 **Activity** 中相应的回调函数中调用这些函数了，具体怎么调用，我想读者心中已经有数了吧。当然，你可以在本章的例程中找到完整的代码。

这个例子虽然简单，但即使是很复杂的游戏（使用 **SurfaceView**）也可以通过这个方法来控制。但是细心的读者可能会发现，例子程序有一些误差，因为在 **surface** 还没有被创建时游戏循环就已经开始了，所以可能会直接看到屏幕显示 **3、4** 等，这当然不是致命的问题，因为大多数游戏不是在一开始就计时的。但是有一种情况却会有一些麻烦，就是我们用电源键让屏幕休眠，再开启屏幕，这时候手机应该处于锁屏状态，但 **onResume** 已经被调用，就是说，游戏循环已经开始了，而用户却无法看到。为了应对这种情况，我们可以重载 **onWindowFocusChanged** 函数，只要程序被其他界面遮挡或遮挡消失，就会调用这个函数。因为这个函数在两种情况都会调用，所以我们必须区分当前程序的状态时被遮挡还是被显示，读者不妨自己动手试试看（提示：函数 **hasWindowFocus** 会很有用）。

本章示例程序 <http://u.115.com/file/f1ca1dfa00>

第十章 游戏循环的设计

前面的几章中，曾多次提到游戏循环，这一章就让我们一起了解游戏循环的相关概念，学习如何使用游戏循环。

我们知道，游戏的主体通常在一个循环体中，最初，我们用一个变量来表示游戏的状态，比如

```
gameState = STATE_STARTMENU;
```

每次循环都判断当前状态，调用不同的函数进行处理。这种方法简单有效，所有的逻辑代码都在一个类里，不必考虑隐藏和划分，只需要定义一些函数即可。但是它的优点也是它的缺点，这种结构化的方法通常让程序显得很大，有些混乱。所以我们采用另一种方式管理游戏循环，将场景（Scene）的概念引入程序中。

我们可以认为一个游戏由一些独立的场景组成，如开始菜单，游戏中画面，过关画面等等都可以是场景，而整个游戏就在这些场景之间不断切换，直到退出为止。场景这个词用在这里非常贴切，所有游戏的元素，不论是选项菜单还是玩家角色，都像是舞台上的演员(Actor)，他们根据不同的场景登上舞台或离开舞台。有了场景的支持，演员们不再挤在一起，而是被划分成一组一组的，彼此隔离，这样会让程序变得清晰有序，而游戏循环就变成了场景循环。

我们首先创建一个 SceneBase 类，作为所有 Scene 的基类：

```
public abstract class SceneBase {  
  
    abstract public void tick();  
    abstract public void update(Canvas c);  
  
}
```

其中 tick 为一个动画帧，让所有演员做出动作。update 将演员们显示在舞台上。

然后，我们在 GameView 建立一个 SceneBase 类型的变量来表示当前的场景，只有当前的场景才会被绘制。

```
static SceneBase currentScene;  
.....  
if(c != null) {
```

```

        currentScene.tick();
        currentScene.update(c);
    }

```

最后通过一个函数改变当前场景，这就实现了最基本场景转换。

```

    public static void setCurrentScene(SceneBase scene) {
        currentScene = scene;
    }

```

为了测试这种方案，我们创建两个场景类 `SceneStartMenu` 和 `SceneMain` 分别表示开始菜单和游戏主循环，通过点击屏幕在这两个场景之间切换。

我们首先在 `GameView` 中定义这两个场景

```

    static SceneStartMenu sceneStartMenu = null;
    static SceneMain sceneMain = null;

```

在 `GameView` 的构造函数中初始化 `sceneStartMenu`，将它作为初始场景

```

    if(sceneStartMenu == null)
        sceneStartMenu = new SceneStartMenu();
    setCurrentScene(sceneStartMenu);

```

然后我们在 `GameView` 中重载 `onTouchEvent`（也要为 `SceneBase` 增加一个响应的函数）

```

@Override
    public boolean onTouchEvent(MotionEvent event) {
        // TODO Auto-generated method stub
        return currentScene.onTouchEvent(event);
    }

```

在 `SceneStartMenu` 中

```

    public boolean onTouchEvent(MotionEvent event) {
        // TODO Auto-generated method stub
        switch(event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                if(GameView.sceneMain == null)
                    GameView.sceneMain = new SceneMain();
                GameView.setCurrentScene(GameView.sceneMain);
                break;
        }
        return true;
    }

```

```
}
```

SceneMain 与之相反即可

然后我们修改两个 scene 的 update 函数

```
public void update(Canvas c) {  
    // TODO Auto-generated method stub  
    c.drawARGB(255, 0, 0, 0);  
    c.drawText("SceneStartMenu", GameView.width/2,  
GameView.height/2, paint);  
}
```

这样就能知道当前的 Scene 是哪一个了

现在让我们运行一下这个程序，点击屏幕，可以看到两个场景之间切换的效果。到此为止，本章的内容就讲完了，很短是吧，但是很有用，有了这个结构，加上前一章的生命周期控制，你的程序会变得更整洁有序。当然，在具体应用中，你必须扩充 scene 的内容，比如场景之间的交互，你可能需要一个或几个 setter 或者 Map 结构（模仿 Activity）。请保持场景之间传递的是数值而不是实例，是设定给演员的指令而不是演员本身。

本章示例程序 <http://u.115.com/file/f1cf493609>

第十一章 演员（Actor）、视口（ViewWindow），演出开始

本章内容与第七章、第八章关系非常密切，如果对这两章的内容不熟悉请大家先浏览一下七、第八章，再回来看本章。

Actor 是一个接口，他的作用是统一类的行为（读者可以阅读一下 Facade 模式相关文章）。我们用一个比喻来说明：演员们有了各自的剧本，导演对所有演员说：做下一个动作！演员们就会各自行动。而不用导演分别告诉每个人，你要这样做，他要那样做。具体到程序中，帧动画、动态图块两种操作会调用完全不同的函数，这样不利于在游戏循环中做出一致的处理。所以我们让他们都实现 Actor 接口，只要调用接口定义的函数，他们就会做出各自的动作。Actor 接口的定义很简单：

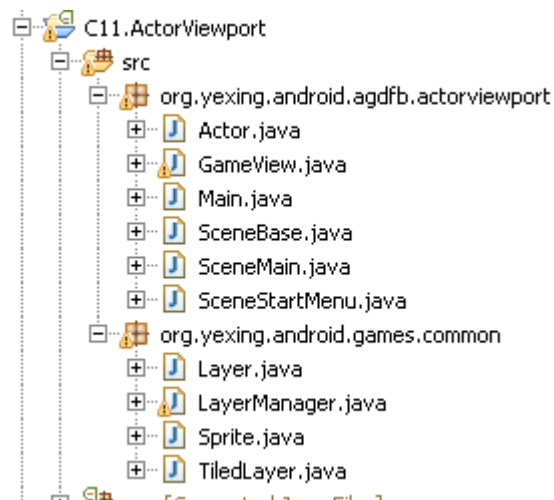

```
public interface Actor {
    public void tick();
}
```

对于实现了 **Actor** 接口的任何类型，发出的指令就只有一个：**tick**。

下面请看实例演示：

我们将创建两个类 **Tank** 和 **Map**，分别继承自 **Sprite** 和 **TiledLayer**，都实现 **Actor** 接口。为 **tank** 创建帧动画，为 **Map** 创建动态图块，然后将他们显示在 **SceneMain** 中。

首先，我们可以复制第十章的例子创建一个新项目，并将第八章中用到的 **org.yexing.android.games.common** 包拷贝到项目中。



创建一个 **Tank** 类，继承自 **Sprite**，实现 **Actor** 接口。

在 **tick** 函数中播放帧动画

```
public void tick() {
    // TODO Auto-generated method stub
    nextFrame();
}
```

然后创建一个 **Map** 类继承自 **TiledLayer**，实现 **Actor** 接口。

在 **tick** 函数中播放动态图块

```
public void tick() {
    // TODO Auto-generated method stub
    if (getAnimatedTile(-1) == 4) {
        setAnimatedTile(-1, 5);
    }
}
```

```

    } else {
        setAnimatedTile(-1, 4);
    }

}

```

最后在 `SceneMain` 中创建这两个类的实例并显示他们

```

Layer layers[] = new Layer[2];
int mapdate[][] = { { 0, 0, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0 },
    { 0, 1, 0, 2, 0, 0, 0, 1, 0, 1, 0, 1, 0 },
    { 0, 1, -1, -1, -1, 0, 1, 1, 0, 1, 2, 1, 0 },
    { 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 0, 0, 0 },
    { 3, 0, 0, 1, 0, 0, 2, 0, 0, 1, 3, 1, 2 },
    { 3, 3, 0, 0, 0, 1, 0, 0, 2, 0, 3, 0, 0 },
    { 0, 1, 1, 1, 3, 3, 3, 2, 0, 0, 3, 1, 0 },
    { 0, 0, 0, 2, 3, 1, 0, 1, 0, 1, 0, 1, 0 },
    { 2, 1, 0, 2, 0, 1, 0, 1, 0, 0, 0, 1, 0 },
    { 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 2, 1, 0 },
    { 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0 },
    { 0, 1, 0, 1, 0, 1, 6, 1, 0, 1, 1, 1, 0 }, };

public SceneMain() {
    paint = new Paint();
    paint.setColor(Color.WHITE);
    paint.setTextAlign(Align.CENTER);

    Bitmap bmpTank =
BitmapFactory.decodeFile("/sdcard/player1.png");
    Bitmap bmpTile = BitmapFactory.decodeFile("/sdcard/tile.png");

    Tank tank = new Tank(bmpTank, 32, 32);
    tank.setFrameSequence(new int[]{0, 1});

    Map map = new Map(13, 13, bmpTile, 32, 32);
    map.createAnimatedTile(4);

    for(int y=0; y<13; y++) {
        for(int x=0; x<13; x++) {
            map.setCell(y, x, mapdate[x][y]);
        }
    }
    layers[0] = map;
}

```

```

        layers[1] = tank;
    }

    @Override
    public void update(Canvas c) {
        // TODO Auto-generated method stub
        c.drawARGB(255, 0, 0, 0);
        c.drawText("SceneMain", GameView.width/2, GameView.height/2,
paint);
        for(int i=0; i<2; i++) {
            ((Actor) layers[i]).tick();
            layers[i].paint(c);
        }
    }
}

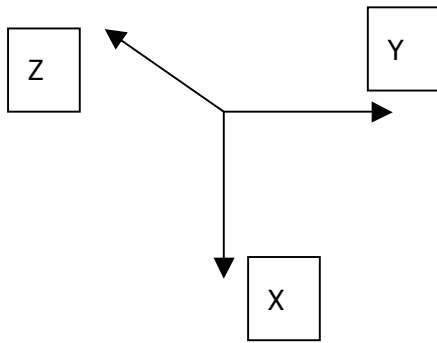
```

需要读者关注的就是 **update** 方法中对 **tick** 的调用。运行程序，我们可以看到动画的效果。

还需要注意一点，我们将位图文件放置在了 **sdcard** 的根目录。读者可以在项目的 **res/drawable** 下找到这两个文件，自行 **push** 到 **sdcard** 中。

下面再来看 **ViewWindow**。我们可以将它理解成舞台的前台，或者相机的取景器。演员只有走到前台，观众才能看得见。而演员走下台，就从观众视线消失。通常，屏幕就是一个视口，因为无论如何我们也看不到屏幕之外的东西。而有时候，我们需要更小的视口，局部的变化，就需要自行定义视口了。其实，视口的功能我们已经实现了，他就在 **LayerManager** 中，只是我们前面没有用到也就没有做介绍。

下面就先让我们来了解一下 **LayerManager** 的功能。我们知道 **Sprite**、**TiledLayer** 都是继承自 **Layer**，那么 **LayerManager**，顾名思义，就是用来管理 **Layer** 的。具体说是管理 **Layer** 的显示的。要使用 **LayerManager**，我们首先调用函数 **append()**或 **insert()**将 **Layer** 加入到 **LayerManager** 中，就如我们将纸张放入文件夹一样，然后调用 **paint** 就可以将所有 **Layer** 一次显示出来。**Layer** 有一个属性：**z**，表示 **Layer** 的 **Z** 坐标，如图：



Z 坐标从屏幕内指向屏幕外，也就是说 Z 坐标越大离用户越近，前面的 Layer 会遮住后面的 Layer。

我们改写前面的程序，用 **LayerManager** 代替 **Layer** 数组

```
public SceneMain() {  
    super();  
.....  
    layerManager.append(map);  
    layerManager.insert(tank, 100);  
.....  
    public void update(Canvas c) {  
.....  
        for(int i=0; i<layerManager.getSize(); i++) {  
            ((Actor) layerManager.getLayerAt(i)).tick();  
        }  
  
        layerManager.paint(c, 0, 0);  
    }  
}
```

熟悉了 **LayerManager** 之后就让我们来学习视口。我们可以将 **Layer** 想像成一张张非常大的画布，他们被放在 **LayerManager** 这个容器中，视口就是在容器上开一个小窗户，使用户只能看到窗口那一部分，其余的区域都不可见。

让我们看一下 **LayerManager** 的构造函数

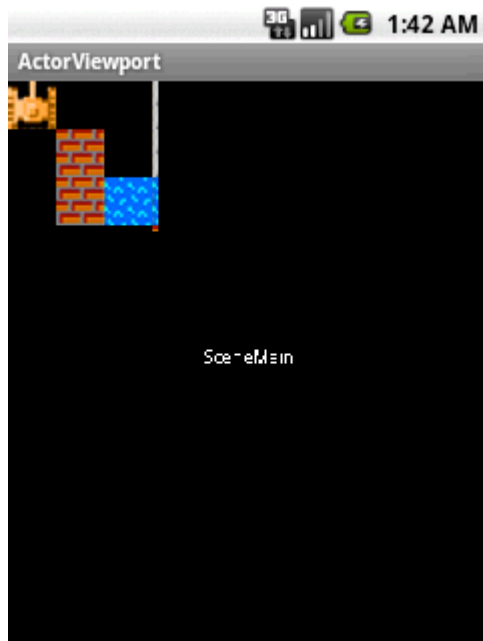
```
public LayerManager() {  
    setViewWindow(0, 0, Integer.MAX_VALUE, Integer.MAX_VALUE);  
}
```

我们可以看到在这里，已经定义了一个视口。这个视口非常大，可以想象他与 **Layer** 一般大，

所以我们看不到他的效果。但是如果我们把视口缩小呢？

```
public void update(Canvas c) {  
.....  
    layerManager.setViewWindow(0, 0, 100, 100);  
    layerManager.paint(c, 0, 0);  
}
```

运行程序，可以看到只有视口内的部分被显示了出来。

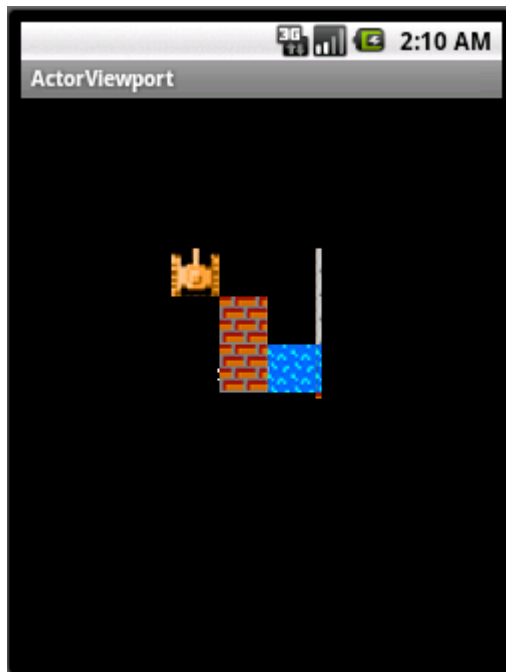


ViewWindow 的难点在坐标。让我们回头看

```
layerManager.setViewWindow(x, y, width, height);  
layerManager.paint(c, x, y);
```

这两个函数都用到了坐标，他们有着完全不同的含义。首先看 **paint**，其中的 **x, y** 表示视口的左上角相对于屏幕的坐标。我们前面说过，视口可以理解为 **LayerManager** 上的窗口，改变这个坐标，整个 **LayerManager** 就会跟着一起移动。让我们修改程序，将视口显示在（100, 100）的位置

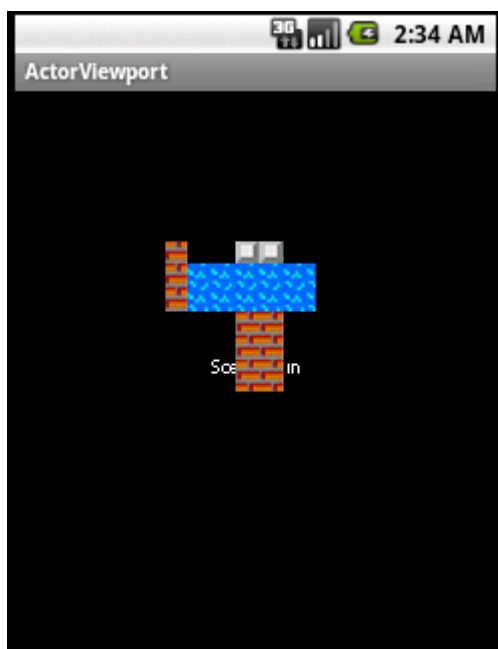
```
layerManager.paint(c, 100, 100);
```



可以看到，所有 Layer 的左上角都移动到了（100，100）的位置。

再来看 `setViewWindow`，其中的 `x`，`y` 表示视口相对于 Layer 左上角的坐标，让我们修改他们的数值，看看效果。

```
layerManager.setViewWindow(50, 50, 100, 100);
```



可以看到，左上角的坦克不见了。

合理的运用 `setViewWindow` 可以很方便的实现滚屏效果。

本章示例程序 <http://u.115.com/file/f198eeda92>

第十二章 音乐与音效

广告说的好“没有声音再好的戏也出不来”，下面就让我们为程序加入声音效果。

一般情况下，游戏中的声音分为音乐和音效两个部分。直观上的区别，音乐播放的时间较长，资源文件较大。音效播放时间较短，资源文件比较小。在 **Android** 中，我们使用两种不同的方法播放音乐和音效。

播放音乐需要用到 **MediaPlayer** 类，**MediaPlayer** 本身比较复杂，这里我们只做一个简单的介绍，满足播放音乐的基本要求即可，读者可以参考帮助文档中的内容深入研究。

为了调用方便，我们创建一个 **Util** 类，将播放背景音乐的方法封装到 **Util** 中

```
public class Util {  
  
    static MediaPlayer mp;  
    static void playBGM(String path, boolean looping) {  
        if (mp == null) {  
            mp = new MediaPlayer();  
        }  
        mp.reset();  
        try {  
            mp.setDataSource(path);  
            mp.prepare();  
        } catch (IllegalArgumentException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        } catch (IllegalStateException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
        mp.setLooping(looping);  
        mp.start();  
    }  
}
```

```

    }

}

```

让我们分析一下这段代码：首先，我们需要一个 **MediaPlayer** 实例，然后重置 **MediaPlayer** 到 **Idle** 状态。注意，这个操作是必需的，如果当前正在播放，不重置状态直接调用 **setDataSource** 会抛出异常。然后指定我们要播放的文件，并调用 **prepare** 方法进行准备。如果你想指定开始时间（**seekTo**）必须在 **prepare** 之后进行。还可以设定是否循环，最后就是调用 **start** 播放了。

封装好后，我们可以方便的使用这个函数，我们在 **SceneStartMenu** 中重载 **start** 方法

```

public void start(SurfaceHolder surfaceHolder){
    Util.playBGM("/sdcard/sample.mid", false);
}

```

这样，程序运行后就开始播放音乐了。还要注意的，我们应该在适当的时候释放 **MediaPlayer** 占用的资源。

运行一下编辑好的代码看看效果吧。**sample.mid** 在 **res/raw** 目录下。

下面让我来学习如何播放音效，它将使用完全不同的方法 **SoundPool**。**SoundPool** 虽然是基于 **MediaPlayer** 的，但是他被优化用来同时播放多个文件，而且不适合播放较大的文件，可以说，他就是为播放音效定制的。下面就让我们看一下播放音效的代码：

```

private static int MAX_CHANNEL = 6;
private static SoundPool soundPool;
private static HashMap<String, Integer> soundPoolMap;
private static AudioManager audioManager;

```

首先设定声道数，如果同时播放的音频超过这个数量，最先播放的音频就会被关闭。然后定义 **SoundPool** 变量。大家注意 **soundPoolMap** 这个变量，下面会重点讲解。

```

public static void playSE(String path, float volume, int loop, float
rate) {
    if(soundPool == null) {
        soundPool = new SoundPool(MAX_CHANNEL,

```



```

AudioManager.STREAM_MUSIC, 10);
    }

    if (soundPoolMap == null) {
        soundPoolMap = new HashMap<String, Integer>();
    }
    if (!soundPoolMap.containsKey(path)) {
        soundPoolMap.put(path, soundPool.load(path, 1));
    }
}

```

这里我们要讲解一下 **soundPoolMap**。在音频被播放前需要先将它载入到内存，载入后，会得到一个 **streamID**，**SoundPool** 中的很多函数都使用这个 **ID** 来控制特定的音频流。所以，我们将这个 **ID** 保存到 **map** 中，并与音频文件的路径相对应。后面就可以通过路径来找到这段音频流了。

```

    if (volume < 0) {
        if (audioManager == null)
            audioManager = (AudioManager)Main.getInstance()
                .getSystemService(Context.AUDIO_SERVICE);
        volume =
audioManager.getStreamVolume(AudioManager.STREAM_MUSIC);
    }
    while (soundPool.play(soundPoolMap.get(path),
        volume, volume, 0, loop, rate) == 0);
}

```

这个方法并不是很科学，最好还是在初始化的时候载入音频，大家可以另行撰写一个 **load** 方法。

```

public void playSE(String path) {
    playSE(path, -1f, 0, 1f);
}

public void stopSE(String path) {
    if (soundPool != null && soundPoolMap != null
        && soundPoolMap.containsKey(path)) {
        soundPool.stop(soundPoolMap.get(path));
    }
}

```

修改 **SceneStartMenu** 中的 **start** 函数测试一下吧

```
    public void start(SurfaceHolder surfaceHolder){  
        //      Util.playBGM("/sdcard/sample.mid", false);  
        Util.playSE("/sdcard/system11.ogg");  
        Util.playSE("/sdcard/system12.ogg");  
    }  
}
```

本章示例程序 <http://u.115.com/file/f1435ee7e8>

现在，这部教程终于可以告一段落了，虽然有点虎头蛇尾，但我大概已经讲解了制作一个简单游戏所用到的各方面的知识。如果还有不足，请读者留言补充，大家互相学习，一起提高。最终我在教程中没有完成坦克大战，不过我会基于坦克大战设计一个新的游戏，不再使用 **Java**，而是一个全新的跨平台框架，到时候再与大家分享我的心得。