**游戏中的碰撞检测[转]**

2011-06-03 22:16:43|  分类： [Android游戏](http://blog.163.com/ppy2790@126/blog/#m=0&t=1&c=fks_084064084083084066084084087095085086082065080084084071085) |  标签：[游戏](http://blog.163.com/ppy2790@126/blog/#m=0&t=3&c=游戏)   |字号大中小 订阅

游戏中的碰撞检测方式有很多，不同的算法之间主要是在精度和速度之间权衡。以下几种方式按照速度排序说明。以2D为例，3D不过是增加了一维罢了，算法理解上没太大区别。

**一、地图格子划分检测**

最简单的一种检测，就是把地图（或者称为场景，总之是指碰撞发生的范围）划成一个个格子，类似仙剑奇侠传这样。假设地图有800\*600px，20\*20个像素为一格。那么可以划为40\*30个格子。地图中参与检测的对象都存储着自身所在的格子坐标，判断碰撞就显而易见了，例如可以认为两个物体在相邻格判为碰撞，或者两个物体在同一格。采用这种方式有个要求，就是地图中所有可能参与碰撞的物体都要是20\*20像素左右大小或者是其整数倍，例如房子占了3\*3个格子，诸如此类。如果不遵守这个规则，有的物体只占了格子的一半，那么在玩家眼里这种检测就显得非常的粗糙。这种检测就像是把地图的像素点放大几十倍一样，与逐像素检测相比，效率提高了几十倍甚至上百倍。这种方式可运用于对检测要求不严格的游戏，例如踩地雷的RPG、推箱子之类的智力游戏。

**二、矩形检测**

当地图中的物体不能严格按照某个块大小的整数倍来绘制时，那么就需要另想其他的方法。这种方法适用于地图中的物体近似为矩形或者虽然不是矩形，但是碰撞精度要求不高的情况下。每个物体记录一个能够将自己框住的最小矩形的左上角坐标和矩形长宽。碰撞退化为判断矩形与矩形之间是否重叠，而这仅需要4次比较即可得出，速度很快。但为了判断整个场景中的物体，必须取第一个物体，迭代其他所有物体进行判断，再取第二个物体，迭代除第一第二个物体外的所有物体进行判断，以此类推。总计要进行（n-1）！次矩形判断才能准确得出场景中所有的碰撞可能。

**三、圆形检测**

与上一种方法类似，区别在于用一个能够包含物体的最小圆代替了矩形。主要是考虑到游戏中的物体外形以平滑为主，例如人物角色。而判断两个圆是否碰撞的计算也很简单，就是判断两个圆心之间的距离是否小于两个圆的半径之和。虽然球形检测在某些情况下提高了精度，但却损失了速度，因为点距离的计算需要用到平方和开方。具体相比慢多少我就不太清楚了。另外，为了计算整个地图的所有碰撞可能，也要进行（n-1）！次比较。

**四、像素检测**

精确到像素级，已经不能比这更精确了，相对的，效率也是最低的。怎样判断两个物体是否碰撞呢？在过去png格式图片还不盛行的时候，游戏中用到的图片中的透明部分是指定用某种颜色来表示的，例如洋红色。就像电影中的绿幕蓝幕，通过处理把这些颜色的像素点当做透明点处理，而为了判断检测，需要准备一张原图像的黑白图，黑色区域表示透明，这张图片中的每个像素值为0或者1，判断检测的时候取两张图片的黑白图，进行与运算，结果为1（有白点重叠），则判为碰撞。但是现在有了PNG和XNA，逐像素检测就相对简单一些。首先仍然需要有一个矩形框包围物体，通过矩形检测得到重叠的矩形区域可以大大减少检测的像素点数量。然后在这个区域内，取两个图片的点逐行逐列迭代，如果遇到某个点两张图片均有颜色存在，即判为碰撞。同理，进行（n-1）！次比较后得到全地图的碰撞可能。

**五、四叉树检测**

准确的说这事在第三四五种方法的基础上的优化策略，或者说是第一种方法同后三种方法的组合应用。主要是针对那最后的（n-1）！次比较。方法是，像第一种方法一样将地图分为格子，格子的大小应该能够容纳10个左右的地图中最大物体，例如一个800\*600的地图可能就划为9个区。同样的，每个物体要记录自己所在的区坐标以及矩形包围盒。如果该物体完全位于该区内，则只要将其与该区内的其他物体判断碰撞。如果该物体虽然位于某个区，但是小部分位于隔壁区，则额外的需要迭代隔壁区的物体，这点效率损失是可以容忍的，相比于迭代全地图的物体。

有个问题，我怎么知道哪些物体是跟该物体位于同一个区呢？那不是还是要迭代一遍所有物体？这时候就是题目发挥的地方的，之所以称为四叉树检测（当然，这名字是我自己取的），就是因为那些区块是以四叉树的方式链接的，即得到一个区块的对象，就可以直接得到其上下左右相邻的区块的对象，而物体可以是存储在所在区的一个列表中。这样就不用遍历所有物体也可以直接取出隔壁区的物体了。当地图很大的时候，四叉树的优势体现得很好。

**六、3D中的碰撞检测**

以上是我所掌握碰撞方法，可能还有更多吧。那么3D中的检测其实是2D的延伸，例如矩形检测变为立方体检测，圆形检测引申为球形检测，四叉树检测进化为八叉树检测。

当然了，凡是有例外。逐像素检测方法在3D中没有相对应的方法，因为3D中的物体的表示最小单元是三角型而非点。其实也可以说逐三角片检测是逐像素检测的3D版，但毕竟是平面碰撞的检测，需要一定的计算公式，而不是与或一下就OK的。这里就不赘述了。

**总结**

选择哪种方法，要根据你的游戏需要，例如推箱子游戏，显然只要格子碰撞检测就足够了。而一些以球作为主要物体的游戏例如射击游戏则可以考虑圆形碰撞。

同样的，在有些碰撞精度要求很高的游戏中，还要对这些方法进行一定的变化，例如格斗游戏，作为判断的单元不是整个人而是四肢、身体等部位，需要更多的包围盒来表示一个物体。

## 任意多边形的碰撞检测——向量积判断方法

左边绘制静态图形 右边绘制动态图形 可由方向键控制移动

本人原本对碰撞测试不大熟，偶尔在API上见到过hitTestObject()函数，一直都天真的以为此函数能对图形的区域进行准确的碰撞…直到昨天和人聊起这个函数，哪知as3为我们提供的碰撞测试函数hitTestObject()只能检测显示对象的矩形边框（靠！那还检测个毛啊！），另外还有一个函数hitTestPoint()虽然有检测实际区域的能力，可只能进行点的检测，功能实在有限啊~

hitTestObject()其实as3还提供一个非矢量的碰撞检测，BitmapData里有个hitTest函数，确实能实现两个显示对象之间的碰撞检测，不过BitmapData要从draw函数得来，将矢量数据变成位图数据才能处理，对于需要旋转或者变形的碰撞，hitTest并不支持，只能将显示对象变形后重新draw下来，效率可想而知…

综合以上种种~我就想自己写一个碰撞类

这次只是抛砖引玉做一个简单的多边形碰撞测试

更加复杂的碰撞还要靠大家自己去思考实践

碰撞的数学理论基础：

一个碰撞的充分不必要条件：显示对象的实际边界有相交

不必要是因为，当一个显示对象会处在另一个显示对象内部时，边界没有相交而他们确实碰撞了（显示对象实心的话）

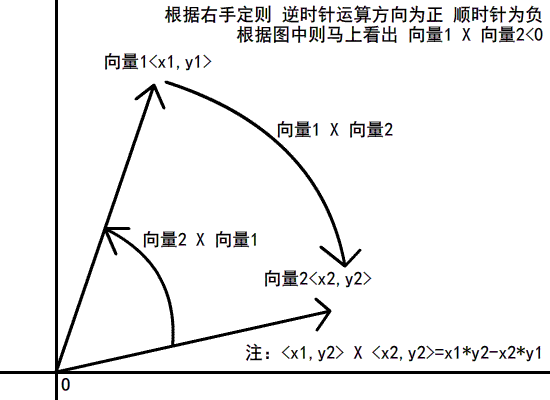
此条件的不必要性导致的问题就是，当物体运动的路径不是连续的，那么一个物体就有可能不通过边界相交，直接穿越到另外一个物体的内部，而碰撞检测程序却完全检测不到，此时碰撞失效了。也就是说，只基于边界相交的碰撞检测依赖于过程。

另外还有个不幸的消息就是，任何的flash动画，物体的运动路径都是不连续的（只可意会不能言传+\_+），不过只要物体每帧改变的位置足够小（相对于物体的尺寸来说,所以在这个示例中，只要你把多边形画得足够小，还是可以穿壳的~），那么基本上不会出现“穿壳”的问题。

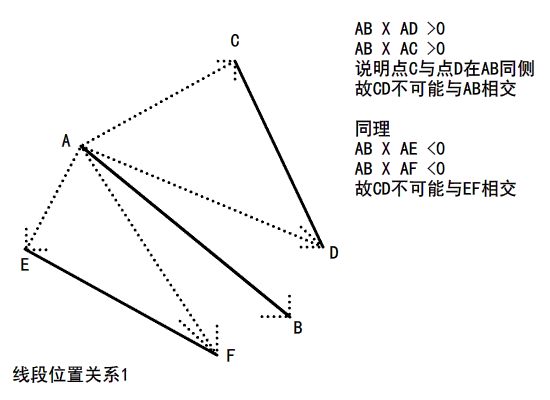
这种基于边界相交检测的碰撞检测，实际上只要再增加一个小小的检测之后，就能解决穿壳到物体内部的问题，处于简单起见，这个问题之做一个小小的提示，有兴趣可以去思考下。

下面先介绍检测两条线段相交的检测方法

首先 应该先了解向量积的计算方法和意义：有两个向量，向量1 <x1,x2> 向量2 <x2,y2>，那么 向量1\*向量2=x1\*y2-x2\*y1=-(x1\*y2-x2\*y1)=-向量2\*向量1。而且 向量1\*向量2 的意义为用右手定则从向量1以不超过180的角度转向向量2，大拇指指向就是 向量1\*向量2 的方向 也就是说如果向量1转向向量2为逆时针，那么他们的向量积就大于0，如果是顺时针，那么就小于0。



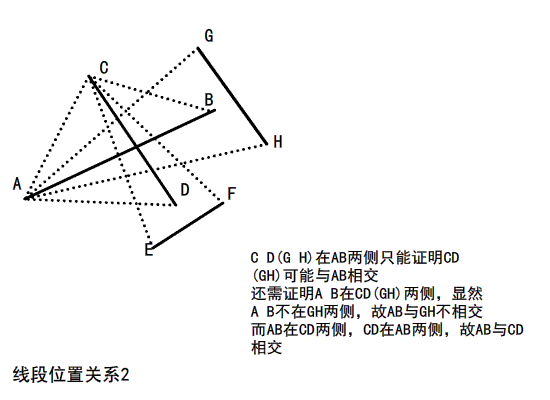
这个原理的意义在于 如果两个点在一个位于原点的向量（暂称原向量）两侧，那么那两个点各自和原点组成的向量必将在原向量的顺时针和逆时针两侧。那么原向量和两个向量分别的向量积必定异号



如果那两个点连成线段（称其新线段），那么原向量所在的直线必然和这个线段相交。

要是将新线段看做一个原向量，得出异号的结论的话，那么我们就能证明，新线段所在的直线和原向量所在的线段（称其旧线段）相交。

既然旧线段所在直线和新线段相交，新线段所在直线和旧线段相交，那么两线段必然相交！



（现在说还是有点抽象~估计有人已经犯困了~或者睡着了Z z z，下面注释程序的时候会更加具体的解释）

一下复制为hitTest.as保存在源文件目录下即可

package {  
import flash.display.Sprite;  
import flash.events.MouseEvent;  
import flash.display.Graphics;  
public class hitTest extends Sprite {  
   public var color:uint;//可以在调用drawShape()之前设置的线条颜色属性  
   private var lineArray:Array;//用来储存多边形每条边的信息的数组  
   private var drawStart:Boolean;//一个判断多边形绘制是否正在进行中的布尔值  
   private var pointN:int;//在多边形绘制中用来计数正在绘制的点的序数  
   public function hitTest():void {//构造函数 对必要的变量进行赋初值  
    lineArray=new Array();  
    color=0x0000FF;  
    this.x=0;  
    this.y=0;  
   }  
   public function init():void {//初始化函数 写这个函数的目的是不必对对象进行重新构造 而对它进行归零操作。对于像能被添加到显示列表上的的一些对象，由于其被添加添加到显示列表的时候，显示列表会创建一个对象引用的副本，如果要彻底删除这个对象，重新建立一个对象，必须先将其从显示列表删除，并保证任何一个应用都不指向这个对象，然后构建新对象时，这个对象才可能从内存删除（这个是FP的事了~），所以与其花那么多力气删除对象，还不如直接写个归零函数将其归零来的方便。  
    lineArray=new Array();//对数组进行归零  
    drawStart=false;//对布尔值进行归零  
    this.graphics.clear();//清除所有graphics的绘图  
    this.x=0;  
    this.y=0;//对位置进行归零  
   }  
   public function drawShape():void {//调用此函数开始绘制多变形  
    lineArray=new Array();  
    drawStart=false;  
    this.x=0;  
    this.y=0;  
    this.graphics.clear();//以上语句和init()中一样，直接调用init()也可以。增加以上语句的目的是当才重复调用此函数时，保证舞台上只有一个多边形。  
    this.graphics.lineStyle(1,color,1);//设置多边形的边界线条样式  
    parent.addEventListener(MouseEvent.CLICK,drawLine);//当该对象的母容器被点击时，调用drawLine真正开始绘制边界了  
   }  
   private function drawLine(e:MouseEvent):void {  
    if (drawStart) {//当此次点击是本次多边形绘制的第一次点击，那么drawStart将为false，如果非第一次点击，那么将执行下面代码  
     this.graphics.lineTo(e.stageX,e.stageY);//直接从上次绘制结束的点或者是moveTo的点绘制到当前鼠标的位置  
     pointN++;//绘制的点加1  
     lineArray[pointN]={a:e.stageX,b:e.stageY,c:0,d:0};//创建一条线的数据，并储存线的开始点  
     lineArray[pointN-1].c=e.stageX;  
     lineArray[pointN-1].d=e.stageY;//把这个点储存到上一条线的结束点  
    } else {  
     this.graphics.moveTo(e.stageX,e.stageY);//如果绘制的是第一个点，那么把会知道移动到鼠标位置  
     pointN=0;//绘制的点归零  
     lineArray[pointN]={a:e.stageX,b:e.stageY,c:0,d:0};//创建一条线的数据，然后储存开始点  
     drawStart=true;//标记已开始绘制  
    }  
   }  
   public function endDrawShape():void {//结束绘制函数  
    drawStart=false;//标记结束绘制  
    parent.removeEventListener(MouseEvent.CLICK,drawLine);  
    if (lineArray[0]!=null) {//存在至少一条数据  
     this.graphics.lineTo(lineArray[0].a,lineArray[0].b);//从最后一个绘制点画一条线到第一个点  
     lineArray[pointN].c=lineArray[0].a;  
     lineArray[pointN].d=lineArray[0].b;//最后一条线的数据的结束点为开始点  
    }  
   }  
   private function simpleLineTest(l1p1x:Number,l1p1y:Number,l1p2x:Number,l1p2y:Number,l2p1x:Number,l2p1y:Number,l2p2x:Number,l2p2y:Number):Boolean {//这个函数是核心，它检测两条线是否相交，每条线有四项数据（开始点的两个坐标，结束点的两个坐标），所以两条线总共需要8个参数  
    var line1p1:Number;  
    line1p1=(l1p2x-l1p1x)\*(l2p1y-l1p1y)-(l2p1x-l1p1x)\*(l1p2y-l1p1y);//第一条线段的向量和（第一条线段的开始点与第二条线段的开始点组成的向量）的向量积  
    var line1p2:Number;  
    line1p2=(l1p2x-l1p1x)\*(l2p2y-l1p1y)-(l2p2x-l1p1x)\*(l1p2y-l1p1y);//第一条线段的向量和（第一条线段的开始点与第二条线段的结束点组成的向量）的向量积  
    var line2p1:Number;  
    line2p1=(l2p2x-l2p1x)\*(l1p1y-l2p1y)-(l1p1x-l2p1x)\*(l2p2y-l2p1y);//第二条线段的向量和（第二条线段的开始点与第一条线段的开始点组成的向量）的向量积  
    var line2p2:Number;  
    line2p2=(l2p2x-l2p1x)\*(l1p2y-l2p1y)-(l1p2x-l2p1x)\*(l2p2y-l2p1y);//第二条线段的向量和（第二条线段的开始点与第一条线段的结束点组成的向量）的向量积  
    if ((line1p1\*line1p2<=0)&&(line2p1\*line2p2<=0)) {//判断方法在先前讲过  
     return true;//相交  
    } else {  
     return false;//否则不相交  
    }  
   }  
   public function lineTest(la:hitTest):Boolean {//判断另外一个对象与本对象是否有任何一条线相交 这个也就是碰撞检测了  
    for (var n:int=0; n<la.lineArray.length; n++) {  
     for (var m:int=0; m<lineArray.length; m++) {//两个for循环嵌套，对两个对象的所有线段进行遍历检测，一旦发现有相交，就返回true  
      if (simpleLineTest(lineArray[m].a+this.x,lineArray[m].b+this.y,lineArray[m].c+this.x,lineArray[m].d+this.y,la.lineArray[n].a+la.x,la.lineArray[n].b+la.y,la.lineArray[n].c+la.x,la.lineArray[n].d+la.y)) {  
       return true;  
      }  
     }  
    }  
    return false;  
   }  
}  
}

主程序的第一帧代码：

//bg是舞台背景，还有四个按钮已经定义好的按钮

import hitTest;//吧刚才写的类导入  
var up:Boolean=false;  
var down:Boolean=false;  
var left:Boolean=false;  
var right:Boolean=false;//这四个变量是控制第二个多边形的四个方向的运动的，初始false，就是不运动  
var aTest:hitTest=new hitTest();  
var bTest:hitTest=new hitTest();//建立两个空的碰撞检测对象  
bg.addChild(aTest);  
bg.addChild(bTest);//把两个对象都添加到bg上  
aTest.color=0x00FF00;//以一个多边形的线条颜色设为绿色  
stage.addEventListener(KeyboardEvent.KEY\_DOWN,setDown);//键盘按下发生事件  
stage.addEventListener(KeyboardEvent.KEY\_UP,setUp);//键盘起来 发生事件  
stage.addEventListener(Event.ENTER\_FRAME,mov);//让第二个多边形移动  
drawA.addEventListener(MouseEvent.CLICK,drawANow);//开始绘制第一个多边形  
drawB.addEventListener(MouseEvent.CLICK,drawBNow);//开始绘制第二个多边形  
stopA.addEventListener(MouseEvent.CLICK,stopANow);//停止绘制第一个多边形  
stopB.addEventListener(MouseEvent.CLICK,stopBNow);//停止绘制第二个多边形  
var lsx:Number=0;  
var lsy:Number=0;//这两个变量是用来储存第二个多边形移动后的上一步的位置  
function drawANow(e:MouseEvent):void {//此函数调用第一个多边形的绘制图形函数  
aTest.drawShape();  
}  
function drawBNow(e:MouseEvent):void {//此函数调用第二个多边形的绘制图形函数  
bTest.drawShape();  
}  
function stopANow(e:MouseEvent):void {//此函数调用第一个多边形的停止绘制函数  
aTest.endDrawShape();  
}  
function stopBNow(e:MouseEvent):void {//此函数调用第二个多边形的停止绘制函数  
bTest.endDrawShape();  
}  
function setDown(e:KeyboardEvent):void {//键盘按下  
if (e.keyCode==Keyboard.DOWN) {//如果是反向键向下  
   down=true;//则允许向下运动  
}  
if (e.keyCode==Keyboard.UP) {//如果方向键向上  
   up=true;//允许向上运动  
}  
if (e.keyCode==Keyboard.LEFT) {//如果方向键向左  
   left=true;//允许向左运动  
}  
if (e.keyCode==Keyboard.RIGHT) {//如果方向键向右  
   right=true;//允许向右运动  
}  
}  
function setUp(e:KeyboardEvent):void {//原理同上 就是哪个键放开 哪个方向的运动就不允许  
if (e.keyCode==Keyboard.DOWN) {  
   down=false;  
}  
if (e.keyCode==Keyboard.UP) {  
   up=false;  
}  
if (e.keyCode==Keyboard.LEFT) {  
   left=false;  
}  
if (e.keyCode==Keyboard.RIGHT) {  
   right=false;  
}  
}  
function mov(e:Event):void {//根据四个方向的布尔值和碰撞检测的结果运动，实现碰撞的运动效果  
if (left==true) {//如果能向做运动  
   bTest.x--;//位置向左  
} else if (right==true) {//如果向左不成立，切且能向右  
   bTest.x++;//向右  
}  
if (aTest.lineTest(bTest)) {//运动完成后碰撞  
   bTest.x=lsx;//返回运动前的位置  
} else {  
   lsx=bTest.x;//没碰撞的话，保存当前位置  
}  
if (up==true) {//一下同理  
   bTest.y--;  
} else if (down==true) {  
   bTest.y++;  
}  
if (aTest.lineTest(bTest)) {  
   bTest.y=lsy;  
} else {  
   lsy=bTest.y;  
}  
}