Introducing Starling – bytearray.org – revision 1.2.6

Starling 框架帮助手册中文版

bytearray.org 出品
S_eVent 翻译

目录

什么是 Statling?

Starling 是一个基于 Stage3D（这是 Flash Player11 及 Adobe AIR 3 中新增的为 3D 加速功能所提供的 API）所开发的一个能够使用 GPU 来加速的 2D Flash 应用程序的 ActionScript3 框架。Starling 主要是为游戏开发而设计的, 但是它的用途不仅限于此。Starling 最大的好处在于你可以很快地写出使用 GPU 加速的应用程序而不必接触那些复杂的底层 Stage3D API。

为什么选择 Starling?

绝大多数的 Flash 开发人员都想很轻松地在自己的开发中用上 GPU 的加速功能而不必自己去钻研那些复杂的底层 Stage3D API 并自己写一套框架出来。Starling 是一个完全基于 Flash Player 的 API 开发的框架，它将复杂的 Stage3D（官方称之为 Molehill） API 抽象、封装起来并允许开发者能够非常轻松、直观地使用 Stage3D 的功能。

很显然，Starling 是为那些 ActionScript3 的开发人员，特别是那些开发 2D 游戏的人服务的，当然，在使用它之前你必须得对 ActionScript 3 有一个基本的了解才行。介于 Starling 轻量级、灵活性高以及易于使用的特点，它可以被用到其他的一些方面，如 UI（可视化组件）的制作上去。可以说，使用 Starling 写的代码是尽可能地直观易懂，即使是 Java 或.NET 开发人员也能够很快地掌握其中奥秘。

Starling 的特色

直观

学会 Starling 的使用方法是一件非常容易做到的事情。 特别是那些 Flash/Flex 的开发人员，他们会觉得使用 Starling 就跟从事老本行（原生 AS3 API）几乎没有什么差别，因为 Starling 的 API 命名规范与原生 AS3 API 极其相似，Starling 中充斥着我们所非常熟悉的东西，如显示列表、 事件模型以及一些熟悉的 API 比如 MovieClip，Sprite，TextField 等等。如果你要用底层 Stage3D API 来开发，那你就不得不接触一些陌生且复杂的概念，如 vertices buffer（顶点缓冲），perspective matrices（视角矩阵），shader programs（着色器编程） and assembly bytecode（字节码装配）【PS：这些名词都是直译，若翻译不正确还请见谅】

轻量级

Starling 的轻量级特点表现在众多方面：它的类数量不多（也就 80K 左右的代码量）；对除 Flash Player 11 或 AIR 3 之外没有额外的依赖性【PS：意思就是说你在开发项目时不需要导入额外的软件开发包或 swc 之类的东东，你所需要的只是导入一个不大的 Starling 框架的 swc

即可】(Starling 将在未来的版本中加入对手机应用开发的支持)。这些特点可以让你所开发的应用保持一个较小的空间占用且让你拥有一个简便的工作流。

免费

Starling 是一个免费但保持不断更新的框架。它通过了 Simplified BSD 标准认证，所以你可以免费地使用它，甚至在商业应用中也一样。我们开发团队每天都在不断地为让此框架更好地发展而不懈努力着，且我们有一个活跃的开发者交流社区，在这里我们可以听取一些意见和建议且不时会有开发者为 Starling 框架提供一些额外插件神马的，这些都让我们亲爱的 Starling 框架保持着充沛活力。

Starling 是怎样工作的

图 1.1 显示了 Starling 是怎样来驱动 GPU 的，由于 Starling 是基于 Stage3D 的 API 开发的，所以它的驱动关系位于 Stage3D 之上，而 Stage3D 有能力去控制 OpenGL、DirectX 这些电脑显卡驱动或 OpenGLES2 这些手机显卡驱动，最终由这些显卡驱动程序去驱动 GPU 工作。作为一个开发人员，还有一点你需要知道，那就是 Starling 是 Sparrow 框架（http://www.sparrow-framework.org，一个 iOS 的开发框架，它依赖于 OpenGL ES2 的 API）在 ActionScript3 领域的一个姐妹框架。【PS：你只需要知道 Starling 和 Sparrow 有基友关系就行了，在后面使用一些素材制作工具时有用】

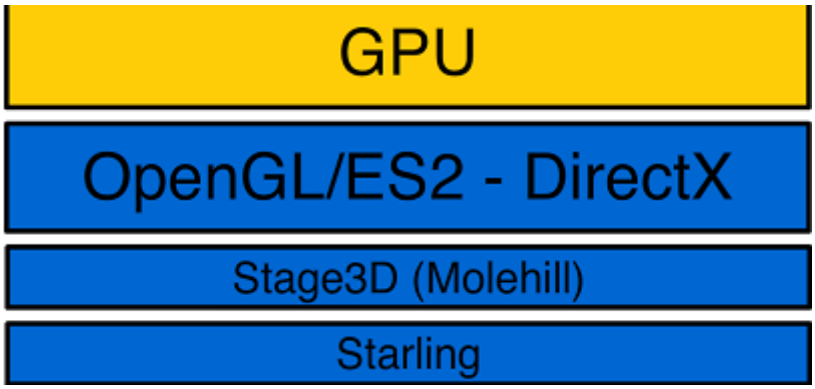

图 1.1
驱动关系图。Starling 位于 Stage3D（Molehill）之上

Starling 重建了许多 Flash 开发人员已经非常熟悉的 API，下面是一张与图形元素有关的 Starling API 树形图。

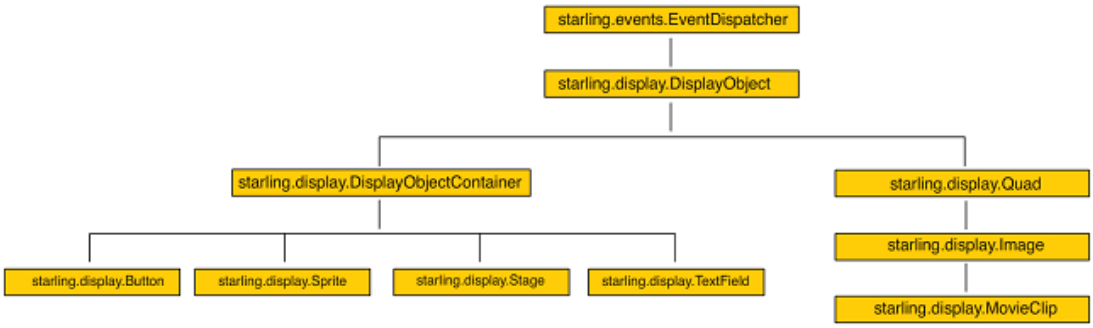

图 1.2

Starling 中的 DisplayObject 继承关系

可能会有很多人产生疑惑，Stage3D 应该只是为渲染 3D 内容而存在的啊，它应该如何渲染 2D 的内容呢？我们使用下面的插图来表现我们的这个疑问，我们该如何使用 Stage3D 中的 drawTriangles 这个 API 来绘制类似 MovieClip 一样的东西呢？

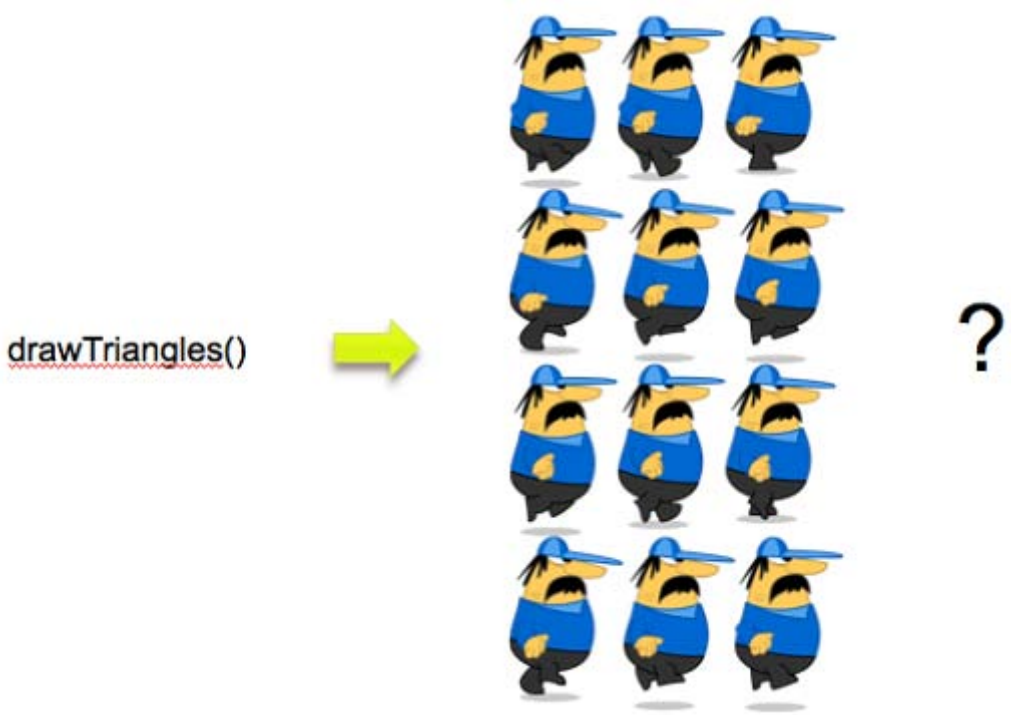

图 1.3

怎样用 drawTriangles 来画 2D 内容呢？

事实上，答案很简单。GPU 的优势在于它绘制三角形极其高效，所以使用 drawTriangles API 将绘制出两个三角形，之后我们再从一个纹理（texture，你可以理解成“素材”的意思）上面采样（或取色）并使用一种称为 UV 映射的机制来着色于这两个三角形上即可，最后我们将能够得到一个纹理化（textured，即着了色的）的方块，我们就用它来作为我们 Sprite 对象的外观。若是在每一帧都改变其纹理，我们就可以得到一个 MovieClip 对象了，很酷吧？【PS：酷你妹！】

好消息是，在 Starling 里面，我们不必过多地关注这些细节了，我们要做的只是提供 MovieClip 每一帧的纹理素材给 Starling MovieClip 即可。怎样？是不是觉得非常简单啊？你颤抖了吗少

年？！【PS：这些话明显是译者加上去的 =。=】

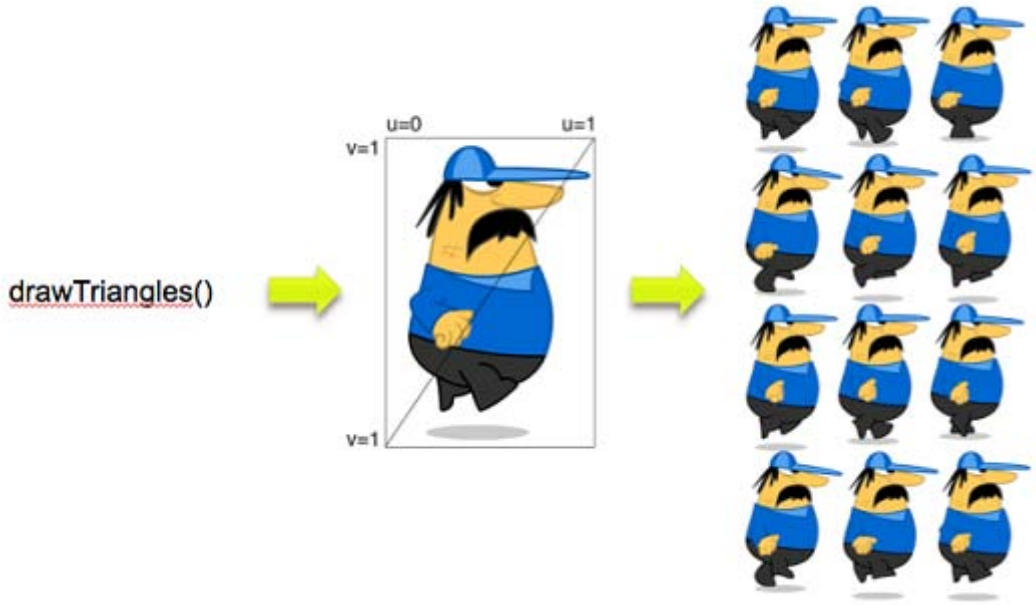

图 1.4
drawTriangles + textured quad（纹理化方块）= 2D 内容

为了更加直观地展示 Starling 简化底层 Stage3D API 的能力，我们接下来一起来看一个例子，下面的例子中我们需要绘制一个纹理化的方块。先看看使用底层 Stage3D 的 API 如何实现之。【PS：下面的代码，有每一行都研究过啥意思的同学请举手，我给你们颁奖状，赐予“爱学习的好孩子”殊荣】

```
// create the vertices
var vertices:Vector.<Number> = Vector.<Number>([
-0.5,-0.5,0, 0, 0, // x, y, z, u, v
-0.5, 0.5, 0, 0, 1,
0.5, 0.5, 0, 1, 1,
0.5, -0.5, 0, 1, 0]);

// create the buffer to upload the vertices
var vertexbuffer:VertexBuffer3D = context3D.createVertexBuffer(4, 5);

// upload the vertices
vertexbuffer.uploadFromVector(vertices, 0, 4);

// create the buffer to upload the indices
var indexbuffer:IndexBuffer3D = context3D.createIndexBuffer(6);

// upload the indices
indexbuffer.uploadFromVector (Vector.<uint>([0, 1, 2, 2, 3, 0]), 0, 6);

// create the bitmap texture
```

```
var bitmap:Bitmap = new TextureBitmap();

// create the texture bitmap to upload the bitmap
var texture:Texture = context3D.createTexture(bitmap.bitmapData.width,

bitmap.bitmapData.height, Context3DTextureFormat.BGRA, false);

// upload the bitmap
texture.uploadFromBitmapData(bitmap.bitmapData);

// create the mini assembler
var vertexShaderAssembler : AGALMiniAssembler = new AGALMiniAssembler();

// assemble the vertex shader
vertexShaderAssembler.assemble( Context3DProgramType.VERTEX,
"m44 op, va0, vc0\n" + // pos to clipspace
"mov v0, va1" // copy uv
);

// assemble the fragment shader
fragmentShaderAssembler.assemble( Context3DProgramType.FRAGMENT,
"tex ft1, v0, fs0 <2d,linear, nomip>;\n" +
"mov oc, ft1"
);

// create the shader program
var program:Program3D = context3D.createProgram();

// upload the vertex and fragment shaders
program.upload( vertexShaderAssembler.agalcode, fragmentShaderAssembler.agalcode);

// clear the buffer
context3D.clear ( 1, 1, 1, 1 );

// set the vertex buffer
context3D.setVertexBufferAt(0, vertexbuffer, 0, Context3DVertexBufferFormat.FLOAT_3);
context3D.setVertexBufferAt(1, vertexbuffer, 3, Context3DVertexBufferFormat.FLOAT_2);

// set the texture
context3D.setTextureAt( 0, texture );

// set the shaders program
context3D.setProgram( program );
```

```
// create a 3D matrix
var m:Matrix3D = new Matrix3D();

// apply rotation to the matrix to rotate vertices along the Z axis
m.appendRotation(getTimer()/50, Vector3D.Z_AXIS);

// set the program constants (matrix here)
context3D.setProgramConstantsFromMatrix(Co
ntext3DProgramType.VERTEX, 0, m, true);

// draw the triangles
context3D.drawTriangles( indexBuffer);

// present the pixels to the screen
context3D.present();
```

运行此段代码，你可以看到下图所示的结果：

图 1.5
一个简单的纹理化方块

很复杂对吧？不管你是否这样觉得，我反正是这样觉得了。这就是直接使用底层 API 的代价啊，虽然你可以控制更多的细节……

但是如果你使用了 Starling 你只需要下列几句代码即可做到：

```
// create a Texture object out of an embedded bitmap
var texture:Texture = Texture.fromBitmap ( new embeddedBitmap() );

// create an Image object our of the Texture
var image:Image = new Image(texture);

// set the properties
image.pivotX = 50;
image.pivotY = 50;
```

```
image.x = 300;
image.y = 150;
image.rotation = Math.PI/4;

// display it
addChild(image);
```

若你是一个 ActionScript 开发人员，你会觉得上面的代码非常熟悉，那感觉就像回到了家一样。要知道，Starling 已经将那些复杂的底层编码都在后台为你完成掉了。
如果你尝试着在 Starling 创建的 Flash 应用中显示重绘区域，你会看到 Starling 渲染出来的内容中并不会出现红框框，虽然下面这个方块不停地在旋转着，但是就是看不见有重绘区域存在其上，我们只能看见左上角那个显示 FPS 的显示对象存在着重绘区域的红框框（因为这个显示对象是添加在原生 Flash 显示列表中，而不是 Stage3D 显示列表中的）

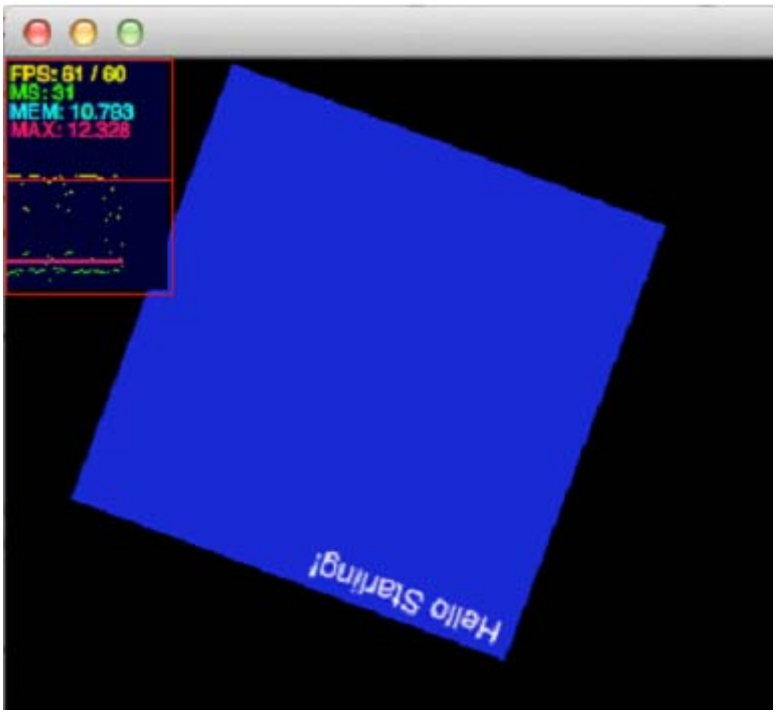

图 1.6
通过 Stage3D 渲染出来的内容

记住，在 Stage 3D 的显示列表中，渲染、混合等操作已全权由 GPU 来负责，因此，显示重绘区域，这个用作 CPU 渲染的显示列表的功能将无法正常运作了。

显示层次限制

作为一个开发人员，在使用 Starling 来开发 Stage3D 应用时，有一个限制因素值得注意。这在之前也提到过一下，就是 Stage3D，光从名字上就能看出它不同于 Flash 以往所使用的 Stage，它是 Flash Player 中出现的一个全新的渲染机制（传统 Stage 使用 CPU 渲染，而 Stage3D 使用 GPU 渲染）。而 GPU 层的显示层次是位于传统显示列表之下的，这意味着被添加到传统显示列表中的内容将始终会遮挡住 Stage3D 中的内容（如果它们位置存在重叠的话）。下图

说明了 Flash Player 中的显示层次结构：

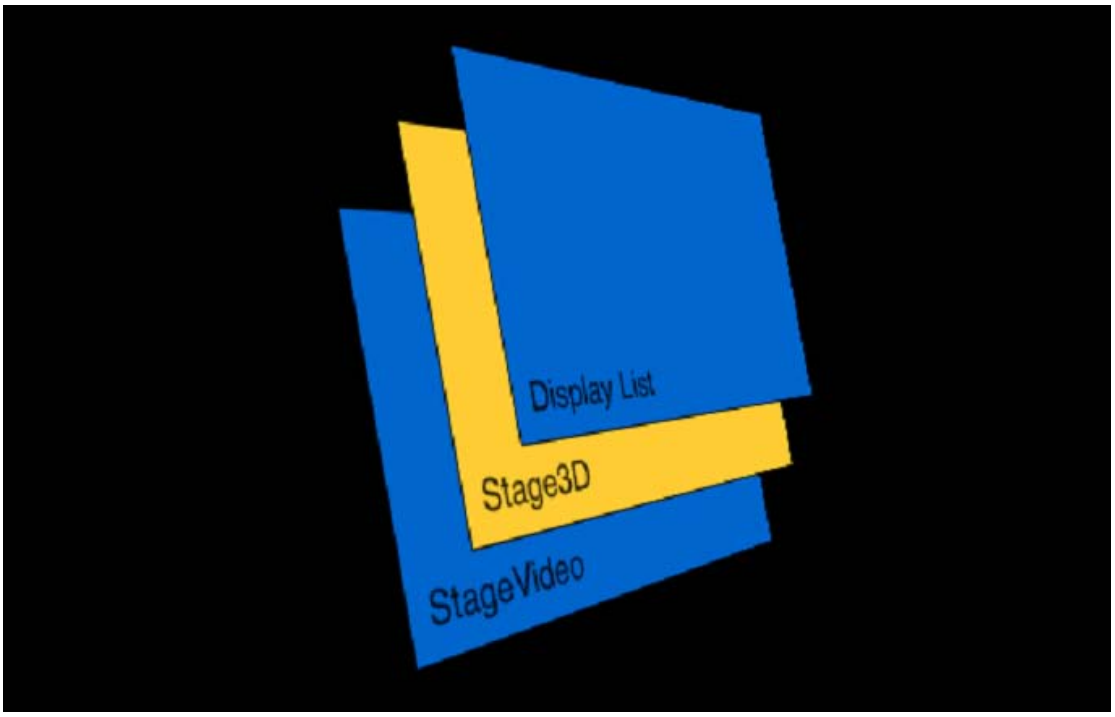

图 1.7

传统显示列表、Stage3D 及 StageVideo 的层次结构

需要注意的是，Stage3D对象是无法被设置成透明的，这个特性允许开发人员运用StageVideo技术来播放视频（在Flash Player 10.2 http://www.adobe.com/devnet/flashplayer/articles/stage_video.html 这篇文章中有介绍过）。且StageVideo播放的视频内容将会叠在那些位于Stage3D层中播放着的视频之上【PS：这一点让我有点困惑，根据上面这个层次图来看明明是StageVideo层在下面的】。上面介绍的这个特性可能会在将来发布的Flash Player以及AIR版本中出现。

让我们开始上手吧

首先，你可以在下面给出的连接中下载得到 Starling：

* Official Starling Github: http://github.com/PrimaryFeather/Starling-Framework/
* Official Starling website: http://www.starling-framework.org

再次重申一遍，Starling 是通过了 simplified BSD 标准认证的，所以你可以在各种类型、各种用途的项目中使用它。你也可以联系 Starling 的幕后开发团队哦，邮箱地址是 office@starling-framework.org

当 Starling 的压缩包被下载完毕后，你就可以在任何 AS3 项目中引用它了（直接拷贝项目源码到项目中或者把其提供的 SWC 包含到项目中）。之后，你还需要搭建 Stage3D 所必要的运行环境：

- 下载最新版本的为 Flash Player11 准备的 playerglobal.swc 文件
  【PS：我找到的下载地址是：http://www.adobe.com/support/flashplayer/downloads.html#fp11】
- 下载 Flex SDK4.5 或以上版本
- 更新 playerglobal.swc 到 IDE 中：找到当前你所使用的 IDE（比如我用的是 Flash Builder）

安装目录，进入到

项目安装目录 → sdks → 当前所用 FlexSDK 版本号 → frameworks\libs\player\11.0（若不存在 11.0 目录就创建之）

把下载了的最新版本的 playerglobal.swc 文件复制到 11.0 目录中

- 打开 IDE 并建立一个新项目，调整项目属性，使项目使用 Flex SDK4.5 或以上版本的 SDK；将需求的 Flash Player 最低版本调至 11.0 或以上；为项目的编译器参数中增加一句“-swf-version=13”
- 确保你安装了 Flash Player11.0 或以上版本（可以是独立版本的也可以是浏览器插件版，不过对于浏览器版本的 FP，你需要修改 swf 文件所嵌入到的 html 文件，将 swf 嵌入的 wmode 参数改成’direct’才行，这个稍后会详谈）

构建场景

对于 Starling，大家已经有了一定的了解了，接下来让我们来学着写一些代码，看看 Starling 到底应该怎么用，它到底能做些什么。Starling 非常容易安装，你只需要创建一个 Starling 对象的实例到你的文档类中即可。

到目前为止当提到 MovieClip，Sprite 等对象的时候我们其实是在说 Starling 中提供的 API (Starling.display.Sprite)而不是原生 Flash Player 中的那些(flash.display.Sprite)

首先，让我们先看看 Starling 的构造函数，它期待多个参数：

```
public function Starling(rootClass:Class, stage:flash.display.Stage,
                         viewPort:Rectangle=null, stage3D:Stage3D=null,
                         renderMode:String="auto")
```

事实上，只有前两个参数是经常用到的。rootClass 参数期待被传入一个继承自 starling.display.Sprite 的类引用（不是对象），这个类引用将作为 Starling 的入口类，即 Starling 中的文档类。第二个参数，一看该参数的类型就知道一定得传入原生 Flash 的舞台了，由 Starling 创建的 Stage3D 对象将位于此 stage 对象的下一层。接下来一起看一段启动 Starling 的试验代码。

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import starling.core.Starling;

    [SWF(width="1280", height="752", frameRate="60", backgroundColor="#002143")]
    public class Startup extends Sprite
    {
        private var mStarling:Starling;

        public function Startup()
        {
            // stats class for fps
```

```
        addChild ( new Stats() );

        stage.align = StageAlign.TOP_LEFT;
        stage.scaleMode = StageScaleMode.NO_SCALE;

        // create our Starling instance
                mStarling = new Starling(Game, stage);

                // set anti-aliasing (higher the better quality but slower performance)
                mStarling.antiAliasing = 1;

                // start it!
        mStarling.start();
           }
    }
}
```

【PS：Stats 类是一个显示 FPS 的类，可以在网上下载得到】

接下来，我们创建一个 Game 类来作为 Starling 入口类，在 Game 类中我们只是创建一个简单的方块并添加到显示列表中来。

```
package
{
    import starling.display.Quad;
    import starling.display.Sprite;
    import starling.events.Event;

    public class Game extends Sprite
    {
        private var q:Quad;

            public function Game()
            {
        addEventListener(Event.ADDED_TO_STAGE, onAdded);
            }

    private function onAdded ( e:Event ):void
    {
        q = new Quad(200, 200);
        q.setVertexColor(0, 0x000000);
        q.setVertexColor(1, 0xAA0000);
        q.setVertexColor(2, 0x00FF00);
        q.setVertexColor(3, 0x0000FF);
        addChild ( q );
    }
```

```
    }
}
```

就像我们以前经常会做的一样，侦听 Event.ADDED_TO_STAGE 事件并在事件处理函数中做一些初始化工作，把初始化工作放在事件处理函数中进行的话就不用担心会因 stage 对象为空而报错了。【PS：注意，此处的 Event 不是我们以往用的 flash.event.Event，而是 starling.events.Event】

再次提醒各位注意这个细节，我们这里创建的 Game 类是继承自 starling.display 包中的 Sprite，而不是 flash.display 包中的那个。在使用 Starling 框架进行编码的时候你最好时刻检查你导入的应是 starling 的包，而非原生 flash 中的那些包。你可能在一开始会偶尔搞混，但是很快你就会习惯了

和原生的 Flash 中的显示列表一样，Starling 中的显示对象默认位置也在 0,0 点，为了让我们的这个小应用更加好看，我们再来写以下两句代码把方块居中至舞台中央位置：

```
q.x = stage.stageWidth - q.width >> 1;
q.y = stage.stageHeight - q.height >> 1;
```

这里提一下，向右位移一位操作(>>1)等价于将一个数字除以 2，它们运算得到的结果是一样的（只不过移位操作运算速度更快一些罢了），下图是我们运行代码得到的结果：

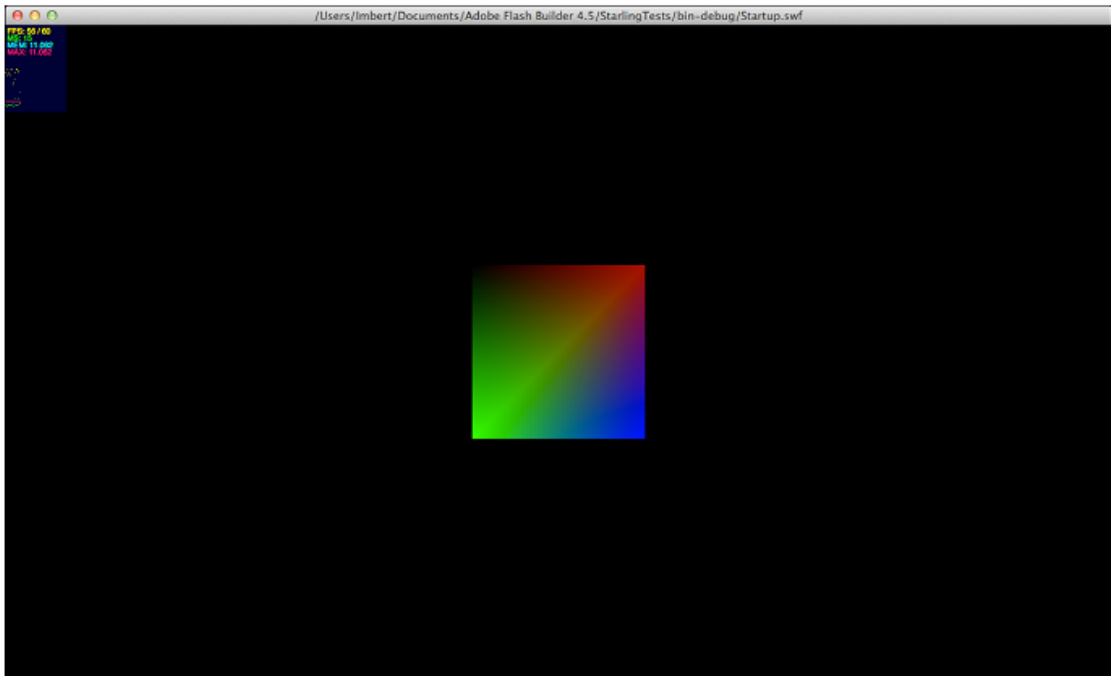

图 1.8

我们创建的第一个方块哦！

在一个 Starling 对象中，你可以通过设置其 antialiasing 属性的值来达到在渲染时使用某种程度的抗锯齿处理，你可以为此属性设置 1-16 的值，不过我们通常会设置其值为 1，这样的渲染效果已经算不错了。下面列出了一些常用的设置值：

* 0 : 无抗锯齿
* 2 : 较低程度的抗锯齿

* 4 : 高质量抗锯齿
* 16 : 极高质量抗锯齿

一般我们不会设置该属性为超过 2 的值，特别是对于 2D 内容来说，必要性不高，不过你可以自己决定是否启用较高级别的抗锯齿。下图显示了两个抗锯齿值分别设置为 1 和 4 的不同：

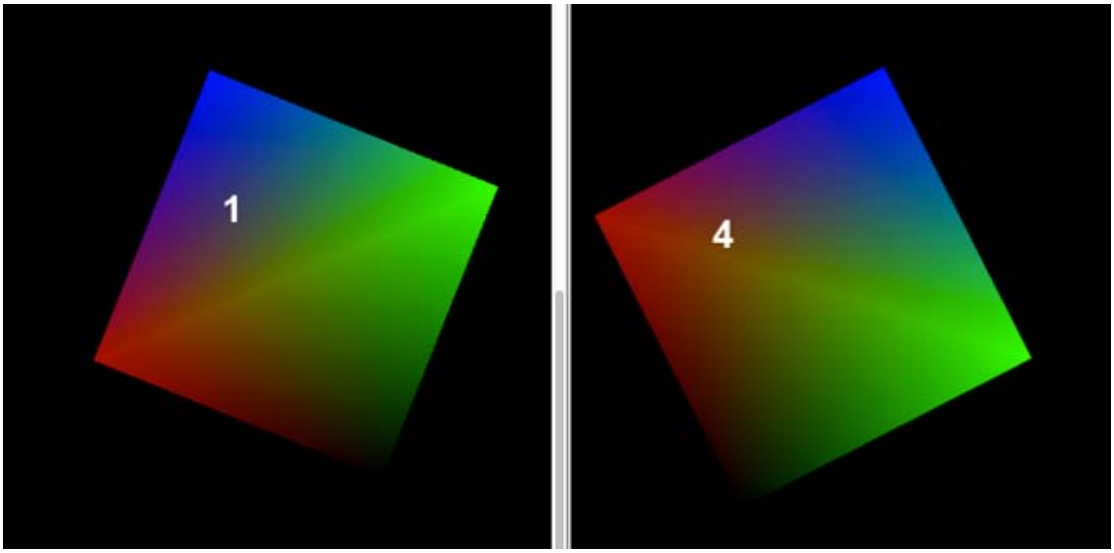

图 1.9
抗锯齿设为不同值后得到的效果

【PS：说实话，我并没有看出这两个图有任何不同之处，可能抗锯齿这个功能在渲染 3D 内容时比较有效吧】

你可以根据你的需要设置更高的抗锯齿质量，但是质量越高越是会消耗性能。

接下来让我们一起看看 Starling 对象的其他一些 API：

* **enableErrorChecking**：允许你启用或禁用错误检测功能。启用此功能后 Starling 将会把渲染器在工作中遇到的错误报告给你的应用程序并输出在控制台上。当 enableErrorChecking 属性为 true 时，Starling 将在内部同步地调用 clear()及 drawTriangles() 方法并可以抛出错误（如果遇到的话）。若此属性为 false（默认值），clear()及 drawTriangles()方法将会被异步地调用且在遇到错误时也不会报告。启用错误检测功能将会降低性能，所以你最好只在调试的时候启用一下。
* **isStarted** ：一个标记，用于指示 start 方法是否被调用过。
* **juggler**： juggler 是一个用于存储一系列实现了 IAnimatable 接口的对象的对象，你可以通过调用 juggler 对象的 advanceTime 方法来让它调度其保存的那些实现 IAnimatable 接口的对象每一帧的行为。当一个动画播放完毕后，juggler 会将其剔除。

【PS：你可以把 juggler 对象作为一个动画管理器，其详细功能稍后会介绍】

* **start**：开始渲染并处理事件
* **stop**：停止渲染和事件处理，你可以在游戏被置于后台（如将 Flash 最小化时）时使用此方法来停止渲染，这样可以避免硬件开销上的浪费。
* **dispose**：当你想要销毁 Starling 中全部的渲染内容并释放其在 GPU 中所占用空间时你可以调用此方法。这个 API 将会在内部同时销毁着色器程序及事件侦听器（GPU 用于绘图的东东）。

一旦当你的 Starling 对象被创建之后，有一条记录着当前使用渲染器（一般指显卡驱动程序）

的调试信息将会自动输出到你的控制台中。默认情况下，当一个 SWF 文件被嵌入到网页中时或者被独立的 Flash Player 播放的时候，Starling 会输出类似于下面的信息：

```
[Starling] Initialization complete.
[Starling] Display Driver:OpenGL Vendor=NVIDIA Corporation Version=2.1 NVIDIA-7.2.9 Renderer=NVIDIA GeForce GT
330M OpenGL Engine GLSL=1.20 (Direct blitting)
```

当然，输出的这些硬件信息将会取决于你机器的配置。这条信息将提醒我们当前运行的 flash 应用是使用哪些显卡驱动来进行 GPU 加速的。不过有时候，出于调试的目的，我们可能会想让 Starling 用回软件渲染（即 CPU 渲染）来调试我们的 Flash 应用程序在无法使用硬件加速情况下的性能。

为了做到这一点，我们只需要简单地设置 Starling 对象的第五个参数即可。

```
mStarling = new Starling(Game, stage, null, null, Context3DRenderMode.SOFTWARE);
```

当 Starling 使用软件渲染时，类似下面的信息将会输出到控制台上：

```
[Starling] Initialization complete.
[Starling] Display Driver:Software (Direct blitting)
```

之前说过，当你的硬件条件不满足硬件渲染条件时（一般来说是显卡驱动发布日期早于 2009 年 1 月 1 日），Starling 将会自动切换为软件渲染模式。

现在，我们来看看，当我们把一个 swf 文件嵌入网页时，若要让它用上 Stage3D 需要满足什么条件。

Wmode

你必须记得，若你想让一个嵌入在网页中的 swf 用上 Stage3D 来进行 GPU 加速的话，你就必须修改 html 源代码，使 wmode = direct（wmode 是在 html 中嵌入 swf 的一个可选参数）。若你在嵌入 swf 时未设置 wmode 参数或设置为其他值，如“transparent”, “opaque” 或“window”，Stage3D 将会不可用，此时 Flash Player 会抛出一个运行时异常来提醒你 Context 3D 不可用。

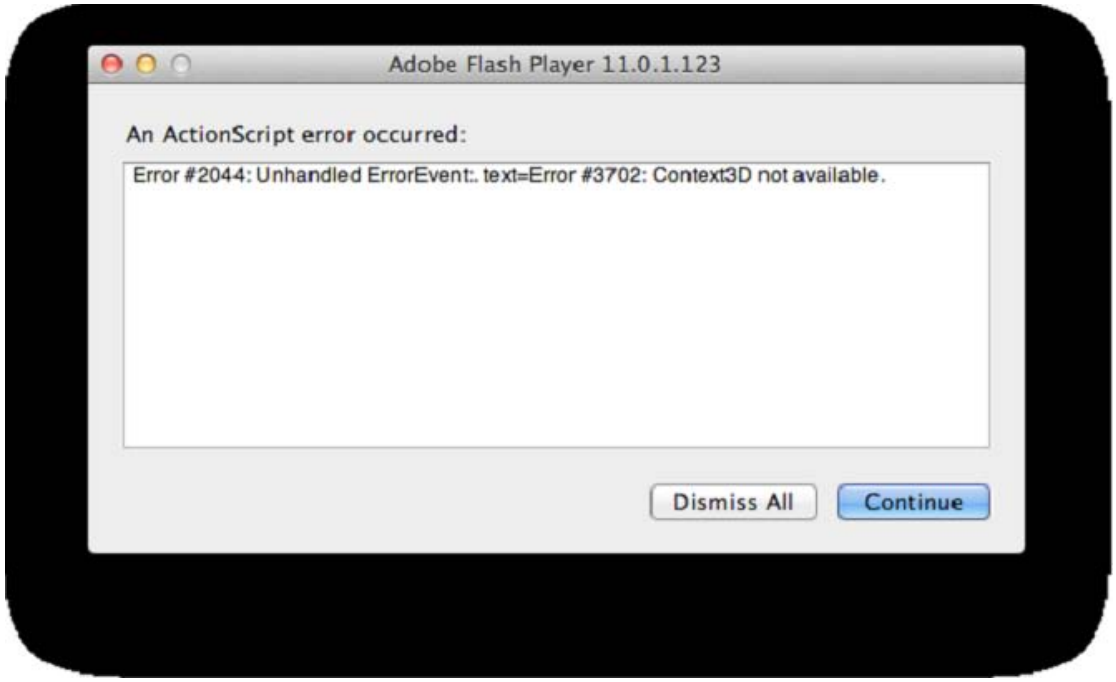

图 1.10
当 Context3D 不可用时 FP 将抛出的运行时异常

一般当看见此种异常时我们很难明白到底是哪里出了差错，不过幸运的是，Starling 在内部已经为我们做了正确的判断，当此异常确实是 wmode 设置错误引起的话，Starling 会在屏幕上打印以下信息以通知我们：

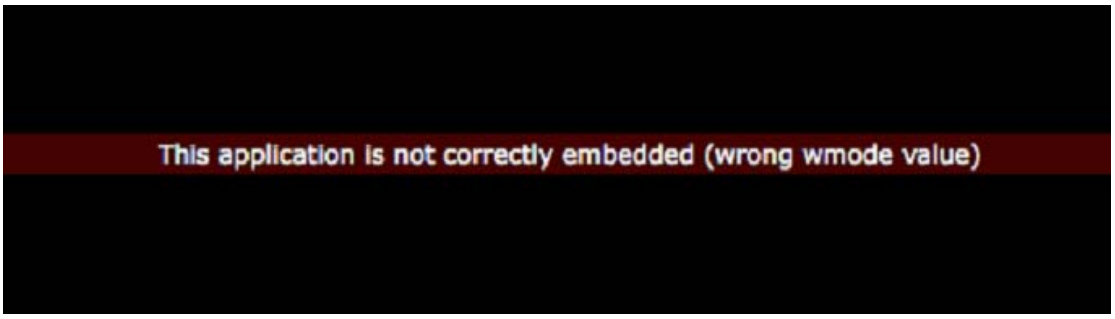

图 1.11
当应用程序没有被正确地嵌入到网页中时输出的信息

Stage 质量

作为一个 Flash 开发人员，stage 质量（quality）这个概念对你来说并不陌生。不过你需要记住的是，当你在 Stage3D 中运行时，不论你咋调整 stage 质量都是没用的，不要做徒劳的事情了，少年！

根据不同渲染模式决定优化策略

之前提到过，当 GPU 加速不可用的时候 Stage3D 将会自动回溯到软件渲染模式，此时将会使用一种叫做 SwiftShader (Transgaming)的软件回溯引擎接管渲染工作。那么，为了保证你的应用在别人机子上跑的时候即使回溯到软件加速也能保证性能，你就需要在编码时知道当前应用用的是什么渲染模式并及时关闭一些潜在的会拖慢运行效率的特效。
虽然在回溯到软件渲染的时候 Stage3D 也能保证不算太差的表现，你最好还是需要知道当前应用使用的渲染模式是否是软件渲染。在 Starling 中存在一个 context 的静态属性，它是一个 Context3D 类型的对象，我们可以根据这个对象中记录着的信息来查询当前渲染模式：

```
// 判断是否使用硬件渲染
var isHW:Boolean = Starling.context.driverInfo.toLowerCase().indexOf("software") == -1;
```

我们建议你最好始终为你的应用留条后路，在软件渲染下使用另一套渲染方案这样子，若是你这样做了，就可以确保你的应用在任何配置的机子上都可以高效运转。

现在呢，让我们来探讨一个更加有趣的话题，Starling 中的显示列表！

显示列表

可以说，Starling 遵循了和原生 Flash 显示列表一样的规则。Starling 中的显示对象将会无法

访问其 stage 属性，直到其被添加到显示列表中。在以往的编程中，我们为了更加安全地访问 stage 对象，我们需要用到一些重要的事件（如 Event.ADDED_TO_STAGE），幸运的是，在 Starling 中我们也可以使用这一些事件（但这些事件所在的包与以往不同）：

* Event.ADDED：当显示对象被添加到一个容器中抛出
* Event.ADDED_TO_STAGE：当显示对象被添加到一个已存在于舞台上的容器中时，也就是其变得可见时抛出
* Event.REMOVED：当显示对象被从一个容器中移除时抛出
* Event.REMOVED_FROM_STAGE ：当显示对象被从一个已存在于舞台上的容器中移除时，也就是其变得不可见时抛出

在之后的实验中我们将会用到这一些事件来帮助我们在合适的时候初始化一些东西或让一些东西失效，这样可以让代码执行效率更佳。

下面是 Starling 中 DisplayObject 类提供的一系列共有方法：

* removeFromParent：从父对象中移除，如果它有父对象的话
* getTransformationMatrixToSpace： 创建一个用于表现本地坐标系和另一个坐标系转换关系的矩阵
* getBounds：得到一个以某个坐标系为参考系的能包含该显示对象的最小矩形。
* hitTestPoint：返回当前坐标系中某个点位置下层次最高（挡在最前面）的显示对象
* globalToLocal：将一个点由舞台坐标转换为当前坐标系坐标
* localToGlobal：将一个点由当前坐标系坐标转换为舞台坐标

下面列出 DispObject 中提供的属性，作为一个 Flash 开发人员，你会很开心地看见在 Starling 中的 DisplayObject 保留了大多数和原生 Flash 中的 DisplayObject 一样名字的属性（功能也一梯 ，且还提供了一些额外的功能，如 pivotX 和 pivotY 属性，允许开发者在运行时动态改变 DisplayObject 的注册点：

* **transformationMatrix**：当前显示对象位置相与其父容器的转换矩阵
* **bounds**：当前显示对象在其父容器中的矩形（Rectangle）对象
* **width、height、root、x、y、scaleX、scaleY、alpha、visible、parent、stage、root**：不解释了，和原生 DisplayObject 一样的功能
* **rotation**：当前显示对象绕其注册点的旋转弧度（非角度）
* **pivotX、pivotY**：当前显示对象的注册点，默认为（0，0）
* **touchable** ：指定当前显示对象是否能够接受 Touch 事件（相当于原生 DisplayObject 的 mouseEnable，因为在 Starling 中所有鼠标事件都被定义为 Touch 事件）

跟原生 Flash API 一样，在 Starling 中，Sprite 类也是你能使用的最轻量级的容器。当然，其作为 DisplayObject 的子类，自然就继承了之前提到过的那些 DisplayObject 中拥有的 API，这就提供了 Sprite 对象之间可以互相嵌套的能力

作为一个容器，Sprite 对象继承自 DisplayObjectContainer 类，下面是 DisplayObjectContainer 类中提供的 API：

* addChild：不解释，和原生 Flash 中的一样，下同
* addChildAt：略
* dispose：完全销毁一个对象，释放其在 GPU 中所占内存，移除其全部事件侦听。
* removeFromParent：略
* removeChild：略
* removeChildAt：略
* removeChildren：移除一个容器中所有的子对象
* getChildAt：略

* getChildByName：根据名称搜索一个子对象
* getChildIndex：略
* setChildIndex：略
* swapChildren：略
* swapChildrenAt：略
* contains：略

一旦你可以访问一个显示对象的 stage 属性，你就可以通过此属性来对舞台进行一系列的设置，舞台对象继承于 DisplayObjectContainer，因此你可以对它调用大多数 DisplayObjectContainer 的 API，不仅如此，你还可以通过 stage 对象的 color 属性来直接改变 Starling 舞台的颜色。默认情况下，Starling 舞台的颜色会和原生 Flash 舞台颜色一致，这意味着你可以直接通过[SWF]元标签设置原生舞台颜色来间接地设置 Starling 舞台颜色：

```
[SWF(width="1280", height="752", frameRate="60", backgroundColor="#990000")]
```

若你想重载此行为（即我在[SWF]元标签中设置了原生 Flash 舞台颜色为某一颜色，现在我想把 Starling 舞台颜色改成另一种颜色），你只需要简单地在任何一个被添加到舞台上的 DisplayObject 中访问 stage 属性并设置其 color 属性即可：

```
package
{
    import starling.display.Quad;
    import starling.display.Sprite;
    import starling.events.Event;

    public class Game extends Sprite
    {
      private var q:Quad;

        public function Game()
        {
      addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }

    private function onAdded ( e:Event ):void
    {
      // set the background color to blue
      stage.color = 0x002143;

      q = new Quad(200, 200);
      q.setVertexColor(0, 0x000000);
      q.setVertexColor(1, 0xAA0000);
      q.setVertexColor(2, 0x00FF00);
      q.setVertexColor(3, 0x0000FF);
      addChild ( q );
    }
    }
}
```

我们之前只是简单地用了一对三角形来形成了一个方块，此方块并没有使用任何的纹理，而只是为它各个顶点设置了不同的颜色，这样会让 GPU 自动为其内部填充上渐变色。当然，如果你想要创建一个实色的方块，只需要设置 Quad 对象的 color 属性即可：

```
package
{
	import starling.display.Quad;
	import starling.display.Sprite;
	import starling.events.Event;

	public class Game extends Sprite
	{
	private var q:Quad;

		public function Game()
		{
	   addEventListener(Event.ADDED_TO_STAGE, onAdded);
		}

	private function onAdded ( e:Event ):void
	{
	   q = new Quad(200, 200);
	   q.color = 0x00FF00;
	   q.x = stage.stageWidth - q.width >> 1;
	   q.y = stage.stageHeight - q.height >> 1;
	   addChild ( q );
	}
	}
}
```

之后你会得到如下结果：

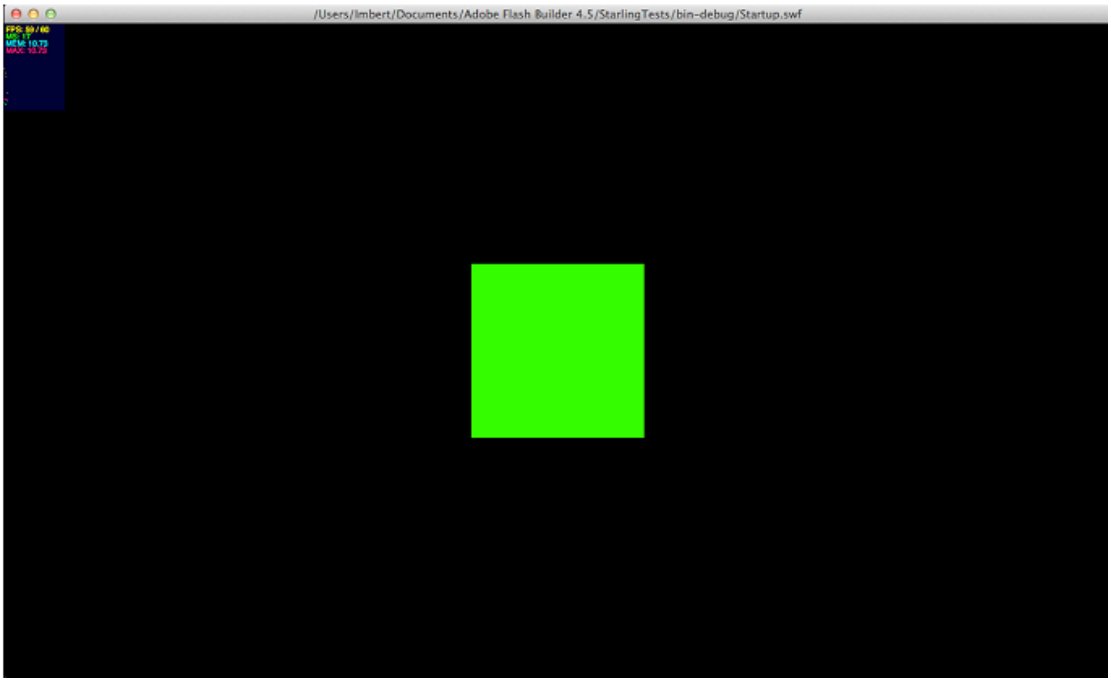

图 1.12

一个绿色实色的方块

我们现在将开始侦听Event.ENTER_FRAME事件并在事件处理函数中为当前方块增加一个颜色变化的补间动画，让其颜色在不同的随机颜色间变化：

```
package
{
   import starling.display.Quad;
   import starling.display.Sprite;
   import starling.events.Event;
   public class Game extends Sprite
   {
      private var q:Quad;
      private var r:Number = 0;
      private var g:Number = 0;
      private var b:Number = 0;
      private var rDest:Number;
      private var gDest:Number;
      private var bDest:Number;
      public function Game()
      {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
      }
      private function onAdded ( e:Event ):void
      {
         resetColors();
         q = new Quad(200, 200);
         q.x = stage.stageWidth - q.width >> 1;
```

```
            q.y = stage.stageHeight - q.height >> 1;
            addChild ( q );
            s.addEventListener(Event.ENTER_FRAME, onFrame);
        }
        private function onFrame (e:Event):void
        {
            r -= (r - rDest) * .01;
            g -= (g - gDest) * .01;
            b -= (b - bDest) * .01;
            var color:uint = r << 16 | g << 8 | b;
            q.color = color;
            // when reaching the color, pick another one
            if ( Math.abs( r - rDest) < 1 && Math.abs( g - gDest) < 1 && Math.abs( b - bDest) )
            resetColors();
        }
        private function resetColors():void
        {
            rDest = Math.random()*255;
            gDest = Math.random()*255;
            bDest = Math.random()*255;
        }
    }
}
```

之后，为了让我们的方块能够再旋转起来，我们需要用上 rotation 这个属性，不过值得注意的是，Starling 中的 DisplayObject 的 rotation 属性是以弧度为单位的，而非原生 Flash 中的以角度为单位。Starling 这样设计的目的是为了和其前身 Sparrow 的规则一致。如果你执意要使用角度作为单位的话，你可以用上 starling.utils.deg2rad 这个方法来将角度转换为弧度：

```
sprite.rotation = deg2rad(Math.random()*360);
```

由于 Starling 中的全部 DisplayObject 都具有 pivotX 及 pivotY 属性，我们可以非常便捷地在运行时改变其注册点，以满足我们在进行如缩放、旋转时的需要。这里我们将让方块以其中心点为轴心进行旋转，所以需要将其注册点放在其中心位置：

```
q.pivotX = q.width >> 1;
q.pivotY = q.height >> 1;
```

对于 Flash 开发人员来说，原生 Flash 中的编程理念在 Starling 中几乎完全一致。作为 DisplayObject 的子类，我们现在用的 Quad 对象可以和一个 Textfield 对象嵌套添加到一个 Sprite 对象中，且我们对此 Sprite 对象所做的一些操作，如缩放、位移等都会同时影响其中所有的子对象，和原生 Flash 中的规则一样，对吧？

```
package
{
    import starling.display.DisplayObject;
    import starling.display.Quad;
    import starling.display.Sprite;
    import starling.events.Event;
```

```
import starling.text.TextField;
public class Game extends Sprite
{
    private var q:Quad;
    private var s:Sprite;
    private var r:Number = 0;
    private var g:Number = 0;
    private var b:Number = 0;
    private var rDest:Number;
    private var gDest:Number;
    private var bDest:Number;
    public function Game()
    {
        addEventListener(Event.ADDED_TO_STAGE, onAdded);
    }
    private function onAdded ( e:Event ):void
    {
        resetColors();
        q = new Quad(200, 200);
        s = new Sprite();
        var legend:TextField = new TextField(100, 20, "Hello Starling!", "Arial", 14, 0xFFFFFF);
        s.addChild(q);
        s.addChild(legend);
        s.pivotX = s.width >> 1;
        s.pivotY = s.height >> 1;
        s.x = (stage.stageWidth - s.width >> 1 ) + (s.width >> 1);
        s.y = (stage.stageHeight - s.height >> 1) + (s.height >> 1);
        addChild(s);
        s.addEventListener(Event.ENTER_FRAME, onFrame);
    }
    private function onFrame (e:Event):void
    {
        r -= (r - rDest) * .01;
        g -= (g - gDest) * .01;
        b -= (b - bDest) * .01;
        var color:uint = r << 16 | g << 8 | b;
        q.color = color;
        // when reaching the color, pick another one
        if ( Math.abs( r - rDest) < 1 && Math.abs( g - gDest) < 1 && Math.abs( b - bDest) )
        resetColors();
        (e.currentTarget as DisplayObject).rotation += .01;
    }
    private function resetColors():void
    {
```

```
                rDest = Math.random()*255;
                gDest = Math.random()*255;
                bDest = Math.random()*255;
            }
        }
    }
```

我们现在就可以把包含有我们的 Quad 对象和 Textfield 对象的 Sprite 对象围绕其中心点进行旋转了，颜色也在不停地变哦~

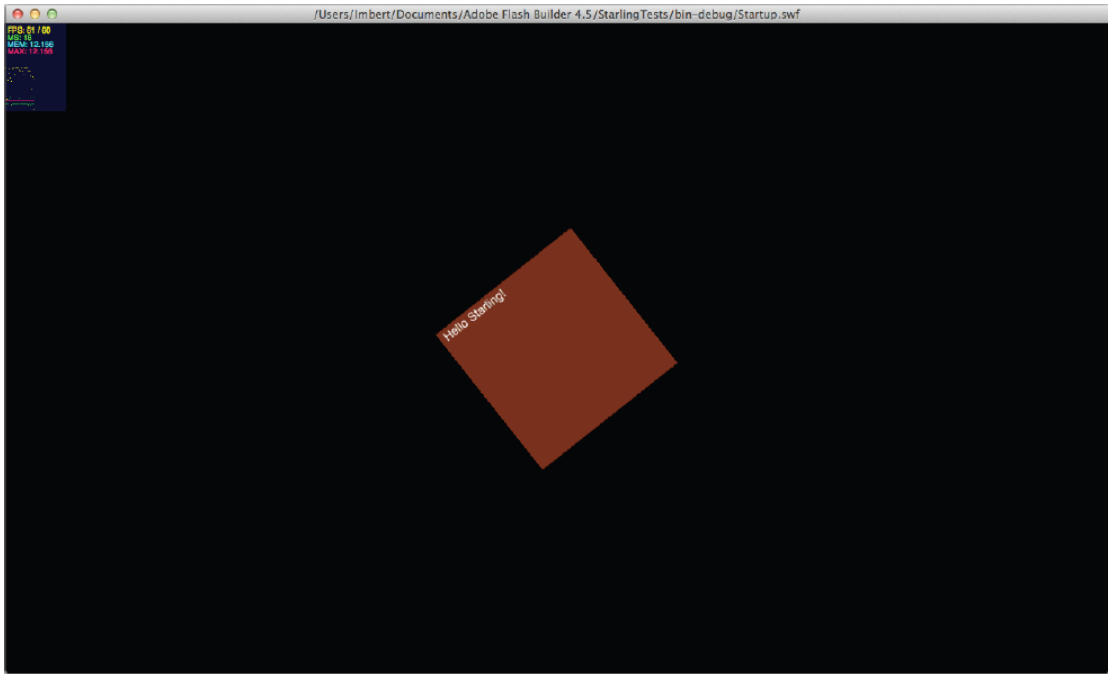

图 1.13

一个包含有 Quad 及 Textfield 对象的 Sprite 对象在旋转

我们的代码现在看起来感觉有点凌乱了，所以我们现在把一些代码封装到一个 CustomSprite 类里面去。下面给出的就是其全部代码：

```
package
{
    import starling.display.Quad;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.text.TextField;
    public class CustomSprite extends Sprite
    {
        private var quad:Quad;
        private var legend:TextField;
        private var quadWidth:uint;
        private var quadHeight:uint;
        private var r:Number = 0;
        private var g:Number = 0;
```

```
private var b:Number = 0;
private var rDest:Number;
private var gDest:Number;
private var bDest:Number;
public function CustomSprite(width:Number, height:Number, color:uint=16777215)
{
    // reset the destination color component
    resetColors();
    // set the width and height
    quadWidth = width;
    quadHeight = height;
    // when added to stage, activate it
    addEventListener(Event.ADDED_TO_STAGE, activate);
}
private function activate(e:Event):void
{
    // create a quad of the specified width
    quad = new Quad(quadWidth, quadHeight);
    // add the legend
    legend = new TextField(100, 20, "Hello Starling!", "Arial", 14, 0xFFFFFF);
    // add the children
    addChild(quad);
    addChild(legend);
    // change the registration point
    pivotX = width >> 1;
    pivotY = height >> 1;
}
private function resetColors():void
{
    // pick random color components
    rDest = Math.random()*255;
    gDest = Math.random()*255;
    bDest = Math.random()*255;
}
/**
* Updates the internal behavior
*
*/
public function update ():void
{
    // easing on the components
    r -= (r - rDest) * .01;
    g -= (g - gDest) * .01;
    b -= (b - bDest) * .01;
```

```
            // assemble the color
            var color:uint = r << 16 | g << 8 | b;
            quad.color = color;
            // when reaching the color, pick another one
            if ( Math.abs( r - rDest) < 1 && Math.abs( g - gDest) < 1 && Math.abs( b - bDest) )
            resetColors();
            // rotate it!
            //rotation += .01;
        }
    }
}
```

现在是我们修改过的 Game 类：

```
package
{
    import starling.display.Sprite;
    import starling.events.Event;
    public class Game extends Sprite
    {
        private var customSprite:CustomSprite;
        public function Game()
        {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded ( e:Event ):void
        {
            // create the custom sprite
            customSprite = new CustomSprite(200, 200);
            // positions it by default in the center of the stage
            // we add half width because of the registration point of the custom sprite (middle)
            customSprite.x = (stage.stageWidth - customSprite.width >> 1 ) + (customSprite.width >> 1);
            customSprite.y = (stage.stageHeight - customSprite.height >> 1) + (customSprite.height >> 1);
            // show it
            addChild(customSprite);
            // need to comment this one ? ;)
            stage.addEventListener(Event.ENTER_FRAME, onFrame);
        }
        private function onFrame (e:Event):void
        {
            // we update our custom sprite
            customSprite.update();
        }
    }
```

}

在 CustomSprite 中开放了的 update 的接口允许外部调用之以实现 CustomSprite 中设置的逻辑循环。

现在，让我们来为我们这个小小的测试程序再多加一个交互功能：让我们的方块跟着鼠标运动。因我们需要添加一些代码（粗体部分）来实现之：

```
package
{
    import flash.geom.Point;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.events.Touch;
    import starling.events.TouchEvent;
    public class Game extends Sprite
    {
        private var customSprite:CustomSprite;
        private var mouseX:Number = 0;
        private var mouseY:Number = 0;
        public function Game()
        {
                addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded ( e:Event ):void
        {
            // create the custom sprite
            customSprite = new CustomSprite(200, 200);
            // positions it by default in the center of the stage
            // we add half width because of the registration point of the custom sprite (middle)
            customSprite.x = (stage.stageWidth - customSprite.width >> 1 ) + (customSprite.width >> 1);
            customSprite.y = (stage.stageHeight - customSprite.height >> 1) + (customSprite.height >> 1);
            // show it
            addChild(customSprite);
            // we listen to the mouse movement on the stage
            stage.addEventListener(TouchEvent.TOUCH, onTouch);
            // need to comment this one ? ;)
            stage.addEventListener(Event.ENTER_FRAME, onFrame);
        }
        private function onFrame (e:Event):void
        {
            // easing on the custom sprite position
            customSprite.x -= ( customSprite.x - mouseX ) * .1;
            customSprite.y -= ( customSprite.y - mouseY ) * .1;
            // we update our custom sprite
            customSprite.update();
        }
```

```
        private function onTouch (e:TouchEvent):void
        {
            // get the mouse location related to the stage
            var touch:Touch = e.getTouch(stage);
            var pos:Point = touch.getLocation(stage);
            // store the mouse coordinates
            mouseX = pos.x;
            mouseY = pos.y;
        }
    }
}
```

值得注意的是，在这段代码中我们没有使用任何 Mouse API（指代鼠标事件），事实上，在 Starling 中并没有设计鼠标的概念，稍后我们会讨论到。

通过对 TouchEvent.TOUCH 事件的侦听，我们可以对鼠标/手指移动的侦测，这个事件的用法就像我们经典的 MousEvent.MOUSE_MOVE 事件一样。每一帧我们都可以依靠 TouchEvent 中的辅助 API 如 getTouch 及 getLocation 来获取并保存当前鼠标位置。在 onFrame 这个事件处理函数中我们使用存储的鼠标位置作为目的地并运用一个简单的缓动方程式来让方块渐渐运动至此。

就像之前我们说的，Starling 不仅让我们能享受到更加便捷的 GPU 应用编码过程，还能让我们在清理、回收对象时更加地方便。举个例子，假如我们需要在鼠标点击方块的时候将此方块对象从舞台上移除，那么你就需要写下面这样的几句代码：

```
package
{
    import flash.geom.Point;
    import starling.display.DisplayObject;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.events.Touch;
    import starling.events.TouchEvent;
    import starling.events.TouchPhase;
    public class Game extends Sprite
    {
        private var customSprite:CustomSprite;
        private var mouseX:Number = 0;
        private var mouseY:Number = 0;
        public function Game()
        {
                addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded ( e:Event ):void
        {
            // create the custom sprite
            customSprite = new CustomSprite(200, 200);
            // positions it by default in the center of the stage
```

```
            // we add half width because of the registration point of the custom sprite (middle)
            customSprite.x = (stage.stageWidth - customSprite.width >> 1 ) + (customSprite.width >> 1);
            customSprite.y = (stage.stageHeight - customSprite.height >> 1) + (customSprite.height >> 1);
            // show it
            addChild(customSprite);
            // we listen to the mouse movement on the stage
            stage.addEventListener(TouchEvent.TOUCH, onTouch);
            // need to comment this one ? ;)
            stage.addEventListener(Event.ENTER_FRAME, onFrame);
            // when the sprite is touched
            customSprite.addEventListener(TouchEvent.TOUCH, onTouchedSprite);
        }
        private function onFrame (e:Event):void
        {
            // easing on the custom sprite position
            customSprite.x -= ( customSprite.x - mouseX ) * .1;
            customSprite.y -= ( customSprite.y - mouseY ) * .1;
            // we update our custom sprite
            customSprite.update();
        }
        private function onTouch (e:TouchEvent):void
        {
            // get the mouse location related to the stage
            var touch:Touch = e.getTouch(stage);
            var pos:Point = touch.getLocation(stage);
            // store the mouse coordinates
            mouseX = pos.x;
            mouseY = pos.y;
        }
        private function onTouchedSprite(e:TouchEvent):void
        {
            // get the touch points (can be multiple because of multitouch)
            var touch:Touch = e.getTouch(stage);
            var clicked:DisplayObject = e.currentTarget as DisplayObject;
            // detect the click/release phase
            if ( touch.phase == TouchPhase.ENDED )
            {
                // remove the clicked object
                removeChild(clicked);
            }
        }
    }
}
```

注意，我们虽然将对象从显示列表中移除了，但是我们还未移除其 Event.ENTER_FRAME

事件的侦听器。为了证实这一点，我们使用 hasEventListener 这个 API 来对 Sprite 对象做一个测试。

```
private function onTouchedSprite(e:TouchEvent):void
{
    // get the touch points (can be multiple because of multitouch)
    var touch:Touch = e.getTouch(stage);
    var clicked:DisplayObject = e.currentTarget as DisplayObject;
    // detect the click/release phase
    if ( touch.phase == TouchPhase.ENDED )
    {
        // remove the clicked object
        removeChild(clicked);
        // outputs : true
        trace ( clicked.hasEventListener(e.type) );
    }
}
```

可见，即使将对象从显示列表中移除了，它的事件侦听器依然残留着。为了更加安全、彻底地移除一个对象，我们需要给 removeChild 方法设置其第二个参数 dispose 为 true，这样可以让我们在从显示列表中移除一个对象的时候自动移除其所有的事件侦听器：

```
private function onTouchedSprite(e:TouchEvent):void
{
    // get the touch points (can be multiple because of multitouch)
    var touch:Touch = e.getTouch(stage);
    var clicked:DisplayObject = e.currentTarget as DisplayObject;
    // detect the click/release phase
    if ( touch.phase == TouchPhase.ENDED )
    {
        // remove and dispose all the listeners
        removeChild(clicked, true);
        // outputs : false
        trace ( clicked.hasEventListener(e.type) );
    }
}
```

我们这样做可以确保完全地移除一个对象，且如果此对象中还有子对象，那么这些子对象也都会被完全移除。dispose 这个参数在其他移除对象的 API 中均存在，如 removeChildren、removeChildAt。需要注意的是，完全移除一个对象，不仅仅移除了其纹理，其在 GPU 中所占内存也会被一并释放掉。若你只是想移除一个纹理，那么你可以通过调用 Texture 或 TextureAtlas 对象的 dispose 方法来做到这一点。

除了使用上面提到的一系列 remove 开头的方法来完全移除一个子对象之外，你还可以直接调用一个 DisplayObject 对象的 dispose 方法来实现对象的自移除。

```
clicked.dispose()
```

我们在刚刚那个小测试中初次使用了一下 Starling 中的事件机制，是不是感觉和原生 Flash 使用方式没多大差别呢？那么现在让我们再花一点时间来了解一下 Starling 中的 EventDispatcher 这个 API。

事件模型

在之前给出的图 1.2 中我们看到，所有的 Starling 对象都继承自 EventDispatcher 类。就像原生 Flash 中一样，Starling 中所有 EventDispatcher 的子类（事实上就是全部类，因为全部类都继承自它）都提供了以下一些 API 来分发/接受事件：

* **addEventListener**：为指定事件添加事件侦听器.
* **hasEventListener**：为指定事件监测是否具备事件侦听器
* **removeEventListener**：移除某个事件侦听器
* **removeEventListeners**：移除一个对象中对某一事件或全部事件添加的侦听器

你应该注意到，在原生 Flash 提供的那些 API 基础上，Starling 又额外提供了一个极其有用的 API——removeEventListeners。在任何时候只要你想移除某个对象对于某个事件注册的全部事件侦听器，只需要调用该方法并传入欲移除的事件类型作为参数即可：

```
button.removeEventListeners(Event.TRIGGERED);
```

若你想要一次性移除某个对象的全部事件侦听器（无论是对什么事件注册的），只需要保持 removeEventListeners 方法的参数为空即可：

```
button.removeEventListeners ();
```

其实，在之前提到过的 removeChild 方法中，若 dispose 参数为 true，目标对象内部就是通过对其全部子对象调用 removeEventListeners 这个方法来实现彻底清理。

事件冒泡机制

在本教程的最开始我们就提到过，Starling在 Stage3D的基础上重现了原生Flash中显示列表的概念，好消息是，在Starling中你还可以持续使用我们以前很喜爱的，功能强大的事件冒泡机制。事件冒泡机制在很多场合都有着非常重要的用处，如让我们尽可能地减少需要注册的事件侦听器且可以简化我们的不少代码。若你对原生Flash的事件冒泡机制还不太属性，你可以参考这里：http://www.adobe.com/devnet/actionscript/articles/event_handling_as3.html

Starling 中的事件机制保留了事件冒泡阶段，却没有事件捕捉阶段。在稍后的例子中我们将一起来看看如何使用事件冒泡。

Touch 事件

之前我们提到过，Starling 的前身是 Sparrow（一款手机应用开发框架），因此 Starling 中会存在一些与手机应用相关的概念也是理所应当，我们知道，当前手机交互一般是依靠手指触摸来实现，因此 Starling 中的 Touch 事件是为手指触摸式交互设计的，但是会让那些桌面应用的开发者产生一些迷惑，他们不清楚在使用鼠标进行交互的场合中要如何去使用此事件。

首先，如果你观察得仔细，你会在图 1.2 中发现，Starling 的类结构中不存在 InteractiveObject 这个类，也就是说，Starling 中所有的 DisplayObject 都是支持交互的【PS：在原生 Flash AS3 中只有继承自 InteractiveObject 的类才能够交互，才能够接受鼠标事件。如 Bitmap、Shape 类只继承自 DisplayObject 而未继承自 InteractiveObject，因此其不支持交互。而 Sprite 对象则支持交互】。换句话说，Starling 中的 DisplayObject 类已经内置了交互功能。

在之前的例子里我们已经对 Touch 事件小试了一下牛刀，现在，让我们从最简单的概念开始了解此事件。假设我们现在需要当鼠标点击到方块时让方块做出点反应【PS：如叫两声“牙卖呆~”、“克一毛基”之类的】，这就需要用到 TouchEvent.TOUCH 事件。

```
// when the sprite is touched
_customSprite.addEventListener(TouchEvent.TOUCH, onTouchedSprite);
```

每当鼠标或手指与一个图形交互时，TouchEvent.TOUCH 事件就会被抛出。Touch 事件不只是一个简单的事件而已，其中还包含了极其丰富的信息，接下来让我们深入地了解一下。在下面的代码中，我们将在 onTouch 这个事件处理函数中输出 TouchEvent 对象中携带的 Touch 对象的 phase 属性：

```
private function onTouch (e:TouchEvent):void
{
    // get the mouse location related to the stage
    var touch:Touch = e.getTouch(stage);
    var pos:Point = touch.getLocation(stage);
    trace ( touch.phase );
    // store the mouse coordinates
    _mouseY = pos.y;
    _mouseX = pos.x;
}
```

当我们开始点击方块或者进行一些交互的时候我们会看到会输出不同的 phase 值，phase 属性的所有可能值都被定义为 TouchPhase 类中的常量，下面列出这些值：

* **began**：鼠标/手指开始交互（类似于mouseDown）
* **ended**：鼠标/手指停止交互（类似于mouseClick）
* **hover**：鼠标/手指悬于物体上（类似于mouseOver）
* **moved**：鼠标/手指在物体上移动（类似于mouseDown + mouseMove）
* **stationary**：鼠标/手指停止与物体的交互但仍停留在其上

接下来我们一起再来看一些 TouchEvent 事件对象中的另一些可用的 API：

* **ctrlKey**：触发Touch事件是是否按住Ctrl键
* **getTouch**：得到此事件的Touch对象
* **getTouches**：得到一组Touch对象（用于多点交互）
* **shiftKey**：触发Touch事件是是否按住Shift键
* **timestamp**：事件触发时间
* **touches**：得到同一时间发生的全部 Touch 对象

shiftKey及ctrlKey属性对于判断是否按下组合键非常有用。因此每次产生交互时，都能得到一个与当前手指或鼠标信息有关的Touch对象。

让我们一起来看Touch对象中的API：

* **clone**：复制一个副本
* **getLocation**：得到Touch事件触发的对应于当前坐标系的位置
* **getPreviousLocation**：得到之前触发的Touch事件对应于当前坐标系的位置
* **globalX、Y**：得到Touch事件触发的舞台位置

* **id**：一个Touch对象所拥有的独一无二的标示符
* **phase**：指示当前事件触发的类型（阶段）
* **previousGlobalX、Y**：得到之前触发的Touch事件舞台位置
* **tapCount**：手指轻拍显示对象的次数（用以识别手指双拍）
* **target**：触发Touch事件的对象
* **timestamp**：事件触发时间（此时间是从应用程序启动开始计时的）

模拟多点触摸

当为移动设备开发应用的时候，你有很多时候会用上多点触摸的功能，如用多点触摸来进行内容缩放。不过当你在电脑上开发用于移动设备的应用时，当你无法实时地在移动设备上进行调试时，不用担心，Starling 为我们提供了一个内置的多点触摸模拟机制。
如果你需要启用多点触摸功能，你只需要设置一下 Starling 对象的 simulateMultiTouch 属性即可：

```
package
{
   import flash.display.Sprite;
   import flash.display.StageAlign;
   import flash.display.StageScaleMode;
   import starling.core.Starling;
   [SWF(width="1280", height="752", frameRate="60", backgroundColor="#002143")]
   public class Startup extends Sprite
   {
      private var mStarling:Starling;
      public function Startup()
      {
         // stats class for fps
         addChild ( new Stats() );
         stage.align = StageAlign.TOP_LEFT;
         stage.scaleMode = StageScaleMode.NO_SCALE;
         // create our Starling instance
         mStarling = new Starling(Game, stage);
         // emulate multi-touch
         mStarling.simulateMultitouch = true;
         // set anti-aliasing (higher the better quality but slower performance)
         mStarling.antiAliasing = 1;
         // start it!
         mStarling.start();
      }
   }
}
```

一旦启用了多点触摸功能，你就可以在应用程序中通过按下 Ctrl 键然后移动鼠标就可以模拟多点触摸的输入了，此时在屏幕上会出现两个圆点，用以指示两个触摸点位置（模拟两根手指），如下图所示：

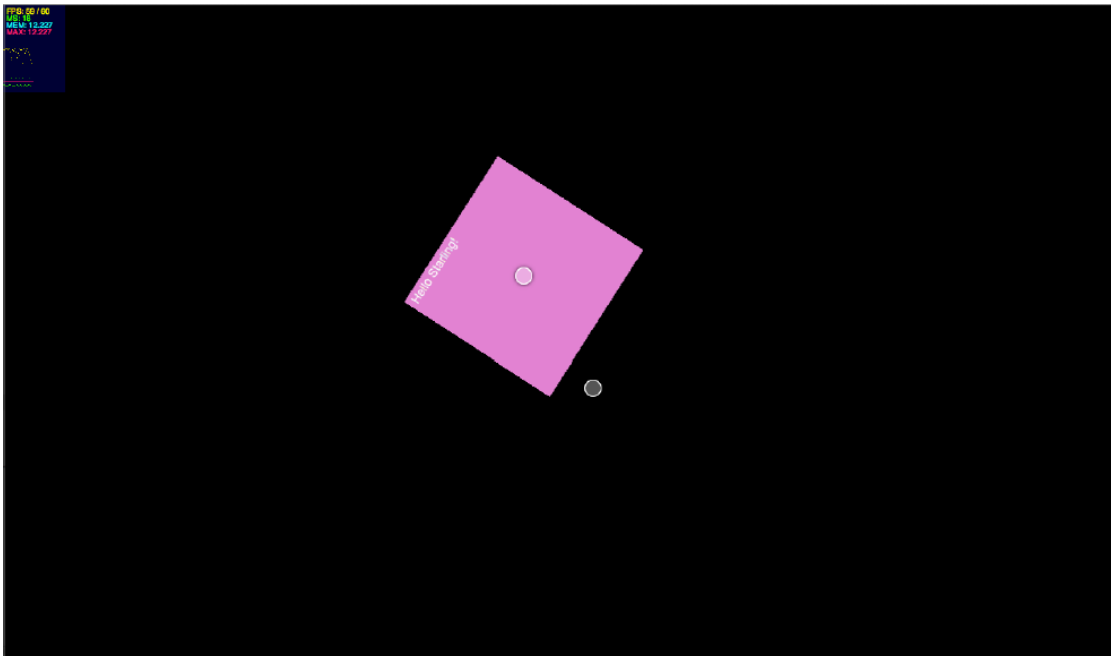

图 1.14
多点触摸模拟

在下面的代码中，我们假设准备利用多点触摸功能来放大一个方块，就像在触屏手机上用两指手指进行放大那样。因此我们就需要使用下面的代码来运算出两触摸点间的距离：

```
package
{
   import flash.geom.Point;
   import starling.display.Sprite;
   import starling.events.Event;
   import starling.events.Touch;
   import starling.events.TouchEvent;
   import starling.events.TouchPhase;
   public class Game extends Sprite
   {
      private var customSprite:CustomSprite;
      public function Game()
      {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
      }
      private function onAdded ( e:Event ):void
      {
         // create the custom sprite
         customSprite = new CustomSprite(200, 200);
         // positions it by default in the center of the stage
         // we add half width because of the registration point of the custom sprite (middle)
         customSprite.x = (stage.stageWidth - customSprite.width >> 1 ) + (customSprite.width >> 1);
         customSprite.y = (stage.stageHeight - customSprite.height >> 1) + (customSprite.height >> 1);
```

```
            // show it
            addChild(customSprite);
            // we listen to the mouse movement on the stage
            //stage.addEventListener(TouchEvent.TOUCH, onTouch);
            // need to comment this one ? ;)
            stage.addEventListener(Event.ENTER_FRAME, onFrame);
            // when the sprite is touched
            customSprite.addEventListener(TouchEvent.TOUCH, onTouchedSprite);
        }
        private function onFrame (e:Event):void
        {
            // we update our custom sprite
            customSprite.update();
        }
        private function onTouchedSprite(e:TouchEvent):void
        {
            // retrieves the touch points
            var touches:Vector.<Touch> = e.touches;
            // if two fingers
            if ( touches.length == 2 )
            {
                var finger1:Touch = touches[0];
                var finger2:Touch = touches[1];
                var distance:int;
                var dx:int;
                var dy:int;
                // if both fingers moving (dragging)
                if ( finger1.phase == TouchPhase.MOVED && finger2.phase == TouchPhase.MOVED )
                {
                    // calculate the distance between each axes
                    dx = Math.abs ( finger1.globalX - finger2.globalX );
                    dy = Math.abs ( finger1.globalY - finger2.globalY );
                    // calculate the distance
                    distance = Math.sqrt(dx*dx+dy*dy);
                    trace ( distance );
                }
            }
        }
    }
}
```

接下来，让我们一起了解一下怎样为 Starling 中的物体穿上材质吧~!

Texture

如果你创建了一个 Image 对象，那你必须为其传入一个 Texture 对象作为外观。这两者的关系就像原生 Flash 中 Bitmap 和 BitmapData 的关系【PS：嗯……有基情】。下面来看看 Texture 对象提供的 API：

* **base**：该Texture对象所基于的Stage3D texture对象
* **dispose**：销毁该Texture对象的潜在纹理数据
* **empty**：静态方法。创建一个指定大小和颜色的空Texture对象
* **frame**：一个Rectangle矩形对象，用于指示一个Texture对象的范围
* **fromBitmap**：静态方法。从一个Bitmap对象创建一个外观与其一致的Texture纹理对象。
* **fromBitmapData**：静态方法。从一个BitmapData对象创建一个外观与其一致的Texture纹理对象。
* **fromAtfData**：静态方法。允许运用ATF (Adobe Texture Format)格式创建一个压缩过的材质。经过压缩的材质能让你节省大量空间，尤其是在像移动设备这样对存储空间异常苛刻的环境中尤为有用。
* **fromTexture**：静态方法。从一个Texture对象创建一个外观与其一致的Texture纹理对象。
* **height 、width**：我想这两个属性就不用多说了
* **mipmapping**：此属性用以指示该材质是否包含了mip映射
* **premultipliedAlpha**：此属性用以指示该texture对象的透明度是否被预乘到了RGB值中(premultiplied into the RGB values).
* **repeat**：用以指定当前材质是否启用重复平铺模式，就像铺壁纸那样。

【PS：API 功能翻译有时会有曲意，在使用时还请以英文 API 介绍为参考】

有很多不同格式的文件可被用作 Texture 对象的数据提供源，下面列出了所有可用的文件格式：

* **PNG**：由于所需素材中经常需要保留透明通道，因此PNG格式的文件是Texture最常用的素材格式。
* **JPEG**：经典的JPEG格式文件也可以为Texture所用。有一点需要注意，就是在GPU中该格式的图片会被解压缩，这意味着JPEG格式的文件将无法发挥其节省空间的优势，且其不保留透明通道哦
* **JPEG**-XR : JPEG XR是一个为了让图片色调更加连贯，视觉效果更加逼真而存在的图片压缩标准及图片文件格式，它是基于一种被称作HD Photo的技术（起初由Microsoft微软公司开发并拥有专利，曾用名为Windows Media Photo）。它同时支持有损和无损压缩，且它是Ecma-388 Open XML Paper Specification文档标准推荐的图片存储优先格式。
* **ATF** : Adobe Texture Format. 这是一种能提供最佳压缩效果的文件格式。ATF文件主要是一个存储有损纹理数据（lossy texture data）的文件容器。它主要使用了两种类似技术：JPEG-XR1压缩技术和基于块的压缩技术（简称块压缩技术），来实现它的有损压缩。

JPEG-XR压缩技术提供了一种非常有竞争力的方式来节省存储空间及网络带宽。而块压缩技术则提供了一种能够在客户端削减纹理存储空间（与一般的RGBA纹理文件所占存储空间的比例为1:8）的方式。ATF提供了三种块压缩技术：DXT12, ETC13 及

PVRTC4.

接下来让我们再更加深入地了解一些有关材质的概念并探索一下 GPU 中的一个基本概念：Mip 映射。Mip 映射是一个重要却简单易懂的概念。将一个纹理保存多个缩小版本的方式就叫做 Mip 映射【PS：如一个 256*256 尺寸的纹理被保存了 128*128、64*64…1*1 这么多版本的纹理于内存中】。在 GPU 中工作时，一个图片若将被缩放，那么它将被缩放到的大小将取决于其原始尺寸。缩放行为一般会发生在镜头向物体移动或者物体向镜头移动时，这两种情况下都会对你的图片（在 GPU 中表现为纹理）产生缩放。

需要注意的是，若要使用Mip映射，那么你的纹理尺寸必须保证为2的倍数（1, 2, 4, 8, 16, 32, 64, 128, 256,512, 1024, 2048），但形状不一定必须是矩形。如果你没有遵守这个规则，那么Starling将会为你自动创建一个与当前纹理尺寸最接近的能被2整除的数值作为尺寸的纹理（如你使用的纹理尺寸为31*31，那么Starling会为你创建一个32*32尺寸的纹理），但这可能会对内存有一点消耗。为了确保尽可能地优化纹理的内存占用，我们建议您最好使用texture atlases（翻译成中文叫做纹理贴图集，但是没多少人会用中文称呼之），也被广泛称作SpriteSheet（翻译成中文叫做精灵表，但是也没人这么叫它，都直接用英文名称呼的）的素材集成、使用方式。稍后我们会接着讨论它。

为了保证最佳的呈现品质，GPU 需要一个图片的全部 Mip 映射等级，即由原始尺寸依次除以二直到除不尽 2 了为止。【PS：对于一个 128*128 尺寸的纹理来说，它的全部 Mip 映射等级为：64*64,32*32,16*16,8*8,4*4,2*2 以及 1*1】Starling 框架能够自动替你生成全部 Mip 映射等级，若是你不用 Starling 框架的话，那你就得通过使用 BitmapData.draw 这个 API 并使用一个缩小一倍的 Matrix 作为参数来手动地生成全部的映射等级。

我们建议为2D内容使用Mip映射，这样可以使它们在缩放时能够减少锯齿的产生

幸运的是，之前我们提到过，Starling 能够自动替你生成全部的 Mip 映射等级，它其实也是用了 BitmapData.draw 来做的，下面是 Starling 中用以生成 Mip 映射的代码：

```
if (generateMipmaps)
{
    var currentWidth:int = data.width >> 1;
    var currentHeight:int = data.height >> 1;
    var level:int = 1;
    var canvas:BitmapData = new BitmapData(currentWidth, currentHeight, true, 0);
    var transform:Matrix = new Matrix(.5, 0, 0, .5);
    while (currentWidth >= 1 || currentHeight >= 1)
    {
        canvas.fillRect(new Rectangle(0, 0, currentWidth, currentHeight), 0);
        canvas.draw(data, transform, null, null, null, true);
        texture.uploadFromBitmapData(canvas, level++);
        transform.scale(0.5, 0.5);
        currentWidth = currentWidth >> 1;
```

```
        currentHeight = currentHeight >> 1;
    }
    canvas.dispose();
}
```

【PS：要在 Starling 中使用 Mip 映射只需要设置某个 Texture 对象的 mipmapping 属性为 true 即可】

当你使用 ATF 格式文件（Adobe Texture Format）作为素材时，你就不需要为它的 Mip 映射问题操心了，它内部已经包含了全部的 Mip 映射等级，且不是在运行时才去生成的，而是已经预生成好的。这是非常有价值的一点，因为这样可以为你在运行时节省生成 Mip 映射的时间。特别是在需要使用大量纹理的时候，使用 ATF 格式作为素材可以为你节省大量的初始化时间。

注意到，在 Texture 对象中存在一个 frame 属性，此属性允许我们设置一个 Texture 对象在 Image 对象【PS：如果把 Texture 对象理解成原生 Flash 中的 BitmapData，那么 Image 对象就可以理解成 Bitmap，用以承载 Texture 并将其显示出来】中的位置。假设你需要让一个 Image 对象存在一点边距，你只需要用一个稍小一些的 Texture 纹理，然后将其位置置于 Image 的中心就好了：

```
texture.frame = new Rectangle(5, 5, 30, 30);
var image:Image = new Image(texture);
```

刚才说到了 Image 对象，那么现在就来详细地探讨一下此对象。

Image

在 Starling 框架中，starling.display.Image 对象扮演的角色与原生 Flash 中的 flash.display.Bitmap 对象一致【PS:但是比 Bitmap 好的地方在于它能响应鼠标事件】：

```
var myImage:Image = new Image(texture);
```

为了让一个图片显示出来，你需要创建一个 Image 对象，之后再传递一个 Texture 对象给它：

```
package
{
    import flash.display.Bitmap;
    import starling.display.Image;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    import starling.utils.deg2rad;
    public class Game2 extends Sprite
    {
        private var sausagesVector:Vector.<Image> = new Vector.<Image>(NUM_SAUSAGES, true);
        private const NUM_SAUSAGES:uint = 400;
        [Embed(source = "../media/textures/sausage.png")]
        private static const Sausage:Class;
        public function Game2()
        {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
```

```
        }
        private function onAdded (e:Event):void
        {
            // create a Bitmap object out of the embedded image
            var sausageBitmap:Bitmap = new Sausage();
            // create a Texture object to feed the Image object
            var texture:Texture = Texture.fromBitmap(sausageBitmap);
            for (var i:int = 0; i < NUM_SAUSAGES; i++)
            {
                // create a Image object with our one texture
                var image:Image = new Image(texture);
                // set a random alpha, position, rotation
                image.alpha = Math.random();
                // define a random initial position
                image.x = Math.random()*stage.stageWidth
                image.y = Math.random()*stage.stageHeight
                image.rotation = deg2rad(Math.random()*360);
                // show it
                addChild(image);
                // store references for later
                sausagesVector[i] = image;
            }
        }
    }
}
```

注意到，在上面的代码中我们使用了 Texture 类中的静态方法 fromBitmap 来创建我们所需的 Texture 对象。下图显示了此段代码的执行结果：【PS：它用到的这个香肠图片没有在实例代码中提供，因此测试时我们需要自己找一个图片素材来用】

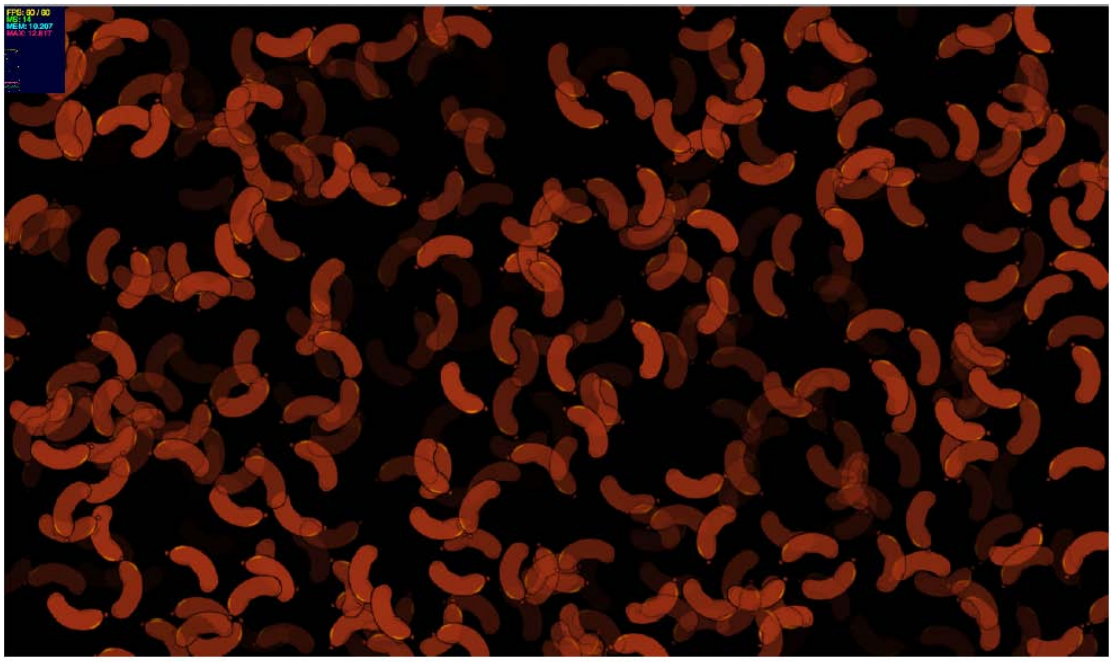

图 1.15
随机排布的香肠

我们刚才所用的位图是来源于嵌入资源的，当然，我们也可以动态加载素材。具体步骤也就是使用一个 Loader 对象进行远程加载素材文件，然后在加载完成后取出加载的 Bitmap 对象最后使用 fromBitmap 这个 Texture 类的静态方法来获取所需纹理。

```
// create the loader
var loader:Loader = new Loader();
// load the texture
loader.load ( new URLRequest ("texture.png") );
// when texture is loaded
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );
function onComplete ( e : Event ):void
{
   // grab the loaded bitmap
   var loadedBitmap:Bitmap = e.currentTarget.loader.content as Bitmap;
   // create a texture from the loaded bitmap
   var texture:Texture = Texture.fromBitmap ( loadedBitmap )
}
```

稍后，我们会讲到如何利用 Texture 类的另一个从 BitmapData 来创建 Texture 对象的静态方法。

Texture 对象是可以重用的。【PS：这一点跟 BitmapData 一样】在下面的代码中，我们在一个循环中创建了多个 Image 对象，他们重用唯一的一个 Texture 对象：

```
// create a Texture object to feed the Image object
var texture:Texture = Texture.fromBitmap(sausageBitmap);
for (var i:int = 0; i < NUM_SAUSAGES; i++)
{
   // create a Image object with our one texture
   var image:Image = new Image(texture);
}
```

正因为如此，我们不建议你为具有同样外观的 Image 对象创建多个 Texture：

```
for (var i:int = 0; i < NUM_SAUSAGES; i++)
{
   // create a Image object by creating a new texture for each sausage
   var image:Image = new Image(Texture.fromBitmap(new Sausage()));
}
```

创建如此多不必要的 Texture 对象会造成许多相同外观的 Image 对象被上传至 GPU，不仅会对内存造成浪费，一般来说也会对应用的呈现效率造成影响。每次对 fromBitmap 的调用都会创建 mip 映射，这样循环多次不会对性能造成影响才怪呢。

接下来我们在原有的例子基础上让全部图片都动起来，为了实现之，需要创建一个 CustomImage 类：

```
package
{
   import starling.display.Image;
```

```
    import starling.textures.Texture;
    public class CustomImage extends Image
    {
        public var destX:Number = 0;
        public var destY:Number = 0;
        public function CustomImage(texture:Texture)
        {
            super(texture);
        }
    }
}
```

之后在我们的应用中用上 CustomImage 类：

```
package
{
    import flash.display.Bitmap;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    import starling.utils.deg2rad;
    public class Game2 extends Sprite
    {
        private var sausagesVector:Vector.<CustomImage> = new
        Vector.<CustomImage>(NUM_SAUSAGES, true);
        private const NUM_SAUSAGES:uint = 400;
        [Embed(source = "../media/textures/sausage.png")]
        private static const Sausage:Class;
        public function Game2()
        {
                addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // create a Bitmap object out of the embedded image
            var sausageBitmap:Bitmap = new Sausage();
            // create a Texture object to feed the Image object
            var texture:Texture = Texture.fromBitmap(sausageBitmap, false);
            for (var i:int = 0; i < NUM_SAUSAGES; i++)
            {
                // create a Image object with our one texture
                var image:CustomImage = new CustomImage(texture);
                // set a random alpha, position, rotation
                image.alpha = Math.random();
                // define a random destination
                image.destX = Math.random()*stage.stageWidth;
```

```
            image.destY = Math.random()*stage.stageWidth;
            // define a random initial position
            image.x = Math.random()*stage.stageWidth
            image.y = Math.random()*stage.stageHeight
            image.rotation = deg2rad(Math.random()*360);
            // show it
            addChild(image);
            // store references for later
            sausagesVector[i] = image;
        }
        // main loop
        stage.addEventListener(Event.ENTER_FRAME, onFrame);
    }
    private function onFrame (e:Event):void
    {
        var lng:uint = sausagesVector.length;
        for (var i:int = 0; i < lng; i++)
        {
            // move the sausages around
            var sausage:CustomImage = sausagesVector[i];
            sausage.x -= ( sausage.x - sausage.destX ) * .1;
            sausage.y -= ( sausage.y - sausage.destY ) * .1;
            // when reached destination
            if ( Math.abs ( sausage.x - sausage.destX ) < 1 && Math.abs ( sausage.y -
            sausage.destY ) < )
            {
                sausage.destX = Math.random()*stage.stageWidth;
                sausage.destY = Math.random()*stage.stageWidth;
                sausage.rotation = deg2rad(Math.random()*360);
            }
        }
    }
}
```

之前提到过的事件冒泡机制，可以让我们在主应用中不必为每个移动着的图片侦听 Touch 事件，让代码更加简洁。取而代之的是，我们只需要对舞台对象侦听 Touch 事件就够了：

```
// we listen to the mouse movement on the stage
stage.addEventListener(TouchEvent.TOUCH, onClick);
```

如果我们测试一下事件的 target 及 currentTarget 属性，我们会发现 currentTarget 对象始终是 Stage 对象，而 target 属性则是所点击到的 Image 对象：

```
private function onClick(e:TouchEvent):void
{
    // get the touch points (can be mulitple because of multitouch)
    var touches:Vector.<Touch> = e.getTouches(this);
```

```
    var clicked:DisplayObject = e.currentTarget as DisplayObject;
    // if one finger only or the mouse
    if ( touches.length == 1 )
    {
        // grab the touch point
        var touch:Touch = touches[0];
        // detect the click/release phase
        if ( touch.phase == TouchPhase.ENDED )
        {
            // outputs : [object Stage] [object CustomImage]
            trace ( e.currentTarget, e.target );
        }
    }
}
```

正因为有了事件冒泡特性，我们往往只需要对显示对象的容器侦听 Touch 事件即可，在此例中，对 Stage 添加的事件侦听器可以在事件冒泡阶段捕捉到子对象抛出的事件。如果在事件处理函数中测试 Touch 事件的 bubble 属性，你会发现该事件是冒泡的：

```
// outputs : [object Stage] [object CustomImage] true
trace ( e.currentTarget, e.target, e.bubbles );
```

以上就是对事件冒泡机制的一个介绍。接下来让我们一起看一下 Image 对象所具备的能力吧。Image 对象除了继承了 DisplayObject 的一系列 API 之外还提供了一个特殊的属性 smoothing 来提供对图像的平滑处理功能。该属性的可能值均存放于 TextureSmoothing 类定义的静态常量中：

* **BILINEAR**：当纹理被缩放时对其应用双线性滤镜 (默认值)
* **NONE**：当纹理被缩放时不使用任何滤镜
* **TRILINEAR**：当纹理被缩放时对其应用三线性滤镜

下面是 smooth 的一个应用示例：

```
//disable filtering when scaled
image.smoothing = TextureSmoothing.NONE;
```

下图是一个图片在应用了双线性滤镜（TextureSmoothing.BILINEAR）后被放大的样子：

图 1.16
TextureSmoothing.BILINEAR

下图则是应用了三线性滤镜（TextureSmoothing.TRILINEAR）后的效果：

图 1.17
TextureSmoothing.TRILINEAR

下图是没有使用任何滤镜（TextureSmoothing.NONE）的效果：

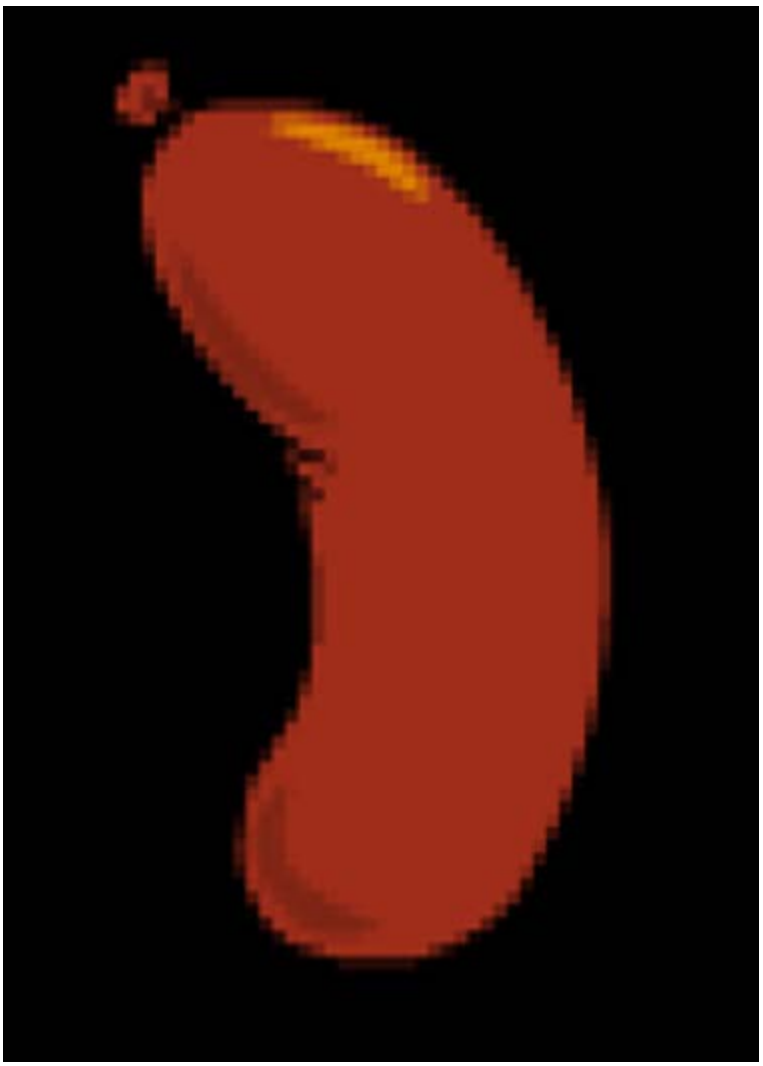

图 1.18
TextureSmoothing.NONE

可以看见，当没有用任何滤镜做平滑处理的图片是如此之酷啊【PS：酷你妹- -】
在 Image 对象中存在一个 color 属性值得大家记住，该属性允许你为一个图片指定一个颜色值。在 Image 对象中，每个像素的颜色值都是由其纹理的颜色值和你指定的 color 颜色值混合的结果。【PS：所以此 color 属性其实就是让你设置一个背景底色】这个特性就允许我们可以为图像进行着色，以创建出使用同一个纹理却有颜色差别的多个变种图片，这样就避免了为每个外观都特定地制作对应的纹理了。
下图演示了这个过程：

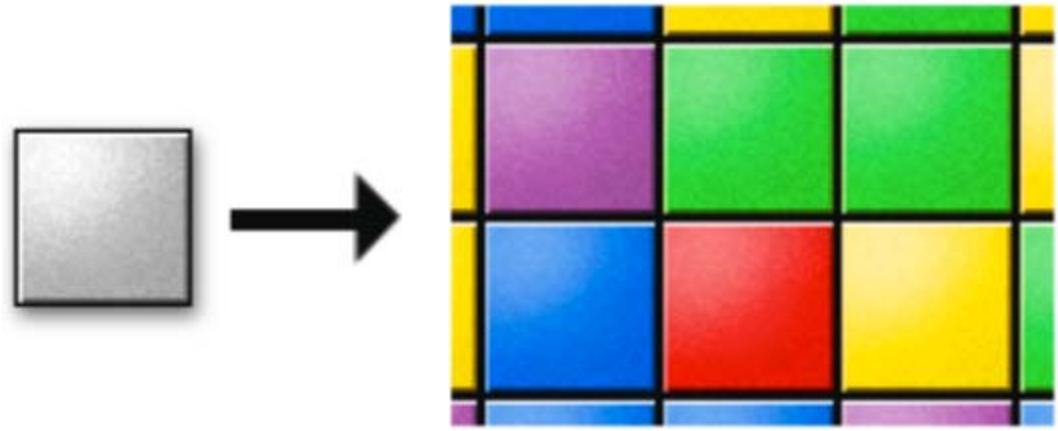

图 1.19
纹理与方块颜色混合后的效果（图片取自 *sparrow-framework.org*）

如果你想在 Starling 中使用不规则图形的碰撞检测怎么办呢？没问题，接下来看下一章节。

碰撞检测

在大多数游戏中，如果要进行碰撞检测，如果不依赖于一些物理引擎（如 Box2D，对于此引擎的使用稍后我们会提及），你可能需要自己来想办法处理一些简单的碰撞检测。当检测两个简单的圆形图形碰撞时，依靠对两圆心间距离是否小于两圆半价之和来判断就够了。在其他的一些情况下，一般用判断两个物体的矩形边框是否交汇的方法可以做简单的碰撞检测，但是如何做到精确的像素级碰撞检测呢？
下图就显示了一个典型的例子：检测两个带透明层的显示对象间的碰撞。

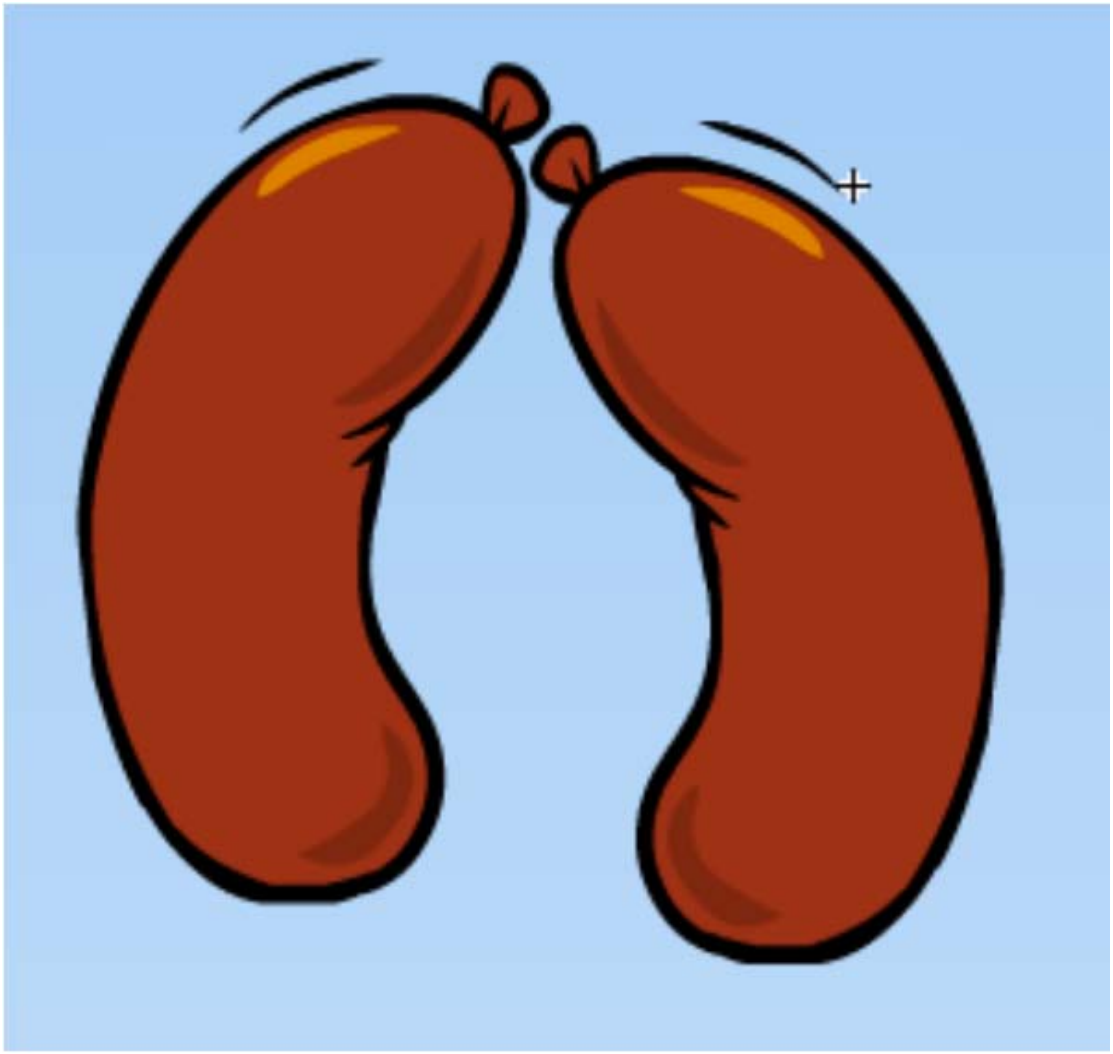

图 1.20
精确的像素级碰撞检测

在这种情况下，我们想用一种能够精确到像素级别的碰撞检测来对付这种不规则图形。如果用 ActionScript 语言自己去实现一套像素级碰撞检测，一定会在实现上花很大力气且执行效率未必高。幸运的是，在 Starling 中，材质可由 BitmapData 来生成，因此我们可用 BitmapData 对象提供的 hitTest 这个 API 来做到像素级碰撞检测。

hitTest 这个 API 是非常简单易用的，不过在第一眼看到它的时候可能会对其用法有点迷茫，让我们先看看它的方法签名：

```
public function hitTest(firstPoint:Point, firstAlphaThreshold:uint, secondObject:Object,
secondBitmapDataPoint:Point = null, secondAlphaThreshold:uint = 1):Boolean
```

注意此方法中的 secondObject 参数可以接受一个 Point、Rectangle 或 BitmapData 类型的对象，这让此方法的应用范围又广了不少。在下面的例子中，我们对两个 BitmapData 执行碰撞检测，这其实就间接地表现了在 Starling 中如何对两个 Image 对象执行像素级碰撞检测（Image 所用的 Texture 是来自于 BitmapData 对象的）。

```
if ( sausageBitmapData1.hitTest(new Point(sausageImage1.x, sausageImage1.y), 255,
sausageBitmapData2, new
Point(sausageImage2.x, sausageImage2.y), 255))
```

```
{
trace ("touched!")
}
```

参数被分为了两组。第一组两个，即第一、二个参数表示了检测源的左上角坐标及其允许的最小检测透明度，这里设置为 255，即完全不透明，表示只有完全不透明的像素会被检测到碰撞。第二组三个，分别表示检测对象、检测对象的左上角坐标及检测对象允许的最小检测透明度。

由于我们每一帧都要执行这段碰撞检测的代码，所以为了节省内存，我们不必每帧都创建一个新的 Point 示例，只需要创建两个非临时 Point 对象并每帧改变其 x,y 属性就可以了。

```
private function onFrame(event:Event):void
{
   point1.x = sausageImage1.x;
   point1.y = sausageImage1.y;
   point2.x = sausageImage2.x;
   point2.y = sausageImage2.y;
   if ( sausageBitmapData1.hitTest(point2, 255, sausageBitmapData1, point1, 255))
   {
      trace("touched!");
   }
}
```

接下来，我们一起来看看在 Starling 中如何使用在原生 AS 中常会用到的绘图 API。

绘图 API

Starling 中并没有提供像你在原生 AS 中习惯使用的 flash.display.Graphics 对象一样的绘图 API。但是在 Starling 中你可以通过 BitmapData 的 draw 方法来把使用绘图 API 绘制好的原生显示对象画到一个 BitmapData 对象中，之后用此对象作为 Texture 的数据源。

假设我们想在 Starling 中绘制一个圆形图案，你可以这么写：

```
// create a vector shape (Graphics)
var shape:flash.display.Sprite = new flash.display.Sprite();
// pick a color
var color:uint = Math.random() * 0xFFFFFF;
// set color fill
s.graphics.beginFill(color,ballAlpha);
// radius
var radius:uint = 20;
// draw circle with a specified radius
s.graphics.drawCircle(radius,radius,radius);
s.graphics.endFill();
// create a BitmapData buffer
var bmd:BitmapData = new BitmapData(radius * 2, radius * 2, true, color);
// draw the shape on the bitmap
buffer.draw(s);
// create a Texture out of the BitmapData
```

```
var texture:Texture = Texture.fromBitmapData(buffer);
// create an Image out of the texture
var image:Image = new Image(texture);
// show it!
addChild(image);
```

解决方案很简单不是吗？只需要先使用原生绘图 API 绘制好要绘制的图案（这工作是由 CPU 完成的），之后将图案复制到 Bitmap 或者 BitmapData 中并将其做成一个 texture 后上传至 GPU 即可。

下图展示了一个在 Starling 中使用绘图 API 制作大量圆形图案的例子：

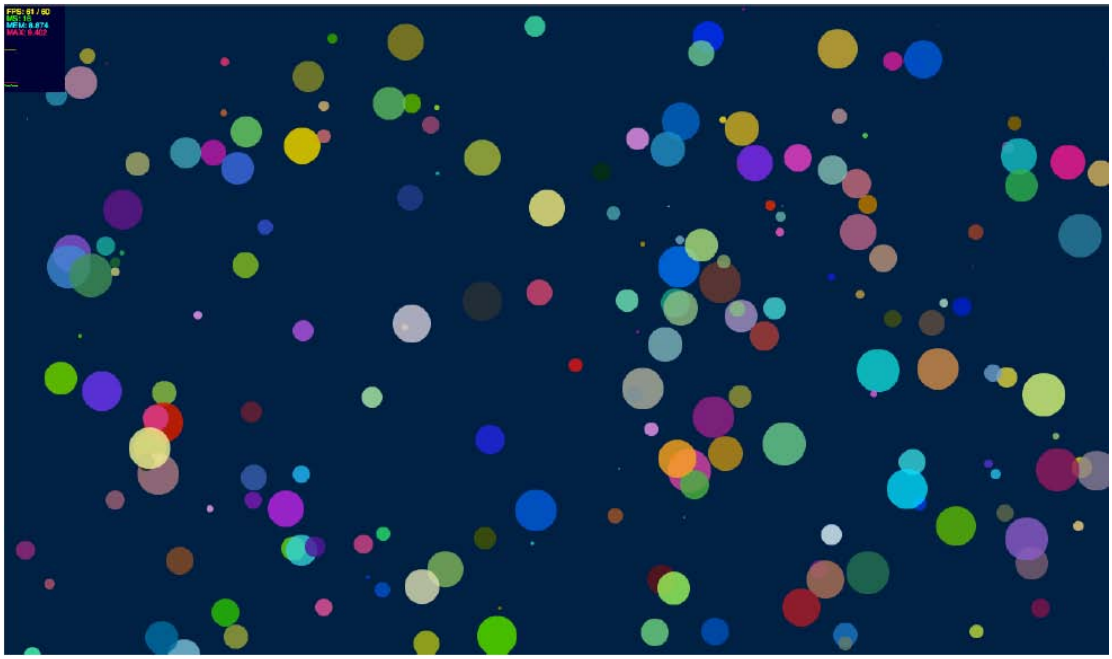

图 1.21
动态自定义图案

你有听说过 Starling 中的一个 flat sprites 的概念吗？如果没有，那就让我们一起看一看接下来的章节，探索一下这一个特性能为我们的性能提升提供多大的帮助。

Flat Sprites

Starling 中提供了一个被称作 flat sprites 的非常强大的特性（使用 Sparrow 框架进行编译的 Sprite），这个特性能极大地提升我们的应用的呈现性能。

为了更好地理解显示列表在默认情况下的工作原理，让我们一起来看下图，这张图展示了一个使用了多层嵌套的 Sprite 对象：

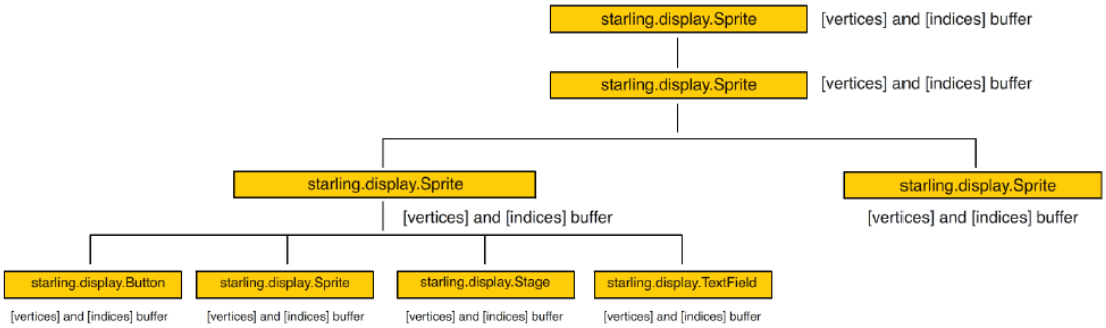

图 1.22

所有子对象都有属于自己的顶点着色器及索引缓冲区

正如你在上图看见的那样，Starling在管理显示列表的时候，每个显示对象都有专属的顶点着色器及索引缓冲区（vertex and index buffer）来进行各自的渲染工作，这样会消耗大量的运算量且在子对象繁多时会显著地影响呈现性能。

然而 Starling 提供了一种优化方案，具体做法可以理解为将全部子对象的顶点着色器及索引缓冲区集成为一个大的并由它来接管全局的渲染工作（包括容器及其子对象），且只需要调用一次绘制方法即可完成渲染的工作，就跟完成一个简单的纹理的渲染一样（如果所有子对象都共享同一个纹理的话，那自然是这样）。

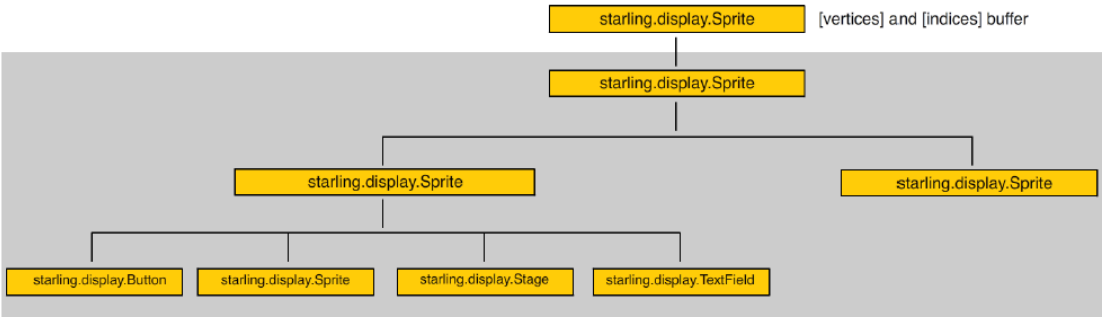

图 1.23

当 Sprite 被 flat 了之后，只需调用一次绘制方法即可完成全局的渲染工作（只在一个顶点着色器和索引缓冲区中）

你可以把这种特性理解为是在 Starling 中可用的 cacheAsBitmap（这玩意儿想必作 Flash 开发的人员应该不会陌生），用它可以做到和原生 Flash 显示列表中使用 cacheAsBitmap 相类似的效果。不过它们之间的区别在于，使用 flat sprites 不会像 cacheAsBitmap 一样只需要设置一次就一劳永逸，在原生 Flash 中，显示对象一旦被设置了 cacheAsBitmap 为 true 之后，只要改显示对象或者其子对象发生了改变，Flash 会自动生成新的位图缓存，保证显示正确的图形，但是 flat sprites 不会自动生成新的。因此你必须再次调用 flatten 方法来手动地重新生成一次才能看到改变。

下面列出与 flat sprites 特性相关的 API：

* **flatten**: 如果你想尽可能地提高存在大量嵌套的Sprite对象的话，调用此方法可以让你达到满意效果。一旦调用了此方法，Starling将会把显示树（即以一个Sprite对象为顶点，所有子对象为子节点）中的渲染工作全部统一管理，在一次绘制操作中完成全部渲染工作 。如果显示树中某个节点发生了外观改变，那么你需要再次调用此方法才能看见改变。

* **unflatten**: 禁用flat行为.
* **isFlatenned**: 判断是否使用了 flat

接下来让我们一起来试一试吧，在下面给出的代码中，我们为一个 Sprite 对象添加多个 Image 子对象并在每一帧中旋转此 Sprite：

```
package
{
    import flash.display.Bitmap;
    import starling.display.Image;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    import starling.utils.deg2rad;
    public class Game6 extends Sprite
    {
        private var container:Sprite;
        private static const NUM_PIGS:uint = 400;
        [Embed(source = "../media/textures/pig-parachute.png")]
        private static const PigParachute:Class;
        public function Game6()
        {
                addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // create the container
            container = new Sprite();
            // change the registration point
            container.pivotX = stage.stageWidth >> 1;
            container.pivotY = stage.stageHeight >> 1;
            container.x = stage.stageWidth >> 1;
            container.y = stage.stageHeight >> 1;
            // create a Bitmap object out of the embedded image
            var pigTexture:Bitmap = new PigParachute();
            // create a Texture object to feed the Image object
            var texture:Texture = Texture.fromBitmap(pigTexture);
            // layout the pigs
            for ( var i:uint = 0; i< NUM_PIGS; i++)
            {
                // create a new pig
                var pig:Image = new Image(texture);
                // random position
                pig.x = Math.random()*stage.stageWidth;
                pig.y = Math.random()*stage.stageHeight;
                pig.rotation = deg2rad(Math.random()*360);
```

```
            // nest the pig
            container.addChild ( pig );
        }
        container.pivotX = stage.stageWidth >> 1;
        container.pivotY = stage.stageHeight >> 1;
        // show the pigs
        addChild ( container );
        // on each frame
        stage.addEventListener(Event.ENTER_FRAME, onFrame);
    }
    private function onFrame (e:Event):void
    {
        // rotate the container
        container.rotation += .1;
    }
  }
}
```

在这个测试中，动画播放得算是比较流畅且能保证 60 帧/秒的帧频，不过我们还可以做一些优化，尽可能地降低对绘图方法的调用次数。因此，我们需要调用 flatten 这个 API：

```
// freeze the children
container.flatten();
```

在代码中加上上句之后，只需要调用一次绘图方法，所有的子对象都能够被绘制，这样就大大地减少了计算机的运算量。可能你在呈现性能上面没有看到明显地改观，但是你会注意到 CPU 占用率已有了明显地下降。通过大量的测试证明，使用 flatten 后，CPU 占用率能减低 10%以上。

需要注意的是，如果子对象所用纹理不一致，那么Starling会对不同的纹理进行独立地绘制工作，这样的话，flatten特性所能带来的性能提升便也不那么明显了

此特性对于手机应用也有着莫大的帮助。当然，此特性的另一个利用价值体现在它可动态地开与关。如果你需要调整外观，可以先 unflatten，待准备就绪后再调用 flatten。你可以在任何时候改变一个子对象的外观并在希望屏幕反应出这个外观变化的时候重新调用一次 flatten 即可。

这里需要提一下的是，flatten 特性只对于静态显示对象有效，Starling 暂时没有提供对于类似 MovieClip 这样的动态显示对象的 flatten 支持及类似优化方案，可能在未来的版本中能做到。

MovieClip

刚才我们一起看了 Starling 中 Sprite 对象的使用方式，那么 Starling 中是否有一种类似于 MovieClip 的动态显示对象呢？每个 Flash 开发人员都对 movieClip 的概念非常熟悉了，而且在过去的几年中，有不少开发人员使用 BitmapData 来重现了 MovieClip 的功能【PS：这样可以大大提升大量动画同屏渲染的呈现效率】

下图是组成一个 MovieClip 动画各帧的 sprite atlas：

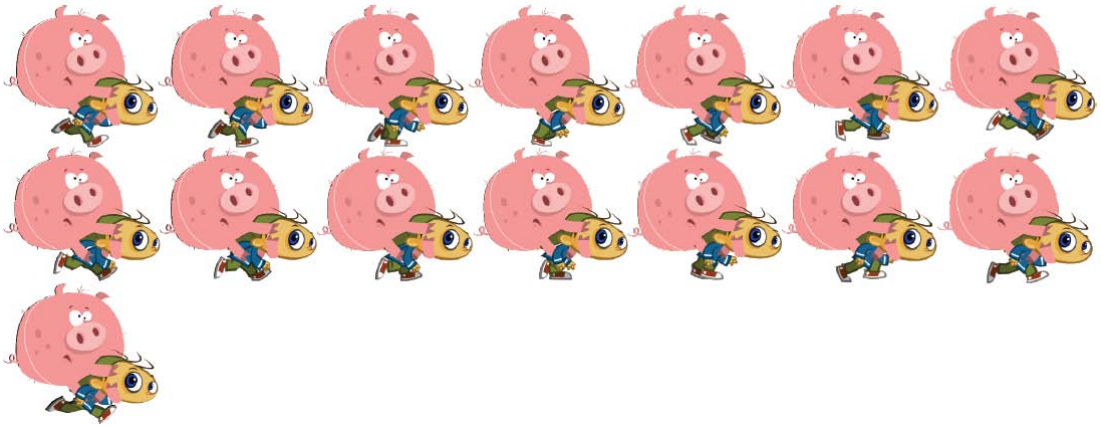

图 1.24
组成一个奔跑男孩动画的 sprite atlas

通过每帧更换纹理，我们可以重建出一个MovieClip的概念，GPU每帧都会重新采样并在屏幕上显示出来。然而，我们该如何制作组成MovieClip的素材呢？类似上图这样的sprite atlas。当然，Flash Professional工具是我们制作素材的好搭档，它可以将一个动画的每帧导出为一组图片序列，之后，我们把这一些图片扔到一些工具，如TexturePacker（http://www.texturepacker.com）中，依靠这些工具，可以把一组图片拼接到单独的一张图片中（我们称之为sprite atlas，或sprite sheet）去，以供我们在Starling的MovieClip类中使用。

下图展示了一个 sprite atlas 的组成格式：

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

图 1.25
一个 sprite atlas 中的各帧

如果你要对你的 sprite atlas 纹理使用 Mip 映射，那么你最好确保其中各帧之间的间隔起码保持 2 像素以上，否则在使用时将会出现像素重叠。意思就是 GPU 在对每帧进行采样时，会采样到相邻帧的一部分颜色。

现在，在制作纹理素材时我们需要记住Stage3D（Molehill）对其尺寸的限制条件（必须为2的整数倍），这一点其实是手机应用开发时的一个条件，现在被Stage3D带到了桌面应用的开发中来了。这个条件事实上是由OpenGL、ES2这些显卡驱动所规定的，然而Stage3D为了去驱动它们就必须得遵循这个限制。不过还好，TexturePacker这个工具提供了一个称为AutoSize（自动调整尺寸）的功能，它不仅能帮助我们很好地遵循此限制条件，还能够替我们计算出一个最佳的纹理宽、高，当然，它也能帮我们约束纹理素材的最大尺寸（对于不同的显卡驱动、Stage3D框架都有着其自规定的最大尺寸，对于Starling来说最大尺寸为2048*2048）：

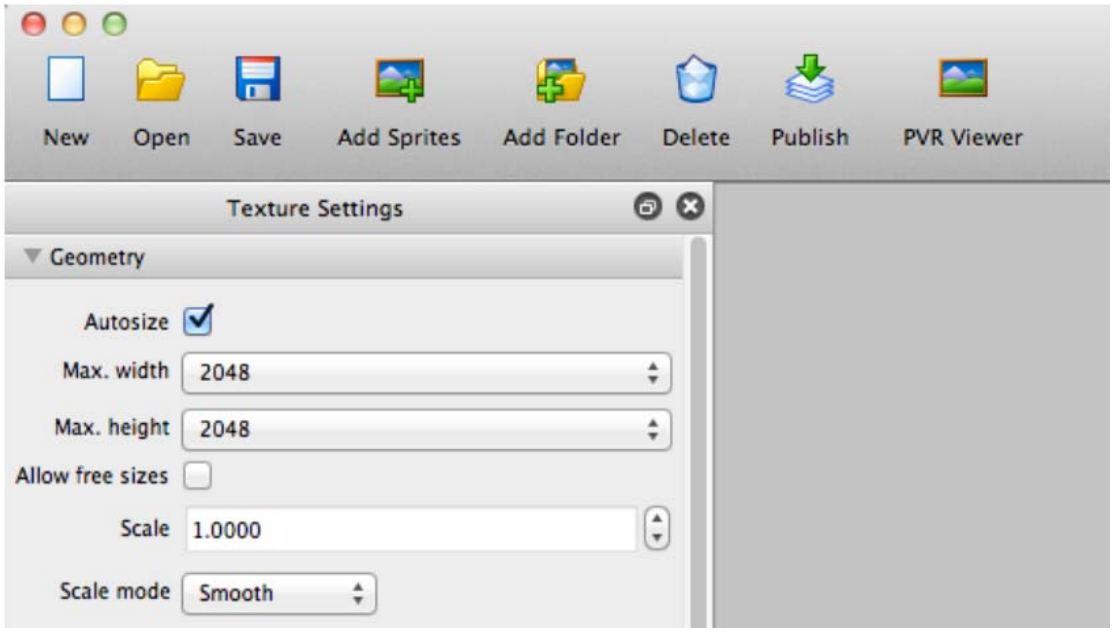

图1.26

TexturePacker中的AutoSize功能

正如之前提过的，TexturePacker可以确保你的素材尺寸为2的整数倍。如果你还是不慎用了一个长或宽不为2的整数倍的素材，那么Starling将会替你将素材拉伸至与当前长/宽值最接近的一个能被2整除的数后上传至GPU使用。

那么为了让Starling知道你所提供的Sprite Sheet中每一帧所在位置与尺寸等信息，你需要为TextureAtlas对象（Starling中用以从单个Texuture中生成一组纹理序列的类）提供一个XML文件。不过这件事TexturePacker也会替你自动完成，在导出一张Sprite Sheet的同时会导出一个记录各帧数据的文件，该文件的格式可以在XML、JSON等格式中选择。

Starling中的TextureAtlas类能识别的XML格式如下：(TexturePacker导出的XML文件就遵循此格式)

```
<?xml version="1.0" encoding="UTF-8"?>
<TextureAtlas imagePath="running-sheet.png">
    <!-- Created with TexturePacker -->
    <!-- http://texturepacker.com -->
    <!-- $TexturePacker:SmartUpdate:2b3f5fa2588769393bcea9b632749826$ -->
    <SubTexture name="running0001" x="0" y="0" width="304" height="284"/>
```

```
    <SubTexture name="running0002" x="304" y="0" width="304" height="284"/>
    <SubTexture name="running0003" x="608" y="0" width="304" height="284"/>
    <SubTexture name="running0004" x="0" y="284" width="304" height="284"/>
    <SubTexture name="running0005" x="304" y="284" width="304" height="284"/>
    <SubTexture name="running0006" x="608" y="284" width="304" height="284"/>
    <SubTexture name="running0007" x="0" y="568" width="304" height="284"/>
    <SubTexture name="running0008" x="304" y="568" width="304" height="284"/>
    <SubTexture name="running0009" x="608" y="568" width="304" height="284"/>
    <SubTexture name="running0010" x="0" y="852" width="304" height="284"/>
    <SubTexture name="running0011" x="304" y="852" width="304" height="284"/>
    <SubTexture name="running0012" x="608" y="852" width="304" height="284"/>
    <SubTexture name="running0013" x="0" y="1136" width="304" height="284"/>
    <SubTexture name="running0014" x="304" y="1136" width="304" height="284"/>
    <SubTexture name="running0015" x="608" y="1136" width="304" height="284"/>
</TextureAtlas>
```

使用纹理序列来组成动画的好处在于，我们得到了动画各帧的掌控权，甚至可以动态调整动画的帧频，这样就允许我们为每个动画设置独立的帧频。好消息是，在 Starling 中的 MovieClip 类就运用了这种方式，并提供了相应属性允许我们来设置其帧频，下面是其构造函数签名：

```
public function MovieClip(textures:Vector.<Texture>, fps:Number=12)
```

在下面的代码中，我们为一个 TextureAtlas 类提供了一个 Sprite Sheet 作为影片各帧的动画源：

```
[Embed(source = "../media/textures/running-sheet.png")]
private const SpriteSheet:Class;
var bitmap:Bitmap = new SpriteSheet();
var texture:Texture = Texture.fromBitmap(bitmap);
```

之后，我们再为其提供一个 XML 文件来描述 spritesheet 各帧信息：

```
[Embed(source="../media/textures/running-sheet.xml", mimeType="application/octet-stream")]
public const SpriteSheetXML:Class;
var xml:XML = XML(new spriteSheetXML());
var sTextureAtlas:TextureAtlas = new TextureAtlas(texture, xml);
```

最后，我们就可以提取出 spritesheet 图片中，组成一个男孩奔跑动画的帧序列：

```
var frames:Vector.<Texture> = sTextureAtlas.getTextures("running_");
```

注意，在 SpriteSheet 配套的 XML 描述文件中记录的每一帧都有其名字，那么对于组成一个动画的帧来说你应该为它们设置相同的名字前缀（如 run_1, run_2, run_3 组成一组动画; walk_1, walk_2, walk_3 组成另一组动画），而向 getTextures 方法传入的参数就是这个统一前缀，TextureAtlas 类会以此为依据，提取出相同前缀的纹理给你，而这一组纹理恰好是我们用来组成一个动画所需的纹理序列。当然，你也可以传入”jump”、”fire”等作为“前缀”参

数传入 getTextures 方法，如果你 XML 文件中有定义带有此前缀的这么一组帧的话。
下面让我们来看完整的一段应用代码：

```
package
{
    import flash.display.Bitmap;
    import starling.core.Starling;
    import starling.display.MovieClip;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    import starling.textures.TextureAtlas;
    public class Game3 extends Sprite
    {
        private var mMovie:MovieClip;
        [Embed(source="../media/textures/running-sheet.xml", mimeType="application/octet-stream")]
        public static const SpriteSheetXML:Class;
        [Embed(source = "../media/textures/running-sheet.png")]
        private static const SpriteSheet:Class;
        public function Game3()
        {
                addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // creates the embedded bitmap (spritesheet file)
            var bitmap:Bitmap = new SpriteSheet();
            // creates a texture out of it
            var texture:Texture = Texture.fromBitmap(bitmap);
            // creates the XML file detailing the frames in the spritesheet
            var xml:XML = XML(new SpriteSheetXML());
            // creates a texture atlas (binds the spritesheet and XML description)
            var sTextureAtlas:TextureAtlas = new TextureAtlas(texture, xml);
            // retrieve the frames the running boy frames
            var frames:Vector.<Texture> = sTextureAtlas.getTextures("running_");
            // creates a MovieClip playing at 40fps
            mMovie = new MovieClip(frames, 40);
            // centers the MovieClip
            mMovie.x = stage.stageWidth - mMovie.width >> 1;
            mMovie.y = stage.stageHeight - mMovie.height >> 1;
            // show it
            addChild ( mMovie );
        }
    }
}
```

运行上面的代码，我们能得到如下结果：

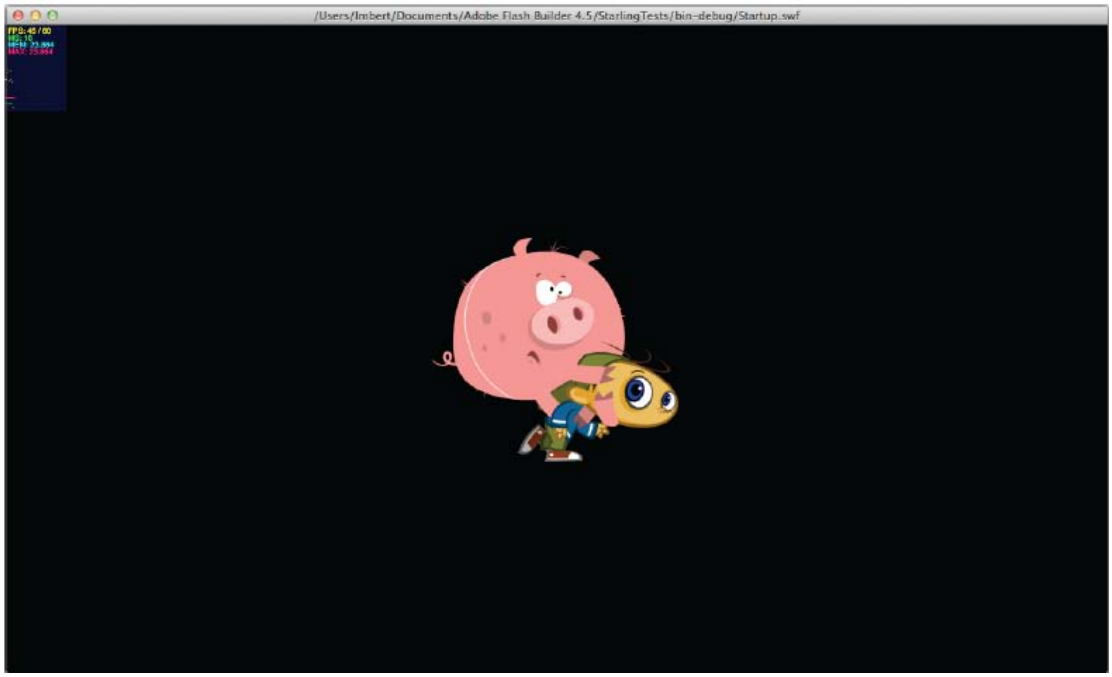

图 1.27
我们可爱的奔跑男孩君被渲染出来喽

稍后，我们会讲一种简单的资源缓存技术，使用这种技术，可以让我们的应用在其生命周期内尽可能少地创建对象数目以节省内存，提高性能。
如果你想反转这个动画对象，你可以使用我们在原生 Flash 中一样的做法，设置 scaleX 属性值为其相反数即可。当然，为了保证其位置不变，还需要设置 x 值到正确的位置：

```
mMovie = new MovieClip(frames, 40);
mMovie.scaleX = -1;
mMovie.x = (stage.stageWidth - mMovie.width >> 1) + mMovie.width;
mMovie.y = stage.stageHeight - mMovie.height >> 1;
```

这段代码运行的结果如下图所示：

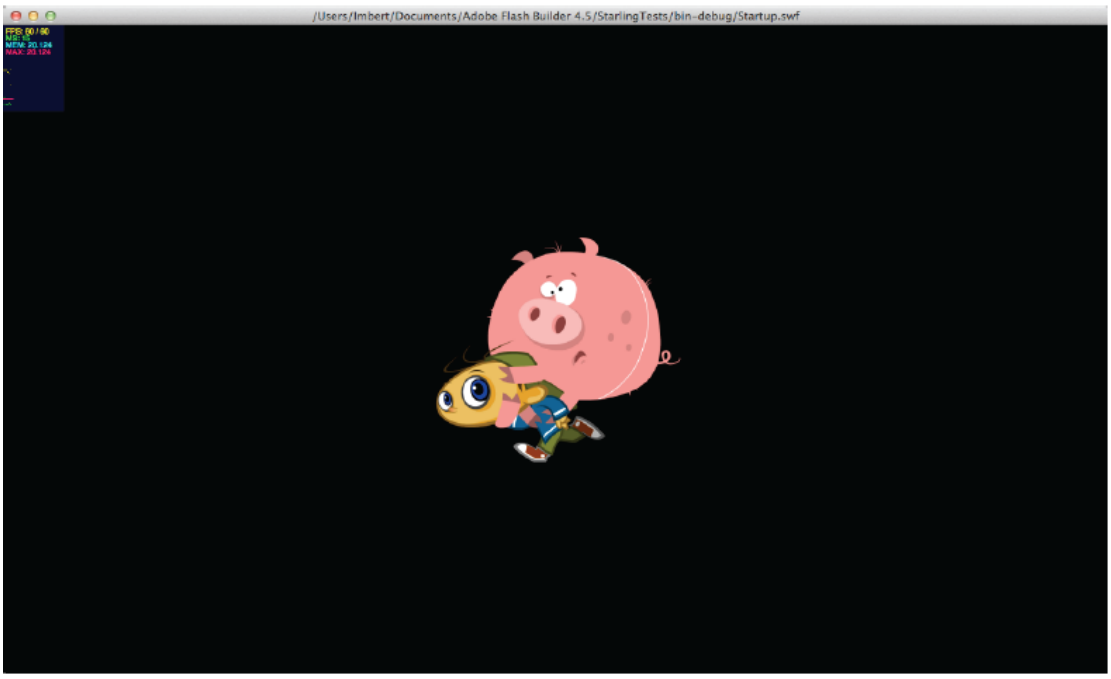

图 1.28
反转的影片剪辑

在之前提过，一个较好，也是使用比较普遍的做法是将所有素材集成在单独的一张 SpriteSheet 图片中，为什么这么说呢？
首先，这样做很方便，所有素材都集成在一个单独的纹理文件中。仅用一张纹理文件，可以最小化 FlashPlayer 向 GPU 上传的图片次数。记住，向 GPU 上传图片的工作是很耗费性能的，尤其是对于手机设备来说。所以，向 GPU 上传图片的次数越少越好。最后，切换纹理也是一个比较耗费性能的工作，使用一个单独纹理可以让你避免频繁地切换纹理。

Texture Atlas

刚才，我们已对sprite atlas的概念有了一定的了解，现在，我将为大家介绍另一个类——Texture Atlas。在一个Texture Atlas对象中，所有的素材都被集成在一个单独的纹理当中。在下图中，我们在刚才使用的示例图片素材中增加了一个新的帧序列：

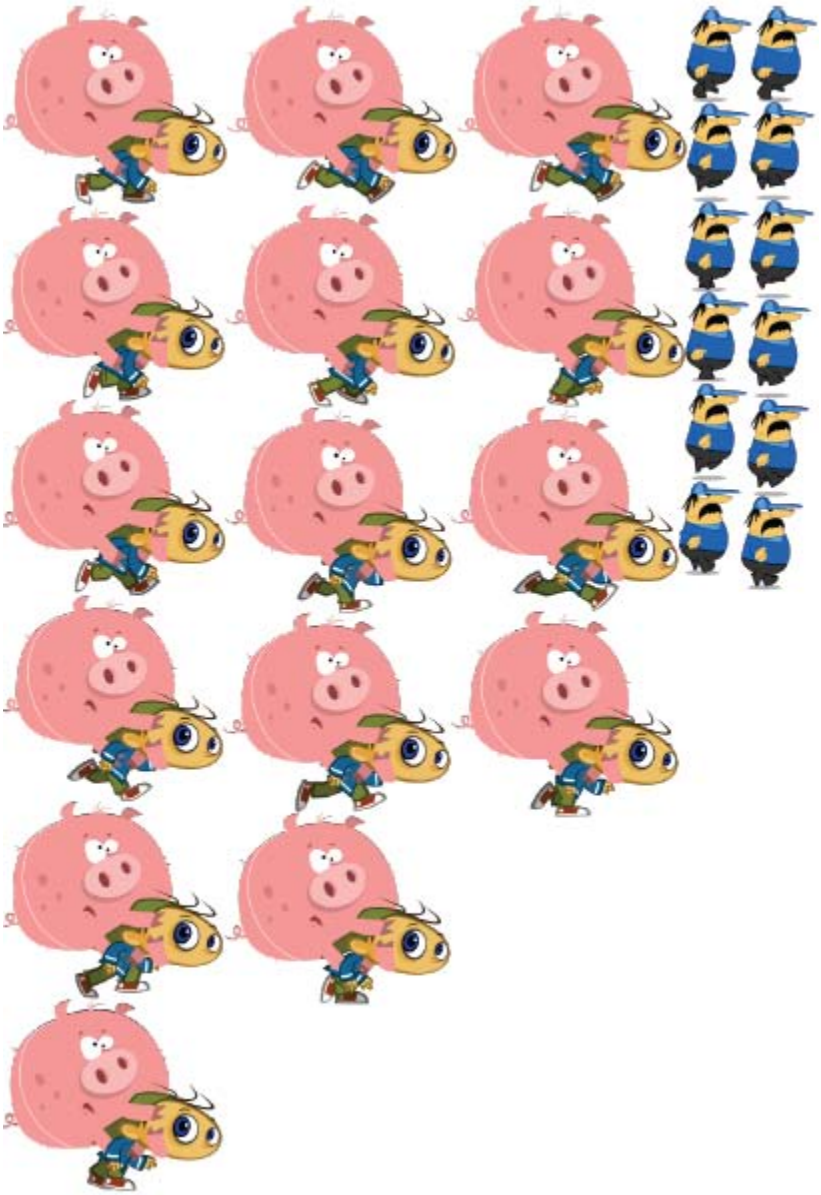

图1.29

Texture Atlas包含了我们所有的素材

那么现在我们提供的XML描述文件中就会多出对于屠夫（就是刚刚新增的那一组帧序列所组成的动画人物）动画的描述信息：

```
<?xml version="1.0" encoding="UTF-8"?>
<TextureAtlas imagePath="running-sheet.png">
    <!-- Created with TexturePacker -->
    <!-- http://texturepacker.com -->
    <!-- $TexturePacker:SmartUpdate:5aa8dfdc90d616e76e06b3079d2c5e80$ -->
    <SubTexture name="french-butcher_01" x="930" y="486" width="77" height="122"
    frameX="-106" frameY="-33"
    frameWidth="275" frameHeight="200"/>
    <SubTexture name="french-butcher_02" x="845" y="588" width="77" height="118"
```

```
    frameX="-106" frameY="-37"
    frameWidth="275" frameHeight="200"/>
    ...
```

现在，我们就可以使用 TextureAtlas 类的 getTextures 这个 API 来提取出这两个动画所需帧了：

```
// retrieve the frames the running boy frames
var frames:Vector.<Texture> = sTextureAtlas.getTextures("running_");
// retrieve the frames the running butcher
var framesButcher:Vector.<Texture> = sTextureAtlas.getTextures("french-butcher_");
// creates a MovieClip playing at 40fps
mMovie = new MovieClip(frames, 40);
// creates a MovieClip playing at 25
mMovieButcher = new MovieClip(framesButcher, 25);
// positiomns them
mMovie.x = stage.stageWidth - mMovie.width >> 1;
mMovie.y = stage.stageHeight - mMovie.height >> 1;
mMovieButcher.x = mMovie.x + mMovie.width + 10;
mMovieButcher.y = mMovie.y;
// show them
addChild ( mMovie );
addChild ( mMovieButcher );
```

运行代码后可以看见我们的男孩和屠夫已经出现在了屏幕上：

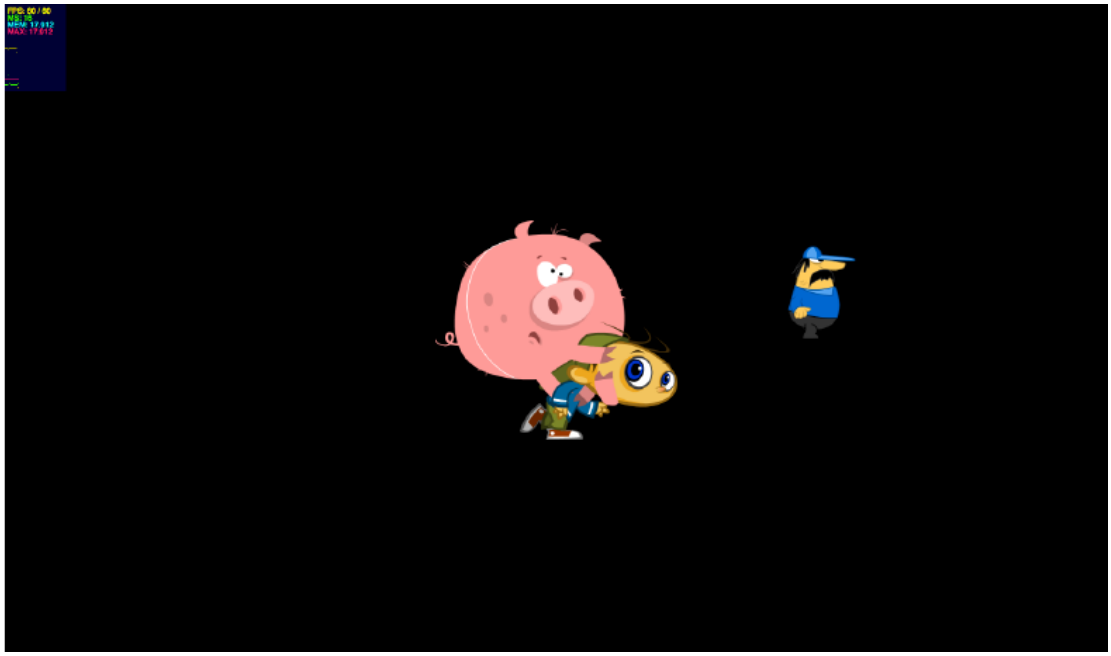

图 1.30

所有动画的素材都取自同一个 Sprite Sheet(texture atlas)

我们注意到，虽然两个影片剪辑都被创建并添加到了舞台上，但他们并没有开始播放动画。那么为了让他们播放起来，我们需要使用一个 Juggler 对象【PS：此对象是 Starling 中用以

管理动画的类，稍后会讲到】。
在 Starling 类中存在一个静态的 juggler 属性，我们可以使用这个默认存在的 juggler 对象来管理上面创建的两个影片剪辑。
我们添加以下两行代码来让我们的男孩和屠夫动起来：

```
// animate them
Starling.juggler.add ( mMovie );
Starling.juggler.add ( mMovieButcher );
```

一旦我们的动画对象被添加到了 juggler 中，它们就会动起来了！当然，如果你想在某个时候暂停或停止动画的播放，只需要使用下面的两个 API：

```
// pause or stop the playback
mMovie.pause();
mMovie.stop();
```

注意这两个 API 的细微差别，调用 pause 后，播放头会停止在当前播放到的帧处，但是调用 stop 后，播放头会被重置回第一帧的位置。
接下来让我们一起看看 Starling 中的 MovieClip 对象提供的全部可用 API，有一些 API 对于 Flash 开发人员来说已经很熟悉了，有一些是额外提供的但是非常有用，如设置独立的帧频、运行时替换或添加帧等等。

* **currentFrame**：指示当前帧序号
* **fps**：默认情况下的帧频，即每秒播放的帧数
* **isPlaying**：判断当前影片剪辑是否正在播放
* **loop**：判断当前影片剪辑是否会循环播放
* **numFrames**：当前影片剪辑包含的总帧数
* **totalTime**：播完一次影片所需时间，单位为秒
* **addFrame**：为影片剪辑增加一帧至时间轴末尾处，你可以设置该帧的持续时间及播放到该帧时需要发出的声音
* **addFrameAt**：添加一帧到指定位置
* **getFrameDuration**：获取某一帧的持续时间（单位为秒）
* **getFrameSound**：获取某一帧播放的声音
* **getFrameTexture**：获取某一帧所用纹理
* **pause**：暂停影片播放
* **play**：开始播放影片．不过在此之前你需要确保影片对象被添加到了一个Juggler对象中
* **removeFrameAt**：移出指定位置处的帧
* **setFrameDuration**：设置某一帧的持续时间（单位为秒）
* **setFrameSound**：设置某一帧播放的声音
* **setFrameTexture**：设置某一帧所用纹理

我们不会逐个讲解这些API的使用方式，但是我们马上会介绍一些比较有用的API，如addFrameAt、removeFrameAt，使用这两个API，允许我们在运行时添加或移除帧，还有一些API允许我们为每一帧设置不同的持续时间，当然，还有一些辅助的API，如isPlaying和loop。
在下面的代码中，我们需要让第五帧的持续时间为2秒：

```
// frame 5 will length 2 seconds
mMovie.setFrameDuration(5, 2);
```

我们还可以动态地为某一帧添加声音：

```
// frame 5 will length 2 seconds and play a sound when reached
mMovie.setFrameDuration(5, 2);
mMovie.setFrameSound(5, new StepSound() as Sound);
```

多亏有了这些 API，让我们可以用嵌入的或外部加载的素材来在运行时动态调整 movieClip 的帧结构，这让 movieClip 在 Starling 中显得更具威力。
在一种常见的情境中会用得上addFrameAt、removeFrameAt这些API，那就是，你需要制作一个多状态的动画，【PS：如一个人物走动的动画，东南西北每个方向都有一套独立的走动动画】如果在原生Flash中，你必须创建多个MovieClip类分别代表每个状态的动画，然后把这些MovieClip类依次添加到一个共有的MovieClip类的时间轴上，若要切换状态，只需要让这个共有的MovieClip类跳帧到某个状态所在帧即可。不过在Starling中，MovieClip对象可不是容器，其中无法再addChild任何子类，因此要在Starling中实现多状态动画就必须动态地改变其中各帧了。
下面给出一个例子来演示多状态动画，我们将使用键盘来控制动画状态的改变，就像以前咱们玩的很多游戏一样：

```
package
{
    import flash.display.Bitmap;
    import flash.ui.Keyboard;
    import starling.animation.Juggler;
    import starling.core.Starling;
    import starling.display.MovieClip;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.events.KeyboardEvent;
    import starling.textures.Texture;
    import starling.textures.TextureAtlas;
    public class Game3 extends Sprite
    {
        private var mMovie:MovieClip;
        private var j:Juggler;
        [Embed(source="../media/textures/running-sheet.xml", mimeType="application/octet-stream")]
        public static const SpriteSheetXML:Class;
        [Embed(source = "../media/textures/running-sheet.png")]
        private static const SpriteSheet:Class;
        public function Game3()
        {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
```

```
            // creates the embedded bitmap (spritesheet file)
            var bitmap:Bitmap = new SpriteSheet();
            // creates a texture out of it
            var texture:Texture = Texture.fromBitmap(bitmap);
            // creates the XML file detailing the frames in the spritesheet
            var xml:XML = XML(new SpriteSheetXML());
            // creates a texture atlas (binds the spritesheet and XML description)
            var sTextureAtlas:TextureAtlas = new TextureAtlas(texture, xml);
            // retrieve the frames the running boy frames
            var frames:Vector.<Texture> = sTextureAtlas.getTextures("running_");
            // creates a MovieClip playing at 40fps
            mMovie = new MovieClip(frames, 40);
            // centers the MovieClip
            mMovie.x = stage.stageWidth - mMovie.width >> 1;
            mMovie.y = stage.stageHeight - mMovie.height >> 1;
            // show it
            addChild ( mMovie );
            // animate it
            Starling.juggler.add ( mMovie );
            // on key down
            stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        }
        private function onKeyDown(e:KeyboardEvent):void
        {
            // repositions the boy accordingly
            if ( mMovie.scaleX == -1 )
                mMovie.x -= mMovie.width;
            // if right key or left key
            var state:int;
            if ( e.keyCode == Keyboard.RIGHT )
                state = 1;
            else if ( e.keyCode == Keyboard.LEFT )
                state = -1;
            // flip the running boy
            mMovie.scaleX = state;
            // repositions the boy accordingly
            if ( mMovie.scaleX == -1 )
                mMovie.x = mMovie.x + mMovie.width;
        }
    }
}
```

如果你想知道一个影片剪辑何时播放完毕，可以侦听 Event.COMPLETE 事件：

```
// listen to the end of the animation
mMovie.addEventListener(Event.MOVIE_COMPLETED, onAnimationComplete);
```

在这个例子里我们再一次用到了 Juggler 对象，那么接下来让我们来系统地看看它的工作原理及其到底能干些什么。

Juggler

Juggler 类允许我们控制所有实现了 IAnimatable 接口的对象的动画播放。MovieClip 类实现了该接口，你也可以自定义一个动画类在 Starling 中播放，你所要做的，仅仅是让你的自定义类实现 IAnimatable 接口，然后重载 advanceTime 方法即可。Starling 粒子插件（用以在 Starling 中实现粒子效果，在本教程最后会介绍）就是这么实现的。
下面代码是 MovieClip 类实现其动画功能的主要逻辑。它每一帧都会切换纹理。看起来有点像我们在原生 Flash 中用切换一个 Bitmap 的 bitmapData 属性来实现动画的方式：

```
// IAnimatable
public function advanceTime(passedTime:Number):void
{
    if (mLoop && mCurrentTime == mTotalTime) mCurrentTime = 0.0;
    if (!mPlaying || passedTime == 0.0 || mCurrentTime == mTotalTime) return;
    var i:int = 0;
    var durationSum:Number = 0.0;
    var previousTime:Number = mCurrentTime;
    var restTime:Number = mTotalTime - mCurrentTime;
    var carryOverTime:Number = passedTime > restTime ? passedTime - restTime : 0.0;
    mCurrentTime = Math.min(mTotalTime, mCurrentTime + passedTime);
    for each (var duration:Number in mDurations)
    {
        if (durationSum + duration >= mCurrentTime)
        {
            if (mCurrentFrame != i)
            {
                mCurrentFrame = i;
                updateCurrentFrame();
                playCurrentSound();
            }
            break;
        }
        ++i;
        durationSum += duration;
    }
    if (previousTime < mTotalTime && mCurrentTime == mTotalTime &&
    hasEventListener(Event.MOVIE_COMPLETED))
    {
        dispatchEvent(new Event(Event.MOVIE_COMPLETED));
```

```
    }
    advanceTime(carryOverTime);
}
```

下面列出 Juggler 中所有可用的 API：

* **add**：添加一个动画对象到 juggler
* **advanceTime**：如果你需要手动调控Juggler的主循环逻辑的话，调用之.
* **delayCall**：延迟调用某个对象的某个方法
* **elapsedTime**：指示一个juggler对象的完整生命周期时间
* **isComplete**：指示一个Juggler的状态，默认情况下它总是返回flase.
* **purge**：一次性移除全部子对象
* **remove**：从juggler中移除一个对象
* **removeTweens**：移除全部类型为 Tween 的且存在指定目标的对象

Juggler 对象中的另一个有趣的特性是 delayCall，它可以延迟调用某个方法。在下面的示例代码中，我们延迟一个显示对象从其父类中移除的时间。

```
juggler.delayCall(object.removeFromParent, 1.0);
```

有些时候，你会需要另一个 Juggler 对象来管理另一些动画对象。比如当你的游戏暂停的时候，会弹出一个菜单面板到你暂停着的游戏面板之上，此时你就需要另一个 Juggler 来负责管理该菜单面板中的动画了。实现起来也非常简单，你只需要创建一个 Juggler 对象并每帧调用其 advanceTime 方法即可。
如果你想遵循上面的这种设计方式，你就需要在你的游戏中，为每个独立的模块或是窗口创建一个 Juggler 对象。当玩家按下“暂停”键的时候，你没必要逐个暂停所有面板的 Juggler，直接调用 Starling 对象的 stop 方法即可做到，当你调用此方法后 Starling 将会停止重绘及 ENTER_FRAME 事件的派发。

```
Starling.current.stop();
```

如果你想在按下“暂停”按钮的时候不会暂停整个游戏的动画，你可以用不同的 Juggler 对象来管理不同的游戏模块（菜单、背景及游戏主界面），这样就让你在暂停或者其他情况下有选择的余地，决定哪个模块该被暂停或是恢复。
在下面的代码中，我们创建了一个自定义类 BattleScene 该类将作为游戏的战场模块：

```
package
{
    import starling.animation.Juggler;
    import starling.display.Sprite;
    public class BattleScene extends Sprite
    {
        private var juggler:Juggler;
        public function BattleScene()
        {
            juggler = new Juggler();
```

```
        }
        // this API will be called from outside
        // stop calling it will pause the content played by this Juggler in this sprite (BattleScene)
        public function advanceTime ( time:Number ):void
        {
            juggler.advanceTime( time );
        }
        public override function dispose():void
        {
            juggler.purge();
            super.dispose();
        }
    }
}
```

好了，当外部想要激活此面板时，只需要侦听 EnterFrameEvent.EVENT 事件并在事件侦听器中，以 EnterFrameEvent 事件对象的 passedTime 属性作为参数调用此面板对象的 advanceTime 方法即可。

```
private function onFrame(event:EnterFrameEvent):void
{
    if ( paused )
        battle.advanceTime( event.passedTime );
}
```

当游戏被暂停后，我们需要停止每帧对 advanceTime 方法的调用，这样就可以停止战场模块的动画内容播放。当然，如果此时我们需要让菜单面板弹出并播放此面板中的动画，只需要添加如下几句：

```
private function onFrame(event:EnterFrameEvent):void
{
    if ( paused )
            alertBox.advanceTime ( event.passedTime );
    else battle.advanceTime( event.passedTime );
    dashboard.advanceTime ( event.passedTime );
}
```

有了如上代码，你只需要更改 paused 属性的值即可控制游戏的暂停/继续行为。
接下来让我们一起看看 Starling 中另一个在交互行为中扮演重要角色的类：Button。

Button

Starling 中已内置了按钮类供开发人员使用，先看看它的签名：

```
public function Button(upState:Texture, text:String="", downState:Texture=null)
```

默认情况下，Button 对象中有一个内置的 TextField 对象来负责文本的显示，且该文本默认摆放在按钮的居中位置。在下面的代码中，我们用一个嵌入资源作为皮肤来创建了一个简单的按钮：

```
package
{
    import flash.display.Bitmap;
    import starling.display.Button;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    public class Game4 extends Sprite
    {
        [Embed(source = "../media/textures/button_normal.png")]
        private static const ButtonTexture:Class;
        public function Game4()
        {
                addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // create a Bitmap object out of the embedded image
            var buttonSkin:Bitmap = new ButtonTexture();
            // create a Texture object to feed the Button object
            var texture:Texture = Texture.fromBitmap(buttonSkin);
            // create a button using this skin as up state
            var myButton:Button = new Button(texture, "Play");
            // createa container for the menu (buttons)
            var menuContainer:Sprite = new Sprite();
            // add the button to our container
            menuContainer.addChild(myButton);
            // centers the menu
            menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
            menuContainer.y = stage.stageHeight - menuContainer.height >> 1;
            // show the button
            addChild(menuContainer);
        }
    }
}
```

注意到我们这里再次使用了 fromBitmap 方法来完成从嵌入资源获取按钮皮肤的工作。

```
// create a Texture object to feed the Button object
var texture:Texture = Texture.fromBitmap(buttonSkin);
```

接下来让我们把包含在一个 Vector 对象中的文本做成一个菜单，只需要使用一个简单的循环即可完成：

```
package
{
    import flash.display.Bitmap;
    import starling.display.Button;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    public class Game4 extends Sprite
    {
        [Embed(source = "../media/textures/button_normal.png")]
        private static const ButtonTexture:Class;
        // sections
        private var _sections:Vector.<String> = Vector.<String>(["Play", "Options", "Rules", "Sign in"]);
        public function Game4()
        {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // create a Bitmap object out of the embedded image
            var buttonSkin:Bitmap = new ButtonTexture();
            // create a Texture object to feed the Button object
            var texture:Texture = Texture.fromBitmap(buttonSkin);
            // createa container for the menu (buttons)
            var menuContainer:Sprite = new Sprite();
            var numSections:uint = _sections.length
            for (var i:uint = 0; i< 4; i++)
            {
                // create a button using this skin as up state
                var myButton:Button = new Button(texture, _sections[i]);
                // bold labels
                myButton.fontBold = true;
                // position the buttons
                myButton.y = myButton.height * i;
                // add the button to our container
                menuContainer.addChild(myButton);
            }
            // centers the menu
```

```
            menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
            menuContainer.y = stage.stageHeight - menuContainer.height >> 1;
            // show the button
            addChild(menuContainer);
        }
    }
}
```

测试这段代码，我们可以得到如下结果：

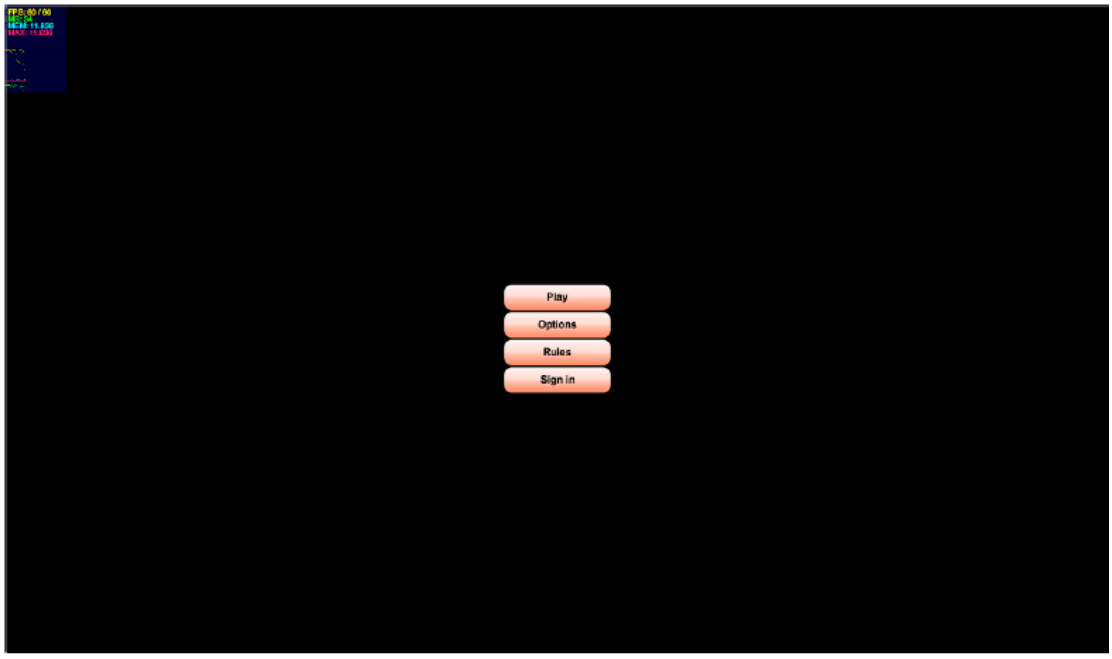

图 1.31
一个简单的由多个按钮组成的菜单

在这个例子中，我们没有用一个 SpriteSheet 来设置一个按钮的全部状态的皮肤（普通状态、按下状态及鼠标移入状态）。而是通过 fromBitmap 方法直接向 GPU 上传了一个 Texture 对象作为按钮的唯一皮肤。如果你准备为全部按钮都用同一个皮肤的话这么做当然没什么问题。不过一个更加好的习惯是将一个按钮的全部状态素材都放在一个 SpriteSheet 中，就像之前我们在创建男孩及屠夫的 movieClip 例子中做的那样。
现在，我们来看看 Button 对象提供的全部 API：

* **alphaWhenDisabled**：当按钮处于不可用状态时的透明度
* **downState**：当按钮被按下时的皮肤纹理
* **enabled**：此属性决定按钮是否可用、可交互
* **fontBold**：按钮文本是否为粗体
* **fontColor**：按钮文本的颜色
* **fontName**：按钮文本所用字体。可以是一个系统字体也可以是一个已经注册了的位图字体
* **fontSize**：按钮文本大小
* **scaleWhenDown**：按钮按下时将被缩放到的值。如果你设置了downState，那么按

钮在按下后将不会被缩放
* **text**：按钮显示的文本
* **textBounds**：按钮文本所在区域
* **upState**：当按钮未产生交互时的皮肤纹理

与原生 Flash 相反的一点在于，Starling 中的 Button 类是 DisplayObjectContainer 类的子类，这意味着你创建的 Button 按钮外观将不受限于其提供的几个皮肤属性。你可以像在用其他容器类一样，往其中加任何你想加入的东西，把你的按钮布置成你想要的效果。
注意，一个 Button 对象将会在你点击它的时候派发一个特殊的事件：Event.TRIGGERED:

```
// listen to the Event.TRIGGERED event
myButton.addEventListener(Event.TRIGGERED, onTriggered);
private function onTriggered(e:Event):void
{
      trace ("I got clicked!");
}
```

Event.TRIGGERED 事件是一个冒泡事件，你可以在事件派发者的父容器中侦听到它：

```
package
{
   import flash.display.Bitmap;
   import starling.display.Button;
   import starling.display.Sprite;
   import starling.events.Event;
   import starling.textures.Texture;
   public class Game4 extends Sprite
   {
      [Embed(source = "../media/textures/button_normal.png")]
      private static const ButtonTexture:Class;
      // sections
      private var _sections:Vector.<String> = Vector.<String>(["Play", "Options", "Rules", "Sign in"]);
      public function Game4()
      {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
      }
      private function onAdded (e:Event):void
      {
         // create a Bitmap object out of the embedded image
         var buttonSkin:Bitmap = new ButtonTexture();
         // create a Texture object to feed the Button object
         var texture:Texture = Texture.fromBitmap(buttonSkin);
         // createa container for the menu (buttons)
         var menuContainer:Sprite = new Sprite();
```

```
            var numSections:uint = _sections.length
            for (var i:uint = 0; i< 4; i++)
            {
                // create a button using this skin as up state
                var myButton:Button = new Button(texture, _sections[i]);
                // bold labels
                myButton.fontBold = true;
                // position the buttons
                myButton.y = myButton.height * i;
                // add the button to our container
                menuContainer.addChild(myButton);
            }
            // catch the Event.TRIGGERED event
            menuContainer.addEventListener(Event.TRIGGERED, onTriggered);
            // centers the menu
            menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
            menuContainer.y = stage.stageHeight - menuContainer.height >> 1;
            // show the button
            addChild(menuContainer);
        }
        private function onTriggered(e:Event):void
        {
            // outputs : [object Sprite] [object Button]
            trace ( e.currentTarget, e.target );
            // outputs : triggered!
            trace ("triggered!");
        }
    }
}
```

上述代码创建了几个使用自定义皮肤的按钮，接下来让我们一起来创建一个滚动着的背景：

```
package
{
    import flash.display.Bitmap;
    import starling.display.Button;
    import starling.display.Image;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    public class Game4 extends Sprite
    {
        [Embed(source = "../media/textures/button_normal.png")]
        private static const ButtonTexture:Class;
```

```
[Embed(source = "../media/textures/background.jpg")]
private static const BackgroundImage:Class;
private var backgroundContainer:Sprite;
private var background1:Image;
private var background2:Image;
// sections
private var sections:Vector.<String> = Vector.<String>(["Play", "Options", "Rules", "Sign in"]);
public function Game4()
{
        addEventListener(Event.ADDED_TO_STAGE, onAdded);
}
private function onAdded (e:Event):void
{
    // create a Bitmap object out of the embedded image
    var buttonSkin:Bitmap = new ButtonTexture();
    // create a Texture object to feed the Button object
    var texture:Texture = Texture.fromBitmap(buttonSkin);
    // create a Bitmap object out of the embedded image
    var background:Bitmap = new BackgroundImage();
    // create a Texture object to feed the Image object
    var textureBackground:Texture = Texture.fromBitmap(background);
    // container for the background textures
    backgroundContainer = new Sprite();
    // create the images for the background
    background1 = new Image(textureBackground);
    background2 = new Image(textureBackground);
    // positions the second part
    background2.x = background1.width;
    // nest them
    backgroundContainer.addChild(background1);
    backgroundContainer.addChild(background2);
    // show the background
    addChild(backgroundContainer);
    // create container for the menu (buttons)
    var menuContainer:Sprite = new Sprite();
    var numSections:uint = sections.length
    for (var i:uint = 0; i< 4; i++)
    {
        // create a button using this skin as up state
        var myButton:Button = new Button(texture, sections[i]);
        // bold labels
        myButton.fontBold = true;
        // position the buttons
        myButton.y = myButton.height * i;
```

```
            // add the button to our container
            menuContainer.addChild(myButton);
        }
        // catch the Event.TRIGGERED event
        // catch the Event.TRIGGERED event
        menuContainer.addEventListener(Event.TRIGGERED, onTriggered);
        // on each frame
        stage.addEventListener(Event.ENTER_FRAME, onFrame);
        // centers the menu
        menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
        menuContainer.y = stage.stageHeight - menuContainer.height >> 1;
        // show the button
        addChild(menuContainer);
    }
    private function onTriggered(e:Event):void
    {
        // outputs : [object Sprite] [object Button]
        trace ( e.currentTarget, e.target );
        // outputs : triggered!
        trace ("triggered!");
    }
    private function onFrame (e:Event):void
    {
        // scroll it
        backgroundContainer.x -= 10;
        // reset
        if ( backgroundContainer.x <= -background1.width )
        backgroundContainer.x = 0;
    }
  }
}
```

【PS：上面的代码过一遍就好了，自己执行的话我们又没有它的这些素材，如果你手头上有素材可以代替的话倒是可以试一下玩玩】
现在可以看到，我们的背景在菜单之后不停地滚动，如下图所示：

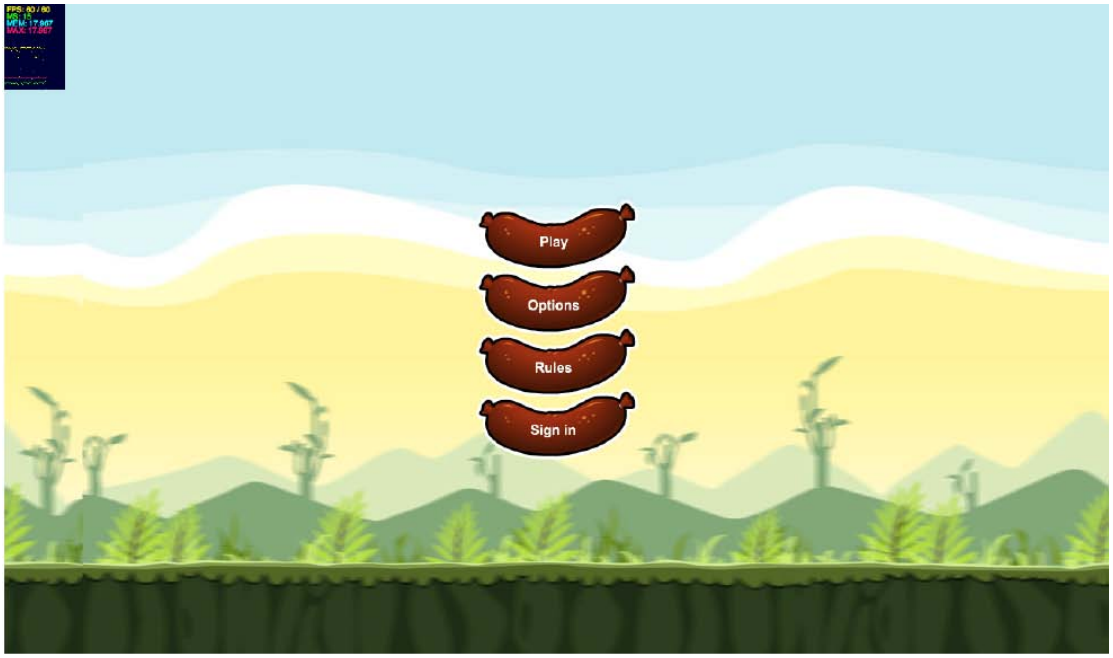

图 1.32
我们的菜单及滚动背景

我们在 Photoshop 中对背景图片做了一点模糊处理，使之能模拟出一种滚动背景的视觉感受。

TextField

在最开头的那个旋转方块的例子中我们已经简单地接触了一下 starling.text.TextField 这个 API，下面让我们再花更多的一点精力来看看 Starling 中文本的显示方式和工作原理吧。你可能会奇怪，GPU 不是只能绘图么，那它怎么渲染文字的呢？其实这不过是个障眼法罢了。在后台，Starling 使用 CPU 创建了一个原生的 Flash TextField 对象，在为其设置完文字及格式后，为其拍了个快照或是截了个屏神马的，之后将此图像传至 GPU 进行渲染。【PS：我估计是用了 BitmapData.draw 方法把他外观绘制下来然后做成 Texture 上传至 GPU】

Staring 只会创建一个原生 TextField 对象作为文字源，该文字源将为多个 starling.text.TextField 对象提供文字纹理。Starling 不会为每个 starling.text.TextField 都创建一个原生 TextField 对象，以避免浪费

在下面的代码中，我们创建了一个 TextField 对象来显示一些用了 Verdana 系统字体的文字：

```
package
{
   import starling.display.Sprite;
   import starling.events.Event;
   import starling.text.TextField;
   public class Game5 extends Sprite
   {
      public function Game5()
```

```
        {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // create the TextField object
            var legend:TextField = new TextField(300, 300, "Here is some text, using an embedded font!",
            "Verdana", 38, 0xFFFFFF);
            // centers the text on stage
            legend.x = stage.stageWidth - legend.width >> 1;
            legend.y = stage.stageHeight - legend.height >> 1;
            // show it
            addChild(legend);
        }
    }
}
```

运行结果如下图所示：

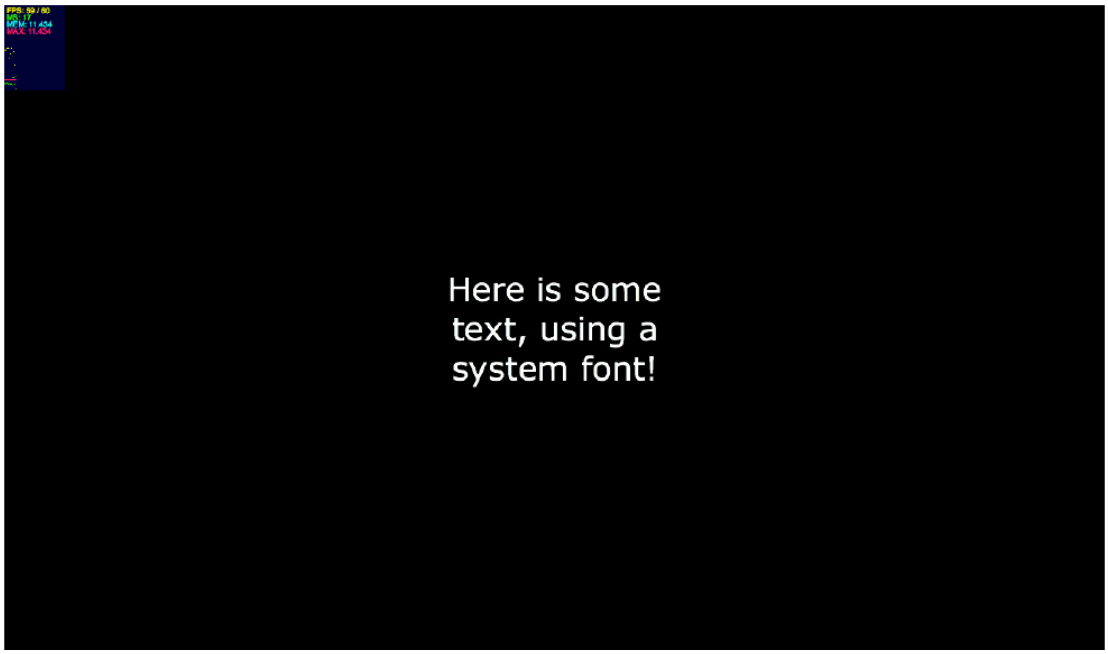

图 1.33
一些简单的文本

再次重申一下，你所看到的并不是真正的 TextField，而是一个 TextField 的快照被绘制在一个 Bitmap 之后封装在 texture 中上传至 GPU 的产物。
让我们进一步地看看 TextField 对象所提供的属性：

* **alpha**：文本透明度
* **autoScale**：是否让文本自动缩放以适应TextField的区域
* **bold**：指定文本是否以粗体显示
* **border**：允许显示textfield的边框。在对textfield进行显式调试时此属性很有用

* **bounds** : textfield所占区域
* **color** : 文本颜色
* **fontName** : 文本字体名
* **fontSize** : 文本大小
* **hAlign** : 文本水平排列方式
* **italic** : 指定文本是否以斜体显示
* **kerning** : 当你在使用位图字体时（前提是存在可用的位图字体），允许你设置字符间距。此属性默认值为 YES.
* **text** : 显示的文本
* **textBounds** : textfield中实际文本所占区域
* **underline** : 指定文本是否显示下划线
* **vAlign** : 文本垂直排列方式

TextField 对象中最 TM 酷的特性就是 autoScale 属性了，稍后我们再来聊一聊它。现在，让我们先来把玩一下其他的几个属性，在下面的代码中，我们让 textfield 显示出它的边框，并把其中文字设置为粗体并改变文字颜色：

```
// create the TextField object
var legend:TextField = new TextField(300, 300, "Here is some text, using an embedded font!", "Verdana",
38,
0xFFFFFF);
// change the color, set bold and enable a border
legend.color = 0x990000;
legend.bold = true;
legend.border = true;
```

运行结果如下图所示：

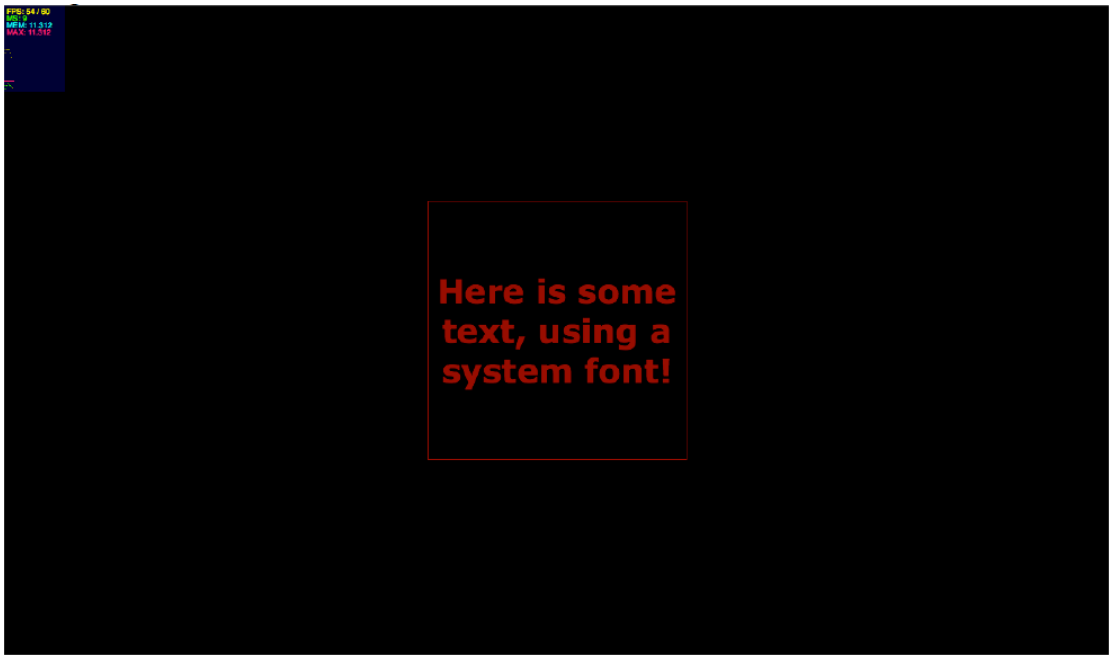

图 1.34

一些简单的，被改变了颜色的文字

需要注意的是，TextField 对象并不支持 HTML 文本。也许在 Starling 的未来版本中我们会推出此特性。如果你很期待此功能的推出的话，千万不要犹豫，赶紧去 Stalring 的论坛上发出你的声音吧少年！记住，Starling 是一个开源的框架，你甚至可以自己实现此功能并以插件的形式分享到 Starling 开发者社区哦。
在之前的例子中我们用了一些非常常见的系统字体，不过现在在我们的项目中我们往往需要用到一些嵌入字体。比如，在一些游戏中，为了给玩家留下较深刻的印象，往往会用上漂亮的嵌入字体。

嵌入字体

Starling 并不像原生 TextField 那样提供一个 embedFonts 属性供我们使用。但是不必担心，事实上在 Starling 中使用嵌入字体会更加地简单。首先要做什么呢？是的，You got it，当然是先嵌入或者从外部加载一个字体文件，之后，实例化该字体类并将其的字体名 fontName 作为参数传至 TextField 构造函数即可。
在下面的代码中，我们首先实例化了一个嵌入的字体资源，之后将该字体对象的 fontName 属性传给 TextField 构造函数以使用之：

```
package
{
   import flash.text.Font;
   import starling.display.Sprite;
   import starling.events.Event;
   import starling.text.TextField;
   public class Game5 extends Sprite
   {
      [Embed(source='/../media/fonts/Abduction.ttf', embedAsCFF='false', fontName='Abduction')]
      public static var Abduction:Class;
      public function Game5()
      {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
      }
      private function onAdded (e:Event):void
      {
         // create the font
         var font:Font = new Abduction();
         // create the TextField object
         var legend:TextField = new TextField(300, 300, "Here is some text, using an embedded font!",
         font.fontName, 38, 0xFFFFFF);
         // centers the text on stage
         legend.x = stage.stageWidth - legend.width >> 1;
         legend.y = stage.stageHeight - legend.height >> 1;
         // show it
```

```
            addChild(legend);
        }
    }
}
```

TextField 对象会根据你传入的字体名去寻找正确的嵌入字体并使用之。下图显示了我们刚才的例子运行结果：

图 1.35
使用了嵌入字体的文字

是不是有一种简单到爆的感觉？有时候，你可能想在你的应用中放一个文本输入框，让用户输入诸如用户名、密码等信息。正如前面所说，既然在 GPU 中渲染的文本都是假象，那么渲染神马文本输入框一定也是不太可能的了。所以，为了在 Stage3D 的场景中使用文本输入框，我们必须玩一些阴招，在用 AS 语言开发手机应用时也一样会用到此阴招。此阴招说难也不难，一说你就懂了，那就是在 Stage3D 场景之上放入原生 Flash 中的显示对象。
在本教程的一开始，我们就讲到了 Flash Player 中三大场景的层次关系，原生 Flash 显示列表采用 CPU 进行渲染，在最上层；Stage3D 显示列表采用 GPU 进行渲染，在原生显示列表之下。这就意味着我们在 Stage3D 的场景中添加的原生 Flash 显示对象都会位于最上层显示，和他位置产生重叠的 Stage3D 内容都会被遮挡。因此，我们只需要创建一个原生 Flash 的 TextField 并设置其 type 为 INPUT 就可以得到一个文本输入框，之后我们使用 Starling 的 nativeOverlay 属性将其添加到原生 Flash 舞台上某个位置即可在屏幕上看到它了。
回忆一下，在一开始我们设置 Stage3D 运行环境的时候，若是 wmode 不正确，Starling 就会在屏幕上显示 wmode 不正确的错误提示。这其实就用到了 native overlay 的特性。【PS：就是在 Stage3D 挂掉的时候用回原生 Flash 显示列表】正如我们所知的那样，Starling 是在 Stage3D 舞台上工作的，不过，通过 native overlay 的特性，你得以在 Starling 对象中访问原生 Flash 显示列表，并可以向其中添加你以前常用的原生 Flash 显示对象，如视频播放器（vedio）、文本输入框（textinput）等，它们将会覆盖于 Stage3D 舞台之上。
当 Starling 检测到网页使用了错误的 wmode 时，会调用其内部的 showFatalError 方法来在

Stage3D 内容之上显示警告信息。很明显的，此时并没有 GPU 可渲染的内容，Starling 依靠的是原生 Flash 显示列表来显示该信息的：

```
private function showFatalError(message:String):void
{
    var textField:TextField = new TextField();
    var textFormat:TextFormat = new TextFormat("Verdana", 12, 0xFFFFFF);
    textFormat.align = TextFormatAlign.CENTER;
    textField.defaultTextFormat = textFormat;
    textField.wordWrap = true;
    textField.width = mStage.stageWidth * 0.75;
    textField.autoSize = TextFieldAutoSize.CENTER;
    textField.text = message;
    textField.x = (mStage.stageWidth - textField.width) / 2;
    textField.y = (mStage.stageHeight - textField.height) / 2;
    textField.background = true;
    textField.backgroundColor = 0x440000;
    nativeOverlay.addChild(textField);
}
```

在下面的代码中，我们创建了一个文本输入框并添加至原生显示列表，它将位于 Starling 所有显示内容之上：

```
var textInput:flash.text.TextField = new flash.text.TextField();
textInput.type = TextFieldType.INPUT;
Starling.current.nativeOverlay.addChild(textInput);
```

当你需要一个文本输入框来为你收集信息的时候，上述特性就会显得非常有用。还有一点值得一提，就是你可以在任何时候通过 Starling 对象的 nativeStage 属性来访问原生 Flash 的舞台对象（flash.display.Stage）：

```
// access the native frame rate from the flash.display.Stage
trace ( Starling.current.nativeStage.frameRate )
```

现在，让我们一起来了解一下位图字体的特性吧，它可能为你在使用 Starling 进行应用开发时带来非常棒的视觉效果的哦。

位图字体

为了获得最佳的视觉体验，降低资源加载量和垃圾回收量，我们可以在 TextField 对象中用上位图字体。位图字体的制作原理很简单，而且我们对其也已经非常熟悉了，就是把所有字体素材都集成在一个 SpriteSheet 中，之后我们每个字的外观都将从该 SpriteSheet 中采样。下图显示的是一个叫做GlyphDesigner的工具（71 squared出品，收费软件）截图，它是专门为制作位图字体的SpriteSheet而诞生的，下图中它正在制作一个叫做Britannic Bold的字体：

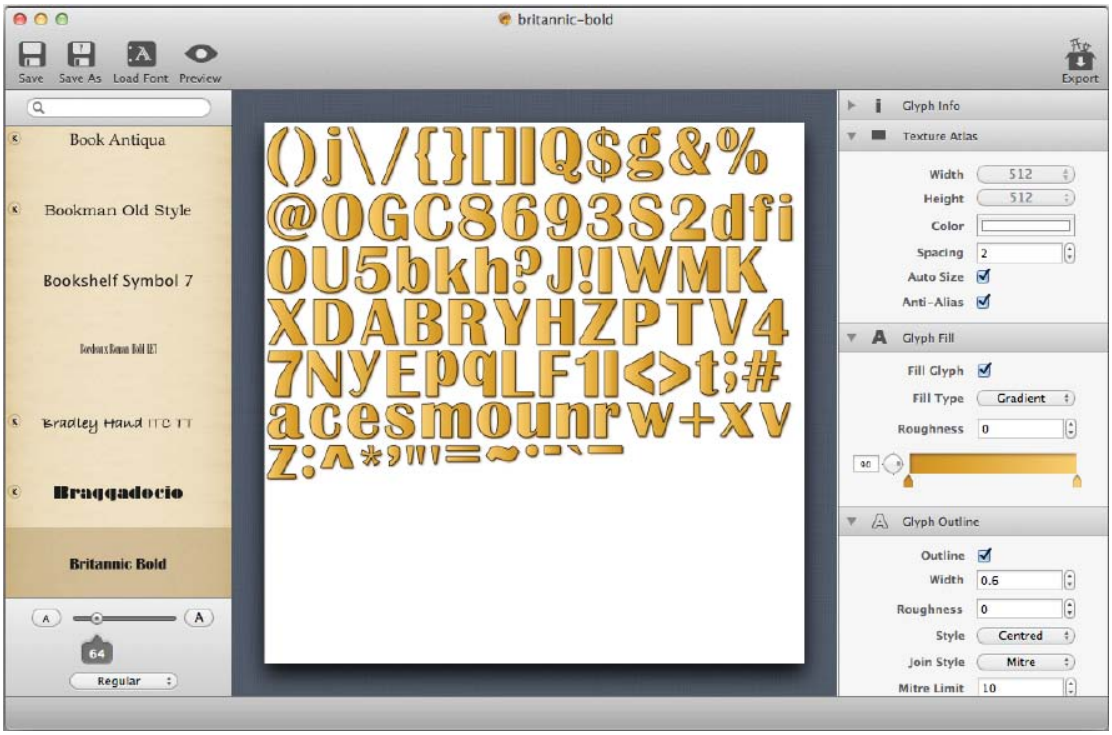

图 1.36

MacOS 系统中的 GlyphDesigner 工具

该软件只能运行于 MacOS 上，若是你使用的是 Windows 系统，那么有一款类似功能的软件叫做 Bitmap Font Generator（Angel Code 出品，免费软件）可以一用：

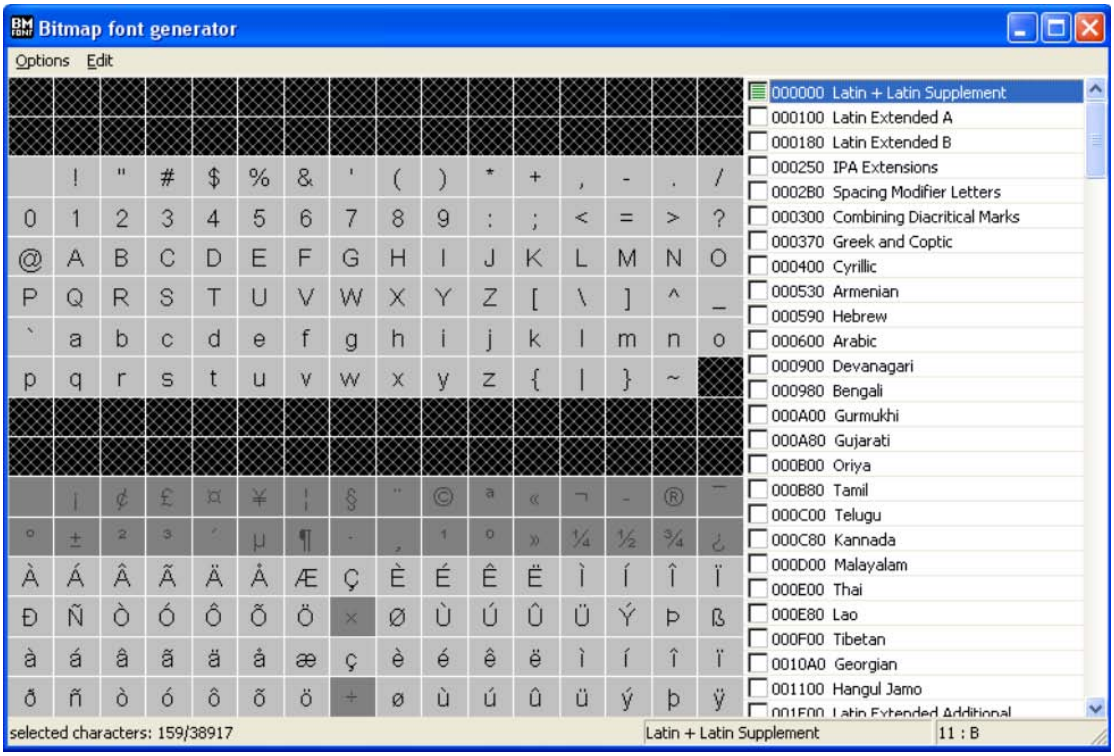

图 1.37

运行于 Windows 平台的 Bitmap Font Generator 工具

是的，相信我，对比这两者，你会更想用 GlyphDesigner， ^_^。
当然，如果你想在运行时生成字体 SpriteSheet 的话也没人会阻止你，生成步骤也很简单，这里只是简单提一下，首先先用 TextField 将所需字体包含进来，之后设置其 text 属性为你要包含进位图字体的文字，并保证不重复（如 0-9，A-Z），最后使用 BitmapData 的 draw 方法将此 TextField 绘制下来上传至 GPU 即可。对于位图字体，是使用运行时生成还是使用嵌入资源，决定权在于你，当然，应用的运行平台（是桌面应用还是手机应用）会很大程度影响你的选择。使用运行时生成的方式可以为你节省空间，而使用嵌入资源能让你的应用启动速度加快，到底用哪种方式，你来决定！
当使用位图字体制作工具导出位图字体的时候，纹理素材都会被保存在一张图片中，而对于该 Spritesheet 的描述信息（每个字所在位置，尺寸等信息）则会被保存于一个.fnt(XML 或文本)的描述文件中，它的格式类似于：

```
<font>
  <info face="BranchingMouse" size="40" />
  <common lineHeight="40" />
  <pages> <!-- currently, only one page is supported -->
    <page id="0" file="texture.png" />
  </pages>
  <chars>
    <char id="32" x="60" y="29" width="1" height="1" xoffset="0" yoffset="27" xadvance="8" />
    <char id="33" x="155" y="144" width="9" height="21" xoffset="0" yoffset="6" xadvance="9" />
  </chars>
  <kernings> <!-- Kerning is optional -->
    <kerning first="83" second="83" amount="-4"/>
  </kernings>
</font>
```

一旦我们导出了位图字体的素材和描述文件，就可以像往常一样在项目中嵌入它们以供使用了：

```
[Embed(source = "../media/fonts/britannic-bold.png")]
private static const BitmapChars:Class;
[Embed(source="../media/fonts/britannic-bold.fnt", mimeType="application/octet-stream")]
private static const BritannicXML:Class;
```

为了使用它们，我们需要用到以下 TextField 的 API：

* **registerBitmapFont**: 注册一个位图字体
* **unregisterBitmapFont**: 注销一个位图字体

我们需要把字体纹理和描述信息传给 BitmapFont 对象的构造函数，然后使用 registerBitmapFont 来注册这个位图字体：

```
package
{
    import flash.display.Bitmap;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.text.BitmapFont;
    import starling.text.TextField;
    import starling.textures.Texture;
    import starling.utils.Color;
    public class Game5 extends Sprite
    {
        [Embed(source = "../media/fonts/britannic-bold.png")]
        private static const BitmapChars:Class;
        [Embed(source="../media/fonts/britannic-bold.fnt", mimeType="application/octet-stream")]
        private static const BritannicXML:Class;
        public function Game5()
        {
                addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // creates the embedded bitmap (spritesheet file)
            var bitmap:Bitmap = new BitmapChars();
            // creates a texture out of it
            var texture:Texture = Texture.fromBitmap(bitmap);
            // create the XML file describing the glyphes position on the spritesheet
            var xml:XML = XML(new BritannicXML());
            // register the bitmap font to make it available to TextField
            TextField.registerBitmapFont(new BitmapFont(texture, xml));
            // create the TextField object
            var bmpFontTF:TextField = new TextField(400, 400, "Here is some text, using an
            embedded font!",
            "BritannicBold", 10);
            // the native bitmap font size, no scaling
            bmpFontTF.fontSize = BitmapFont.NATIVE_SIZE;
            // use white to use the texture as it is (no tinting)
            bmpFontTF.color = Color.WHITE;
            // centers the text on stage
            bmpFontTF.x = stage.stageWidth - bmpFontTF.width >> 1;
            bmpFontTF.y = stage.stageHeight - bmpFontTF.height >> 1;
            // show it
            addChild(bmpFontTF);
        }
    }
```

```
}
```

一旦注册了位图字体，我们就可以用该字体的字体名来创建 TextField 对象并使用了。
以上代码的运行结果如下图所示：

图 1.38
一些使用了位图字体的文本

现在，让我们把要显示的文本增加一些，看看会发生什么事：

```
var bmpFontTF:TextField = new TextField(400, 400, "Here is some longer text that is very likely to be cut, using an embedded font!", "BritannicBold", 10);
```

不出所料，由于我们 TextField 对象的尺寸太小，不能显示完全全部的文字：

图 1.39
一些长的位图文字不能完全显示

幸好，Starling 为 TextField 对象提供了一个 autoScale 属性：

```
// make the text fit into the box
bmpFontTF.autoScale = true;
```

当你在游戏的开发过程中，需要把文本集中显示于一个指定的区域内时，使用此特性会非常有用，它可以让你确保文字不会显示不完整。在很多时候，出于设计的目的，使用此属性可以保证你文字排版的一致性，即不论文字长短，都可以保证文字能够整齐地排列并占满整个文本区域（有时候会对文字大小产生一定的缩放）。
下图为我们启用 autoScale 属性后文本产生的细微变化：

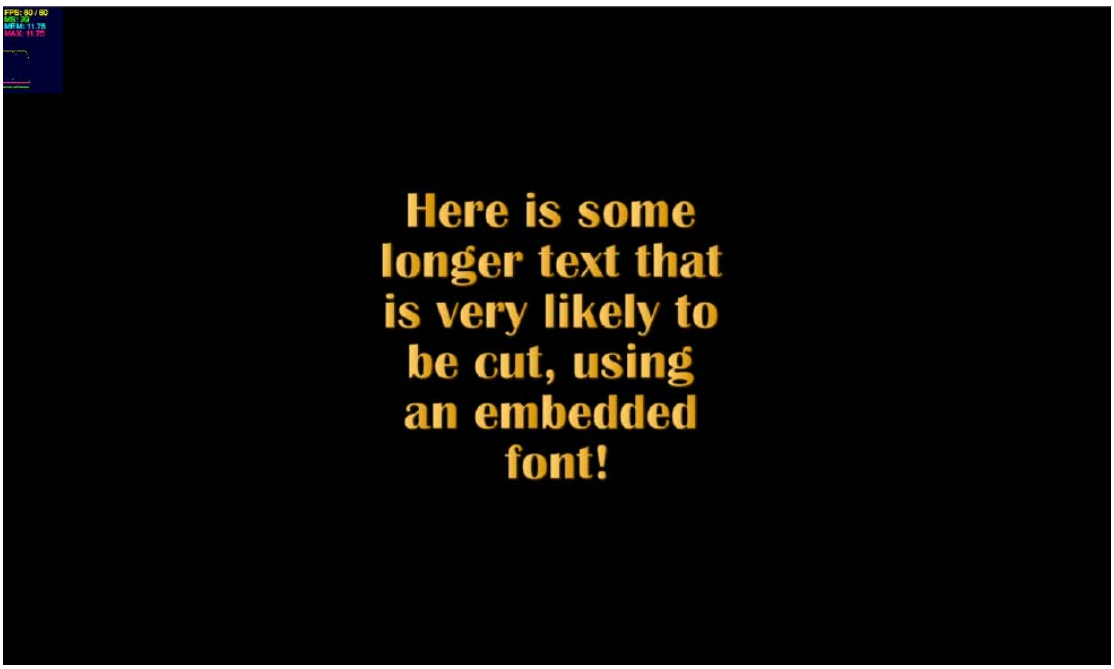

图 1.40

超长文本段落被缩放到了适合的尺寸

在之前那个按钮组成的菜单的例子基础上，我们用上位图字体来强化其表现力：

```
package
{
    import flash.display.Bitmap;
    import flash.geom.Rectangle;
    import starling.display.Button;
    import starling.display.Image;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.text.BitmapFont;
    import starling.text.TextField;
    import starling.textures.Texture;
    public class Game4 extends Sprite
    {
        [Embed(source = "../media/textures/sausage-skin.png")]
        private static const ButtonTexture:Class;
        [Embed(source = "../media/textures/background.jpg")]
        private static const BackgroundImage:Class;
        [Embed(source = "../media/fonts/hobo-std.png")]
        private static const BitmapChars:Class;
        [Embed(source="../media/fonts/hobo-std.fnt", mimeType="application/octet-stream")]
        private static const Hobo:Class;
        private static const FONT_NAME:String = "HoboStd";
        private var backgroundContainer:Sprite;
```

```
private var background1:Image;
private var background2:Image;
// sections
private var sections:Vector.<String> = Vector.<String>(["Play", "Options", "Rules", "Sign in"]);
public function Game4()
{
        addEventListener(Event.ADDED_TO_STAGE, onAdded);
}
private function onAdded (e:Event):void
{
    // creates the embedded bitmap (spritesheet file)
    var bitmap:Bitmap = new BitmapChars();
    // creates a texture out of it
    var texture:Texture = Texture.fromBitmap(bitmap);
    // create the XML file describing the glyphes position on the spritesheet
    var xml:XML = XML(new Hobo());
    // register the bitmap font to make it available to TextField
    TextField.registerBitmapFont(new BitmapFont(texture, xml));
    // create a Bitmap object out of the embedded image
    var buttonSkin:Bitmap = new ButtonTexture();
    // create a Texture object to feed the Button object
    var textureSkin:Texture = Texture.fromBitmap(buttonSkin);
    // create a Bitmap object out of the embedded image
    var background:Bitmap = new BackgroundImage();
    // create a Texture object to feed the Image object
    var textureBackground:Texture = Texture.fromBitmap(background);
    // container for the background textures
    backgroundContainer = new Sprite();
    // create the images for the background
    background1 = new Image(textureBackground);
    background2 = new Image(textureBackground);
    // positions the second part
    background2.x = background1.width;
    // nest them
    backgroundContainer.addChild(background1);
    backgroundContainer.addChild(background2);
    // show the background
    addChild(backgroundContainer);
    // create container for the menu (buttons)
    var menuContainer:Sprite = new Sprite();
    var numSections:uint = sections.length
    for (var i:uint = 0; i< numSections; i++)
    {
        // create a button using this skin as up state
```

```
            var myButton:Button = new Button(textureSkin, sections[i]);
            // font name
            myButton.fontName = FONT_NAME;
            myButton.fontColor = 0xFFFFFF;
            // positions the text
            myButton.textBounds = new Rectangle(10, 38, 160, 30);
            // font size
            myButton.fontSize = 26;
            // position the buttons
            myButton.y = (myButton.height-10) * i;
            // add the button to our container
            menuContainer.addChild(myButton);
        }
        // catch the Event.TRIGGERED event
        menuContainer.addEventListener(Event.TRIGGERED, onTriggered);
        // on each frame
        stage.addEventListener(Event.ENTER_FRAME, onFrame);
        // centers the menu
        menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
        menuContainer.y = stage.stageHeight - menuContainer.height >> 1;
        // show the button
        addChild(menuContainer);
    }
    private function onTriggered(e:Event):void
    {
        // outputs : [object Sprite] [object Button]
        trace ( e.currentTarget, e.target );
        // outputs : triggered!
        trace ("triggered!");
    }
    private function onFrame (e:Event):void
    {
        // scroll it
        backgroundContainer.x -= 10;
        // reset
        if ( backgroundContainer.x <= -background1.width )
            backgroundContainer.x = 0;
    }
  }
}
```

运行结果如下图所示：

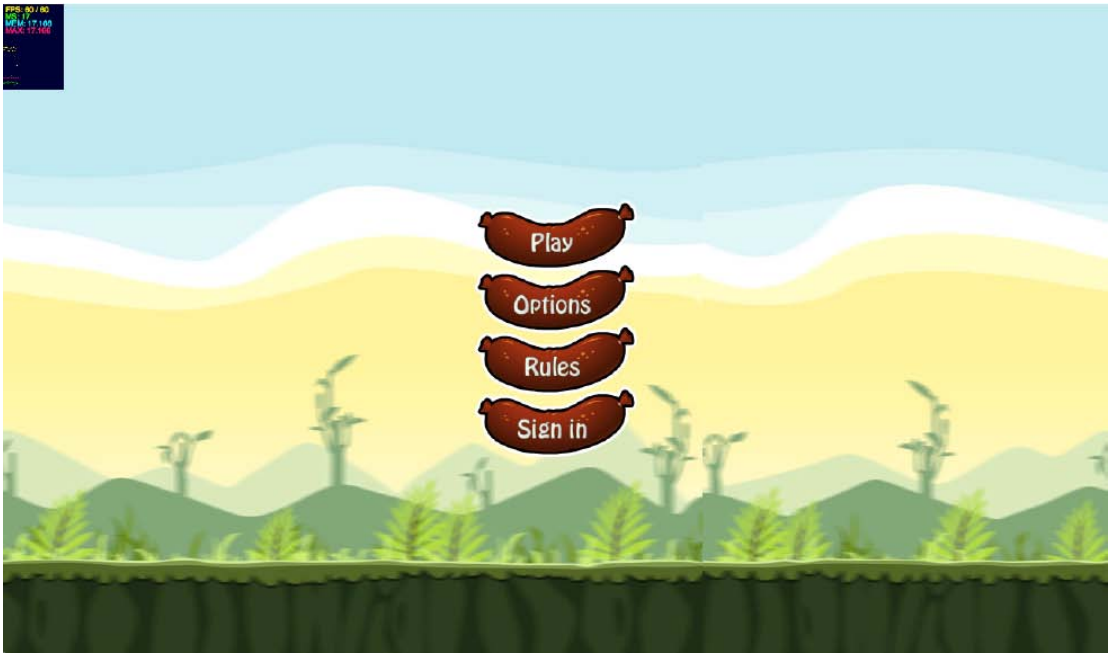

图 1.41
用上了位图字体的菜单

接下来让我们再来看另一个 Starling 中比较好用的特性，叫做 rendered textures。

RenderTexture

starling.textures.RenderTexture API 允许开发者在 Starling 中实现无损绘画功能。如果你是一名 Flash 开发人员，那么你可以把该 API 想成是原生 Flash 中的 BitmapData 对象。当你在创建类似绘图工具的应用程序时该特性会十分有用，RenderTexture 对象可以作为一张“画布”，满足你在一个 Texture 纹理对象中进行绘画的愿望，而且每次绘画时不会抹除上一次绘画的结果。
在下面的代码中，我们使用 Starling 框架在 GPU 中重现了原生 Flash 中类似 BitmapData.draw 方法提供的功能：

```
package
{
    import flash.display.Bitmap;
    import flash.geom.Point;
    import starling.display.Image;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.events.Touch;
    import starling.events.TouchEvent;
    import starling.events.TouchPhase;
    import starling.textures.RenderTexture;
    import starling.textures.Texture;
    public class Game10 extends Sprite
```

```
{
    private var mRenderTexture:RenderTexture;
    private var mBrush:Image;
    [Embed(source = "/../media/textures/egg_closed.png")]
    private static const Egg:Class;
    public function Game10()
    {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
    }
    private function onAdded (e:Event):void
    {
        // create a Bitmap object out of the embedded image
        var brush:Bitmap = new Egg();
        // create a Texture object to feed the Image object
        var texture:Texture = Texture.fromBitmap(brush);
        // create the texture to draw into the texture
        mBrush = new Image(texture);
        // set the registration point
        mBrush.pivotX = mBrush.width >> 1;
        mBrush.pivotY = mBrush.height >> 1;
        // scale it
        mBrush.scaleX = mBrush.scaleY = 0.5;
        // creates the canvas to draw into
        mRenderTexture = new RenderTexture(stage.stageWidth, stage.stageHeight);
        // we encapsulate it into an Image object
        var canvas:Image = new Image(mRenderTexture);
        // show it
        addChild(canvas);
        // listen to mouse interactions on the stage
        stage.addEventListener(TouchEvent.TOUCH, onTouch);
    }
    private function onTouch(event:TouchEvent):void
    {
        // retrieves the entire set of touch points (in case of multiple fingers on a touch screen)
        var touches:Vector.<Touch> = event.getTouches(this);
        for each (var touch:Touch in touches)
        {
            // if only hovering or click states, let's skip
            if (touch.phase == TouchPhase.HOVER || touch.phase == TouchPhase.ENDED)
                    continue;
            // grab the location of the mouse or each finger
            var location:Point = touch.getLocation(this);
            // positions the brush to draw
            mBrush.x = location.x;
```

```
                mBrush.y = location.y;
                // draw into the canvas
                mRenderTexture.draw(mBrush);
            }
        }
    }
}
```

当按住鼠标键并移动鼠标时，会不断地绘制我们提供的绘制对象：

图 1.42
无损绘画

接下来我们要讨论的是一个对于 Flash 开发者来说知名度很高的话题：补间动画（Tween）。

Tweens

Starling 天生支持补间动画并实现了一套其特有的补间动画引擎，其支持绝大多数的缓动方程，如下图所示：

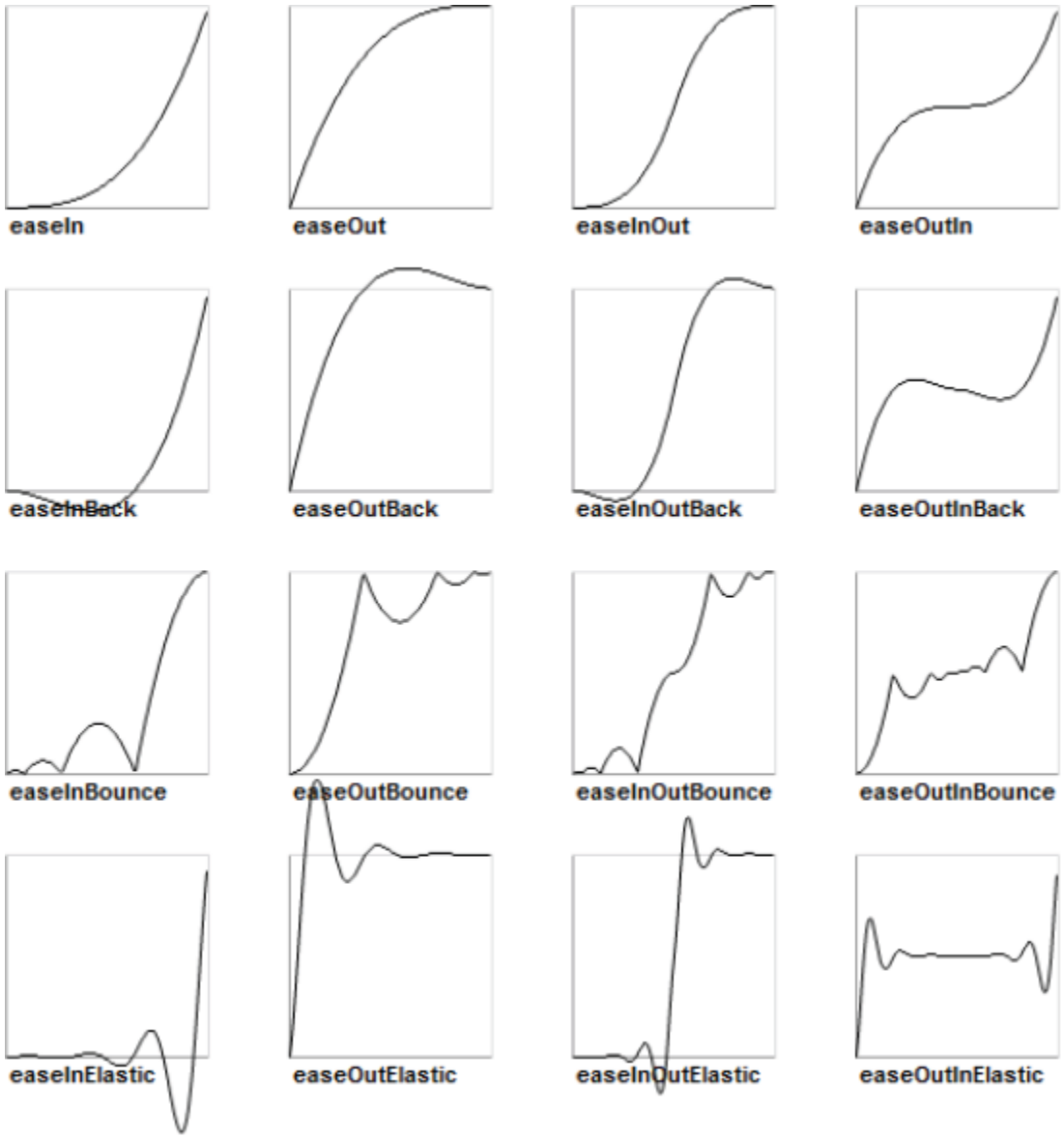

图 1.43

Starling 中支持的缓动方程（插图来源于 sparrow-framework.org）

在下面的代码中，我们使用补间动画来缓动改变一个 TextField 对象的 x，y 属性，并用上一个 bounce 的缓动特效：

```
// create a Tween object
var t:Tween = new Tween(bmpFontTF, 4, Transitions.EASE_IN_OUT_BOUNCE);
// move the object position
t.moveTo(bmpFontTF.x+300, bmpFontTF.y);
// add it to the Juggler
Starling.juggler.add(t);
```

下面列出 Tween 对象中可用的 API：

* **animate**：缓动一个对象的某个属性至指定目标值. 你可以在一个Tween对象中多次

调用此方法
* **currentTime**：补间动画当前播放到的时间
* **delay**：补间动画开始前需要等待的延迟时间
* **fadeTo**：缓动一个对象的透明度至指定目标值．你可以在一个Tween对象中多次调用此方法
* **isComplete**：此标记用以判断一个补间动画是否播放完毕
* **moveTo**：同时缓动一个对象的x、y属性至指定目标值
* **onComplete**：补间动画播放完毕回调函数
* **onCompleteArgs**：补间动画播放完毕回调函数参数
* **onStart**：补间动画开始播放回调函数
* **onStartArgs**：补间动画开始播放回调函数参数
* **onUpdate**：补间动画每帧更新时都会调用此方法
* **onUpdateArgs**：需要传入补间动画每帧更新时都会调用的方法的参数
* **roundToInt**:若该标记为true，则所有带小数的属性值都会去掉小数变为整数
* **scaleTo**：同时将scaleX及scaleY属性值缓动至指定目标值.
* **target**：缓动目标对象
* **totalTime**：补间动画所要消耗的总时间（单位为秒）.
* **transition**：指定补间动画所用缓动方程

下面给出一个使用示例，在该例子中，某对象在完成缓动后被完全销毁：

```
// create a Tween object
var t:Tween = new Tween(bmpFontTF, 4, Transitions.EASE_IN_OUT_BOUNCE);
// move the object position
t.moveTo(bmpFontTF.x+300, bmpFontTF.y);
t.animate("alpha", 0);
// add it to the Juggler
Starling.juggler.add(t);
// on complete, remove the textfield from the stage
t.onComplete = bmpFontTF.removeFromParent;
// pass the dispose argument to the removeFromParent call
t.onCompleteArgs = [true];
```

在下面的代码中，我们通过为onStart、onUpdate及onComplete设置回调函数来侦听一个补间动画的各个阶段：

```
package
{
    import flash.text.Font;
    import starling.animation.Transitions;
    import starling.animation.Tween;
    import starling.core.Starling;
    import starling.display.Sprite;
    import starling.events.Event;
```

```
import starling.text.TextField;
public class Game5 extends Sprite
{
    [Embed(source='/../media/fonts/Abduction.ttf', embedAsCFF='false', fontName='Abduction')]
    public static var Abduction:Class;
    public function Game5()
    {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
    }
    private function onAdded (e:Event):void
    {
        // create the font
        var font:Font = new Abduction();
        // create the TextField object
        var legend:TextField = new TextField(300, 300, "Here is some text, using an embedded font!",
        font.fontName, 38, 0xFFFFFF);
        // centers the text on stage
        legend.x = stage.stageWidth - legend.width >> 1;
        legend.y = stage.stageHeight - legend.height >> 1;
        // create a Tween object
        var t:Tween = new Tween(legend, 4, Transitions.EASE_IN_OUT_BOUNCE);
        // move the object position
        t.moveTo(legend.x+300, legend.y);
        // add it to the Juggler
        Starling.juggler.add(t);
        // listen to the start
        t.onStart = onStart;
        // listen to the progress
        t.onUpdate = onProgress;
        // listen to the end
        t.onComplete = onComplete;
        // show it
        addChild(legend);
    }
    private function onStart():void
    {
            trace ("tweening complete");
    }
    private function onProgress():void
    {
            trace ("tweening in progress");
    }
    private function onComplete():void
    {
```

```
            trace ("tweening complete")
        }
    }
}
```

接下来，让我们看看如果更加系统地来管理我们项目中的素材。到目前为止，我们一直在用简单的嵌入元标签来帮我们实现所需资源的嵌入工作。但是在一个较大的项目中，你需要将资源进行集中管理。在接下来的章节中，我们将一起来学习几个简单的资源整合、重用技术。

资源管理器

到目前为止，我们都是以一种相当简单的方式来使用资源的。为了优化我们使用资源的方式，我们建议你创建一个类（假设其名字为 Assets）来作为唯一的资源供应源。这个 Assets 对象负责对资源进行缓存以便资源能够得以被重复使用，这样可以避免重复创建冗余资源对象，减轻垃圾回收的负担。
在下面的代码中，我们为 Assets 类设计了一个 getTexture 方法以供我们检索一个嵌入资源，若是嵌入资源已实例化过，则重用之，否则实例化它：

```
public static function getTexture(name:String):Texture
{
    if (Assets[name] != undefined)
    {
        if (sTextures[name] == undefined)
        {
            var bitmap:Bitmap = new Assets[name]();
            sTextures[name] = Texture.fromBitmap(bitmap);
        }
        return sTextures[name];
    } else throw new Error("Resource not defined.");
}
```

我们使用了一个简单的 Dictionary 对象来存储已实例化的资源，【PS：我们可以称该对象为对象池】下一次我们要用到相同资源时就可以从中取出相应实例进行重用，这样就可以避免重复实例化造成的资源浪费。
像往常一样，我们还是用到 Embed 元标签来嵌入资源：

```
[Embed(source = "../media/textures/background.png")]
private static const Background:Class;
```

Starling 并没有要求你必须使用嵌入的资源，你可以把资源放在外部，在运行时使用 Loader 对象来进行资源的动态加载：

```
// create the Loader
var loader:Loader = new Loader();
```

```
// listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );
// load the image
loader.load ( new URLRequest ("texture.png" ) );
function onComplete ( e:Event ):void
{
    // create the Bitmap
    var bitmap:Bitmap = e.currentTarget.data;
    // creates a texture out of it
    var texture:Texture = Texture.fromBitmap(bitmap);
    // create the Image object
    var image:Image = new Image (texture);
    // show the image
    addChild (image);
}
```

到目前为止，我们用的纹理都是取自于图片或是一些预定义好的 Bitmap 类，那么如果我们想动态地创建一些纹理来使用怎么办呢？
此时我们可以使用 Texture 类的静态方法 fromBitmapData：

```
// creates a dynamic bitmap
var dynamicBitmap:BitmapData = new BitmapData (512, 512);
// we draw a custom vector shape coming from the library or drawn at runtime too
dynamicBitmap.draw ( myCustomShape );
// creates a texture out of it
var texture:Texture = Texture.fromBitmapData(bitmap);
// create the Image object
var image:Image = new Image (texture);
// show the image
addChild (image);
```

你的 Starling 应用程序可能会在移动设备或是桌面浏览器上面运行，这就要面临一个问题，就是不同的屏幕尺寸可能会影响你应用程序的布局。不过 Starling 允许你很容易地就能够处理屏幕尺寸的改变带来的问题，在接下来的章节中让我们一起来见识一下。

处理屏幕尺寸改变

作为一名Flash开发者，我们在过去可以依赖一个非常简单的事件Event.RESIZE来在屏幕/页面尺寸改变时进行处理。还可以依赖stage.stageWidth及stage.stageHeight属性来布置对象的位置，不论屏幕/页面尺寸怎样。
为了实现类似功能，Starling 中的舞台对象 starling.display.Stage 将会在屏幕/页面尺寸改变时抛出一个类似的事件，叫做 ResizeEvent.RESIZE，允许你动态地处理屏幕改变时的逻辑。
在下面的代码中，我们在此教程首个例子的基础上，增加屏幕尺寸改变时更新方块位置的功

能：

```
package
{
    import flash.geom.Rectangle;
    import starling.core.Starling;
    import starling.display.Quad;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.events.ResizeEvent;
    public class Game extends Sprite
    {
        private var q:Quad;
        private var rect:Rectangle = new Rectangle(0,0,0,0);
        public function Game()
        {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded ( e:Event ):void
        {
            // listen to the event
            stage.addEventListener(ResizeEvent.RESIZE, onResize);
            q = new Quad(200, 200);
            q.color = 0x00FF00;
            q.x = stage.stageWidth - q.width >> 1;
            q.y = stage.stageHeight - q.height >> 1;
            addChild ( q );
        }
        private function onResize(event:ResizeEvent):void
        {
            // set rect dimmensions
            rect.width = event.width, rect.height = event.height;
            // resize the viewport
            Starling.current.viewPort = rect;
            // assign the new stage width and height
            stage.stageWidth = event.width;
            stage.stageHeight = event.height;
            // repositions our quad
            q.x = stage.stageWidth - q.width >> 1;
            q.y = stage.stageHeight - q.height >> 1;
        }
    }
}
```

一旦我们的 SWF 尺寸发生了改变，ResizeEvent.RESIZE 事件都会被派发。新的尺寸将会被保存于 ResizeEvent 事件对象的相应属性中，我们手动地将这些新尺寸赋值给 stageWidth 及 stageHeight 属性。之后我们就可以重新定位舞台中的内容了，所有位置依赖于舞台尺寸的内容都可以保持相对正确的位置。

在 Starling 中使用 Box2D 作为插件

Starling 的显示列表 API 允许我们能够非常容易地将其与别的一些框架集成起来使用。比如，我们想在我们的游戏中使用 Box2D 来提供一些物理效果的支持。
你可以在 http://box2dflash.sourceforge.net/对Box2D做进一步的了解并下载其类库，我们将在下面的例子中用到它。
在下面的代码中，我们创建了一些盒子并让他们在重力的作用下摔倒地面上。当然，一切的渲染工作都由 GPU 来完成：

```
package
{
   import Box2D.Collision.Shapes.b2CircleShape;
   import Box2D.Collision.Shapes.b2PolygonShape;
   import Box2D.Common.Math.b2Vec2;
   import Box2D.Dynamics.b2Body;
   import Box2D.Dynamics.b2BodyDef;
   import Box2D.Dynamics.b2FixtureDef;
   import Box2D.Dynamics.b2World;
   import starling.display.DisplayObject;
   import starling.display.Quad;
   import starling.display.Sprite;
   import starling.events.Event;
   public class PhysicsTest extends Sprite
   {
      private var mMainMenu:Sprite;
      private var bodyDef:b2BodyDef;
      private var inc:int;
      public var m_world:b2World;
      public var m_velocityIterations:int = 10;
      public var m_positionIterations:int = 10;
      public var m_timeStep:Number = 1.0/30.0;
      public function PhysicsTest()
      {
            addEventListener(Event.ADDED_TO_STAGE, onAdded);
      }
      private function onAdded (e:Event):void
      {
         // Define the gravity vector
         var gravity:b2Vec2 = new b2Vec2(0.0, 10.0);
```

```
// Allow bodies to sleep
var doSleep:Boolean = true;
// Construct a world object
m_world = new b2World( gravity, doSleep);
// Vars used to create bodies
var body:b2Body;
var boxShape:b2PolygonShape;
var circleShape:b2CircleShape;
// Add ground body
bodyDef = new b2BodyDef();
//bodyDef.position.Set(15, 19);
bodyDef.position.Set(10, 28);
//bodyDef.angle = 0.1;
boxShape = new b2PolygonShape();
boxShape.SetAsBox(30, 3);
var fixtureDef:b2FixtureDef = new b2FixtureDef();
fixtureDef.shape = boxShape;
fixtureDef.friction = 0.3;
// static bodies require zero density
fixtureDef.density = 0;
// Add sprite to body userData
var box:Quad = new Quad(2000, 200, 0xCCCCCC);
box.pivotX = box.width / 2.0;
box.pivotY = box.height / 2.0;
bodyDef.userData = box;
bodyDef.userData.width = 34 * 2 * 30;
bodyDef.userData.height = 30 * 2 * 3;
addChild(bodyDef.userData);
body = m_world.CreateBody(bodyDef);
body.CreateFixture(fixtureDef);
var quad:Quad;
// Add some objects
for (var i:int = 1; i < 100; i++)
{
    bodyDef = new b2BodyDef();
    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = Math.random() * 15 + 5;
    bodyDef.position.y = Math.random() * 10;
    var rX:Number = Math.random() + 0.5;
    var rY:Number = Math.random() + 0.5;
    // Box
    boxShape = new b2PolygonShape();
    boxShape.SetAsBox(rX, rY);
    fixtureDef.shape = boxShape;
```

```
            fixtureDef.density = 1.0;
            fixtureDef.friction = 0.5;
            fixtureDef.restitution = 0.2;
            // create the quads
            quad = new Quad(100, 100, Math.random()*0xFFFFFF);
            quad.pivotX = quad.width / 2.0;
            quad.pivotY = quad.height / 2.0;
            // this is the key line, we pass as a userData the starling.display.Quad
            bodyDef.userData = quad;
            bodyDef.userData.width = rX * 2 * 30;
            bodyDef.userData.height = rY * 2 * 30;
            body = m_world.CreateBody(bodyDef);
            body.CreateFixture(fixtureDef);
            // show each quad (acting as a skin of each body)
            addChild(bodyDef.userData);
        }
        // on each frame
        addEventListener(Event.ENTER_FRAME, Update);
    }
    public function Update(e:Event):void
    {
        // we make the world run
        m_world.Step(m_timeStep, m_velocityIterations, m_positionIterations);
        m_world.ClearForces() ;
        // Go through body list and update sprite positions/rotations
        for (var bb:b2Body = m_world.GetBodyList(); bb; bb = bb.GetNext()){
            // key line here, we test if we find any starling.display.DisplayObject objects and apply
            the
            physics to them
            if (bb.GetUserData() is DisplayObject)
            {
                // we cast as a Starling DisplayObject, not the native one !
                var sprite:DisplayObject = bb.GetUserData() as DisplayObject;
                sprite.x = bb.GetPosition().x * 30;
                sprite.y = bb.GetPosition().y * 30;
                sprite.rotation = bb.GetAngle();
            }
        }
        bodyDef.position.Set(10, 28);
    }
  }
}
```

测试以上代码，将会得到类似如下图的结果：

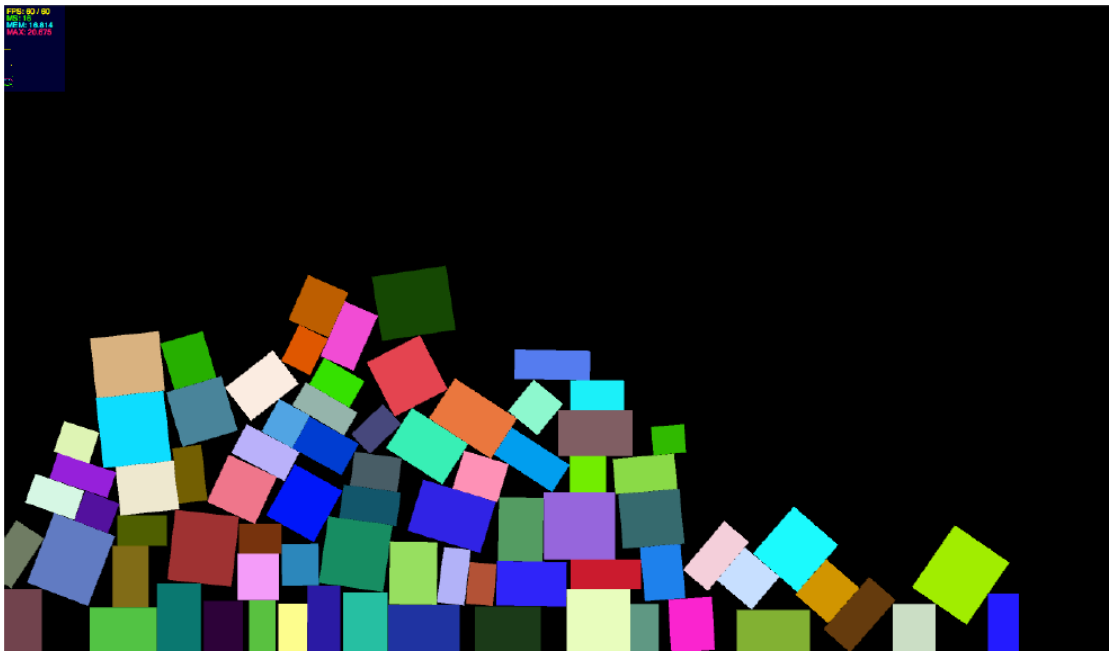

图 1.44
在 Starling 中使用 BOX2D

而且该例子跑在下一个版本的 AIR 运行时（该版本 AIR 支持 Stage3D）运行的结果也是一样的，效率没有差多少：

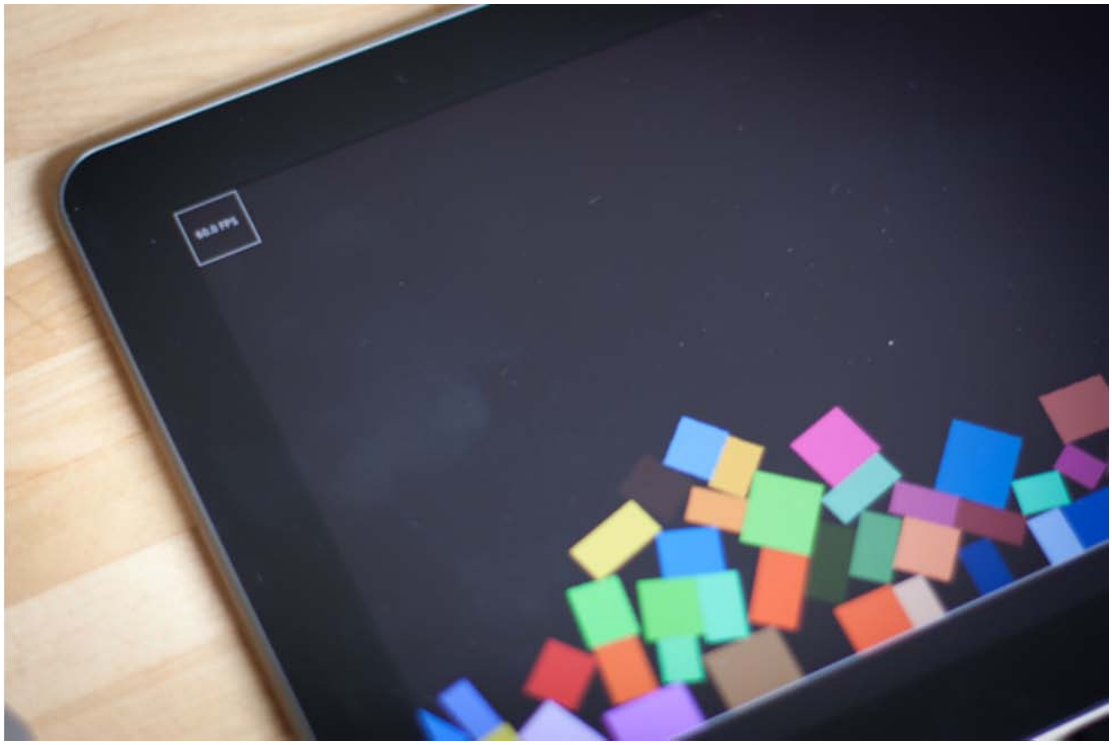

图 1.45
同样的一个应用程序在平板电脑上运行也能保持 60FPS 的帧频

当然，我们可以很便捷地把这些方块换个外形让我们的游戏变得更加精彩。鉴于此，我们可以使用这些插件/框架来创建出超棒的 GPU 加速的 2D 应用程序。

在 Starling 中进行概要分析

当谈及到对表现性能做分析，那么首先浮现于脑海中的自然就是对帧率的监测。一般来说，开发人员都会在调试程序的时候全程保持着调试控制台处于可视位置，然后一边看着控制台中的帧频变化，一边还要看这应用程序的运行效率变化，这样会很痛苦不是吗？我们建议直接在你的应用程序中放入一个小小的调试窗口。

就像我们从本案例一开始就在使用的实时概要分析类Stats，该类由mr.doob出品，可以在 https://github.com/mrdoob/Hi---ReS---Stats下载得到。

我们可以一直用该类来进行概要分析，正如我们之前所做的那样。不过，这也有一个缺陷，尤其是在手机上运行时。在未来版本的 AIR（手机版）中也可以用上 Stage3D 了。那时候，当你在手机设备上，比如 Android 系统中运行时，如果你把原生 Flash 显示列表和 Stage3D 一起用，那么你就在拿你的呈现性能开玩笑。在一些 GPU 上运行时，同时使用原生显示列表和 Stage3D，会让 Flash Player 分别为这两种显示列表分配帧频，因此，当你设置了 60FPS 的帧频后运行，会出现的结果是，原生显示列表分到 30FPS，而 Stage3D 分到 30FPS 这样。因此，即使是出于调试的目的，你也最好不要往原生显示列表中添加东西，保证全部内容都添加在 stage3D 里面。

咱们刚才介绍过了位图字体的概念，那么我们是否能够自己动手来开发一个小小的 FPS 类来实时地在 stage3D 中显示帧频呢？

下图显示了我们在该类中将要用到的位图字体的 Spritesheet，我们只筛选出将会用到的一些文字以保证 Spritesheet 尽可能地简洁：

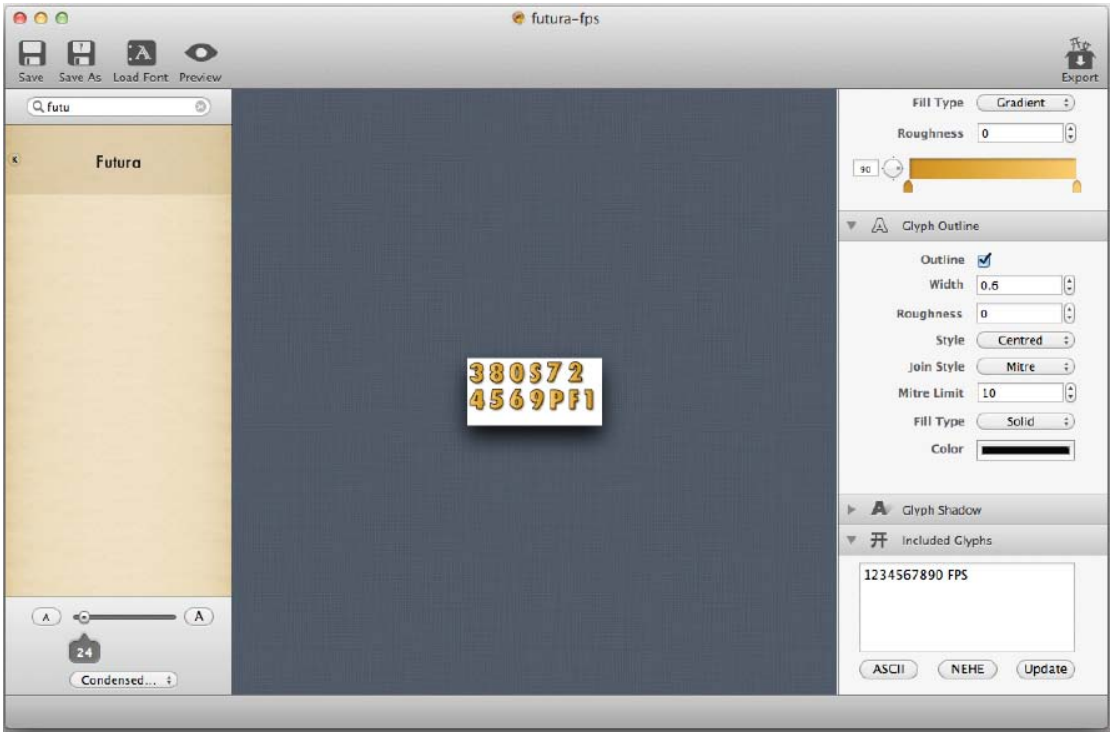

图 1.46

为 FPS 计数器制作的文字材质

一旦导出了 Spritesheet 及其相应的描述文件，我们就可以使用 TextField 结合 BitmapFont 来显示出 FPS 类了：

```
package
{
    import flash.display.Bitmap;
    import flash.utils.getTimer;
    import starling.display.Sprite;
    import starling.events.Event;
    import starling.text.BitmapFont;
    import starling.text.TextField;
    import starling.textures.Texture;
    import starling.utils.Color;
    public class FPS extends Sprite
    {
        private var container:Sprite = new Sprite();
        private var bmpFontTF:TextField;
        private var frameCount:uint = 0;
        private var totalTime:Number = 0;
        private static var last:uint = getTimer();
        private static var ticks:uint = 0;
        private static var text:String = "--.- FPS";
        [Embed(source = "../media/fonts/futura-fps.png")]
        private static const BitmapChars:Class;
        [Embed(source="../media/fonts/futura-fps.fnt", mimeType="application/octet-stream")]
        private static const BritannicXML:Class;
        public function FPS()
        {
              addEventListener(Event.ADDED_TO_STAGE, onAdded);
        }
        private function onAdded (e:Event):void
        {
            // creates the embedded bitmap (spritesheet file)
            var bitmap:Bitmap = new BitmapChars();
            // creates a texture out of it
            var texture:Texture = Texture.fromBitmap(bitmap);
            // create the XML file describing the glyphes position on the spritesheet
            var xml:XML = XML(new BritannicXML());
            // register the bitmap font to make it available to TextField
            TextField.registerBitmapFont(new BitmapFont(texture, xml));
            // create the TextField object
            bmpFontTF = new TextField(70, 70, "... FPS", "Futura-Medium", 12);
            // border
            bmpFontTF.border = true;
            // use white to use the texture as it is (no tinting)
            bmpFontTF.color = Color.WHITE;
```

```
            // show it
            addChild(bmpFontTF);
            // on each frame
            stage.addEventListener(Event.ENTER_FRAME, onFrame);
        }
        public function onFrame(e:Event):void
        {
            ticks++;
            var now:uint = getTimer();
            var delta:uint = now - last;
            if (delta >= 1000) {
                var fps:Number = ticks / delta * 1000;
                text = fps.toFixed(1) + " FPS";
                ticks = 0;
                last = now;
            }
            bmpFontTF.text = text;
        }
    }
}
```

如果要使用它，很简单，在你的 Starling 项目中任何位置添加如下代码即可：

```
// show the fps counter
addChild ( new FPS() );
```

下图展示了 FPS 类运作时的样子：

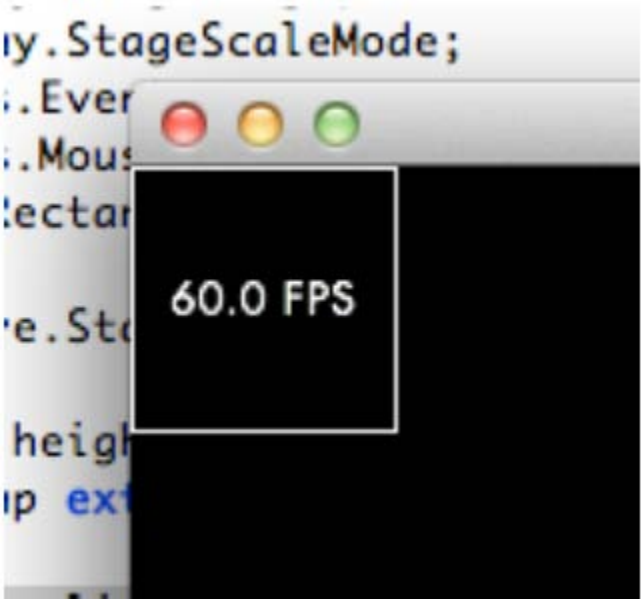

图 1.47
使用 GPU 渲染的 FPS 类

现在，我们就可以把这个 FPS 计数器放入之前那个带物理引擎的例子里面去了：

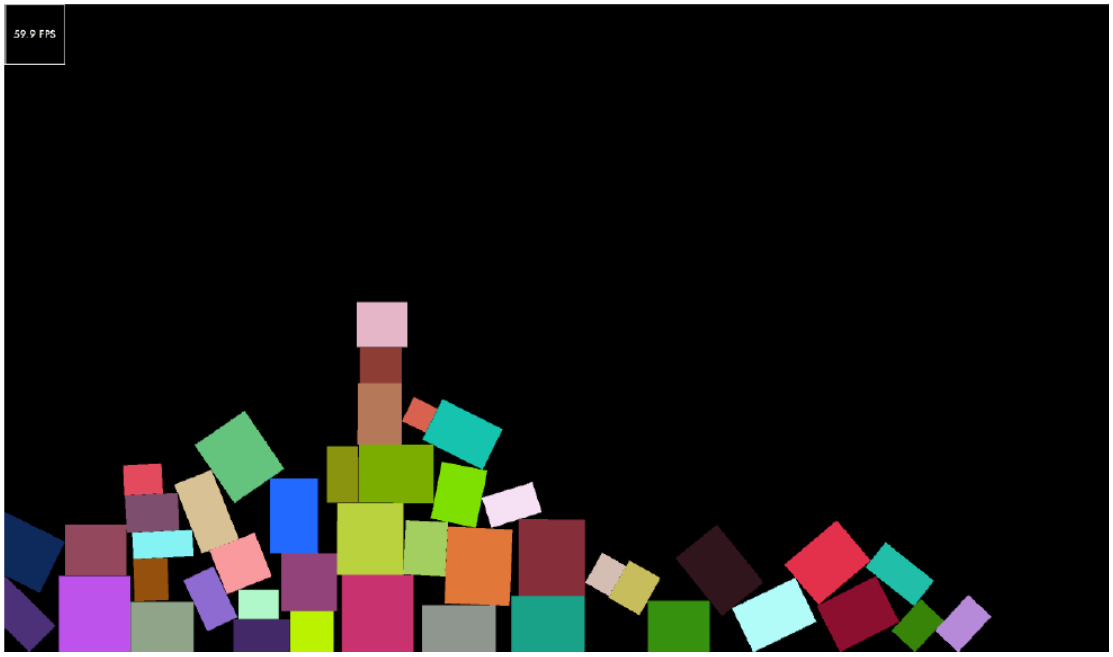

图 1.48
我们的 FPS 计数器被放入了应用程序中

现在呢，即使在手机设备上我们也可以进行无损的性能分析了。正如之前所说的，现在我们的应用程序中已经是清一色地由 GPU 来渲染的内容了，这样，原生显示列表就与我们没关系了，保证了呈现性能的稳定。
现在，之前提到的由mr.doob出品的Status类已经放出了为Starling框架开发的版本，可以在 http://forum.starling-framework.org/topic/starling-port-of-mrdoobs-stats-class找到。

粒子

每当提及漂亮的特效，我都会想起粒子特效，这玩意儿可是哥的最爱。粒子特效虽然看起来很绚丽，很酷，但是实现起来却不怎么复杂，不论你信不信，我反正是信了。从技术层面上来讲，所谓粒子效果，就是一些着色了的方块在不停地各自运动，并加以一些特殊的混合模式之后所得到的一种漂亮效果。
如果你准备在 Starling 中设计粒子效果，那么我推荐你一个非常方便的工具叫做 ParticleDesigner，该工具依然是由同一家公司（71 squared）在 GlyphDesigner（我们用此来制作位图字体）之后的出品的。下图是 ParticleDesigner 的一个截图，注意在右侧的模拟视图面板中我们可以预览到当前设置的粒子效果。

图 1.49

运行于 MacOS 的 ParticleDesigner

主窗口中提供了一系列的参数供设置。你可以花一些时间去简单地调一调试试这些参数，以配置出自己满意的效果。这里事实上还提供了一个 Randomize 按钮来让你生成一个参数随机的粒子效果。

下图显示了 ParticleDesigner 工具中点击 Save As 键后弹出的另存为窗口，选择好保存路径后将导出一个 ParticleEmitter 文件（可以理解成 XML 描述文件）和一个用以渲染粒子的纹理图片，这两个文件将被用作在 Starling 中创建 ParticleDesignerPS 对象。

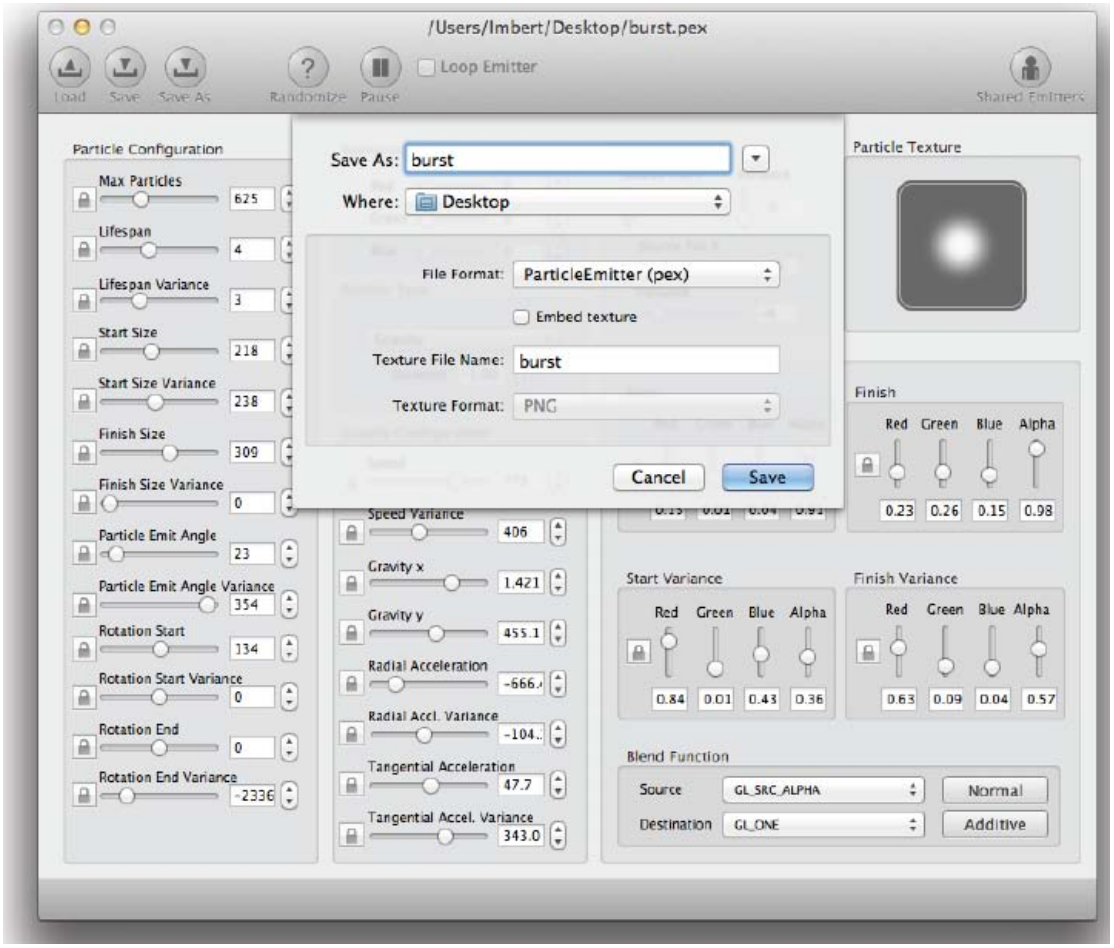

图 1.50

ParticleDesigner“另存为”窗口

下图显示了一个用 ParticleDesigner 创建素材并在 Starling 中实现的粒子效果：

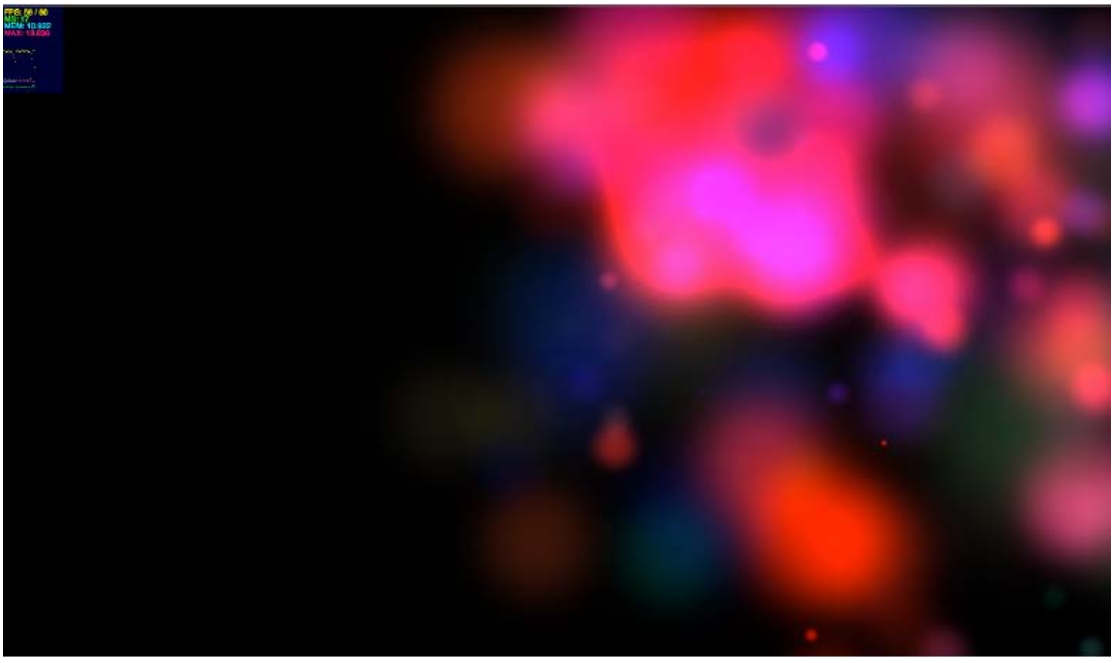

图 1.51

在 Starling 中运行的自定义粒子效果

很漂亮吧？如果你也想亲手试试，我先提醒你一下，Starling 中并不天生提供粒子效果的创建特性，是的，制作粒子效果的这个粒子引擎是一个 Starling 的插件，你可以在以下地址下载得到：

https://github.com/PrimaryFeather/Starling-Extension-Particle-System

【PS：具体使用方式在此粒子引擎下载后的压缩包中有实例代码】

下面给出另一个华丽的粒子效果图：

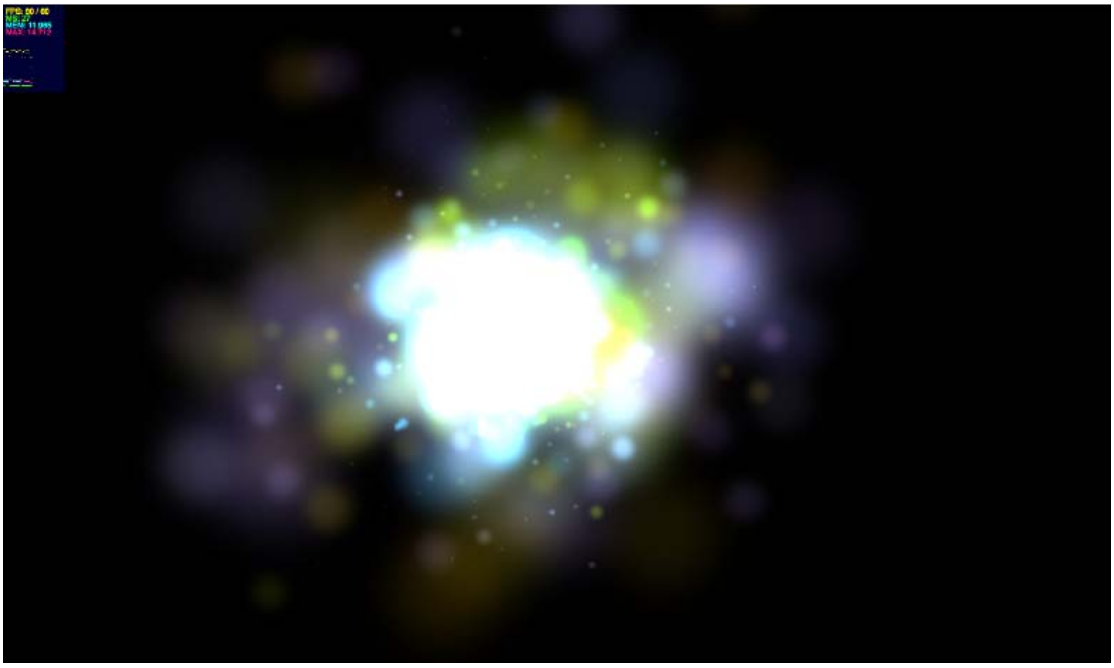

图 1.52

在 Starling 中运行的自定义粒子效果

需要注意的是，在 Starling 中，粒子被认为是一个动画对象，因此在使用它的时候应该把粒子对象添加到一个 Juggler 对象中才能看到动画的播放：

```
// load the XML config file
var psConfig:XML = XML(new StarConfig());
// create the particle texture
var psTexture:Texture = Texture.fromBitmap(new StarParticle());
// create the particle system out of the texture and XML description
mParticleSystem = new ParticleDesignerPS(psConfig, psTexture);
// positions the particles starting point
mParticleSystem.emitterX = 800;
mParticleSystem.emitterY = 240;
// start the particles
mParticleSystem.start();
// show them
addChild(mParticleSystem);
// animate them
```

```
Starling.juggler.add(mParticleSystem);
```

ParticleDesignerPS 对象是 DisplayObject 的子类，因此你可以用上所有你能想到的 API。当然，你还得注意做垃圾回收的工作，当粒子动画播放完毕后，你需要记得及时将粒子对象从 juggler 中移出，并对 ParticleDesignerPS 对象调用 dispose 方法进行对象销毁。

Lee Brimelow(leebrimelow.com)创建了下图所示的效果，一个粒子效果被创建并用来模拟宇宙飞船喷射出的火焰：

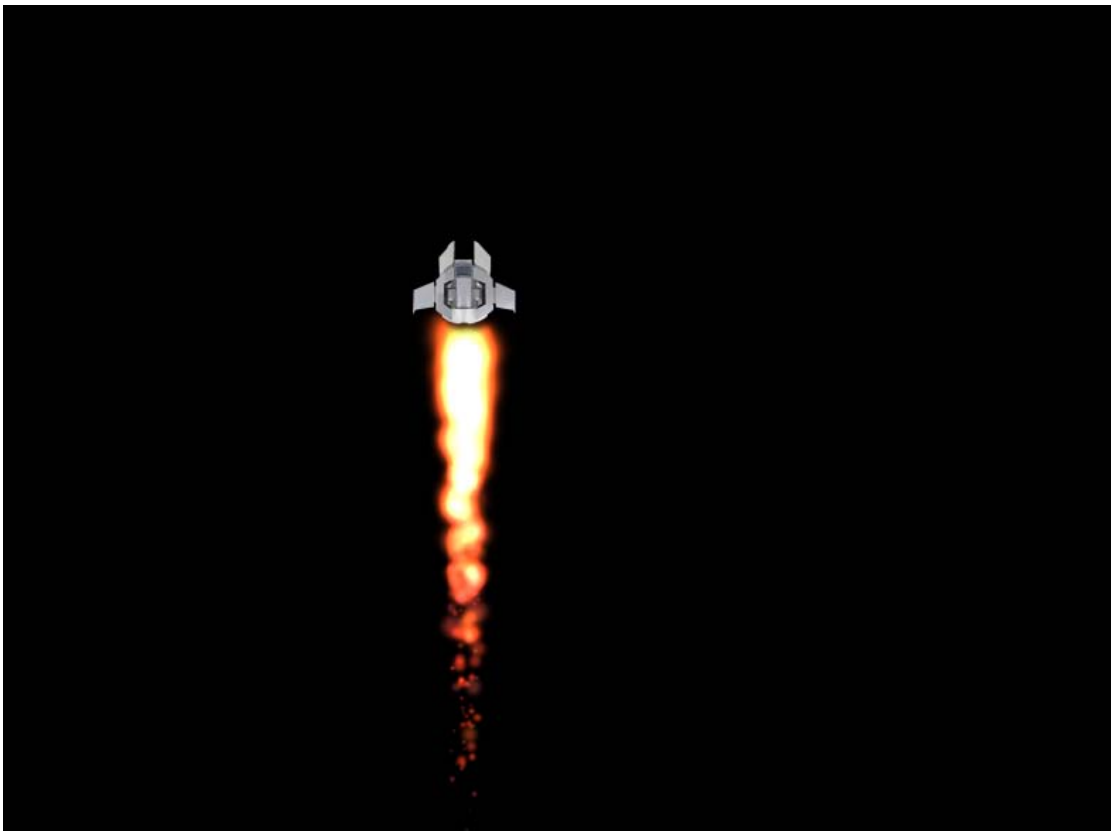

图 1.53

宇宙飞船与粒子效果

现在，我们还可以为宇宙飞船发射的导弹配套一个粒子效果：

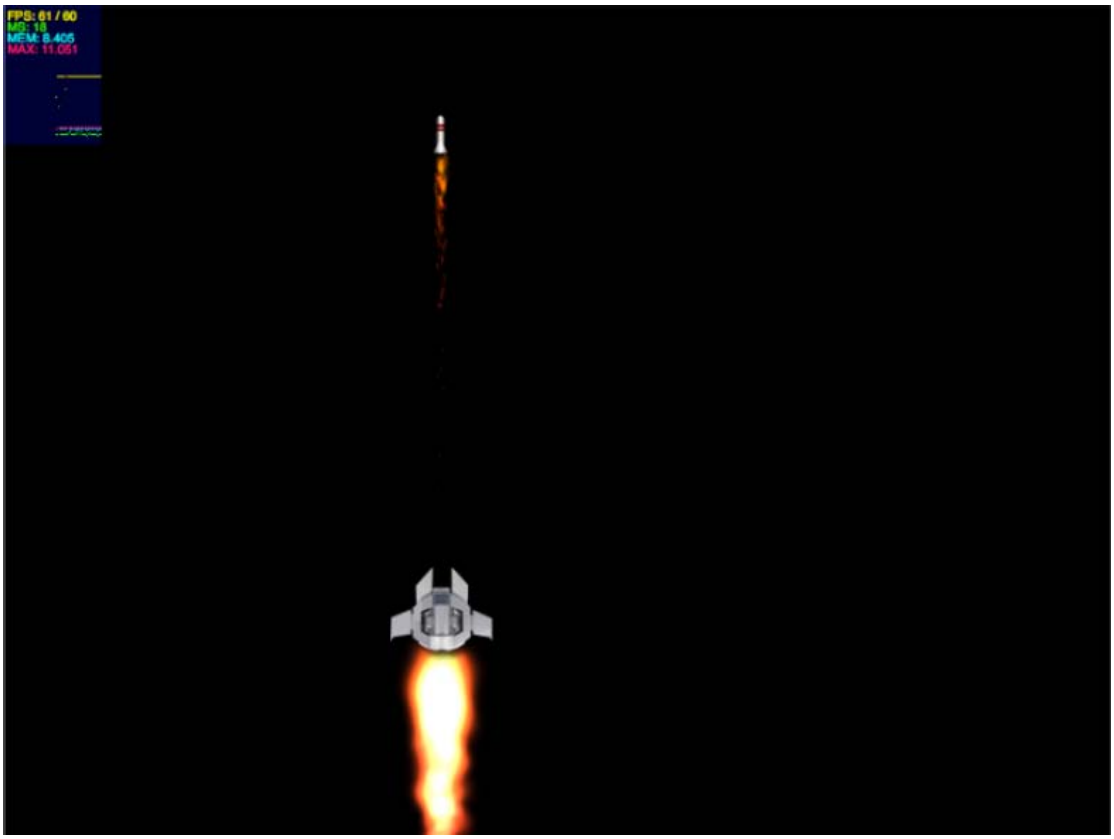

图 1.54
带火焰的导弹

一旦我们发射出去的导弹飞出了屏幕，我们就需要将其从 juggler 和舞台上移除：

```
Starling.juggler.remove(this.particle);
this.removeFromParent(true);
```

事实上我们刚才漏掉了一个细节，就是忘了销毁粒子对象了，不销毁粒子对象的话就无法释放其占用的 GPU 空间，因此，我们上面的代码应该改成：

```
Starling.juggler.remove(this.particle);
this.particle.dispose();
this.removeFromParent(true);
```

你还可以通过调用 removeChild 方法来一次性完成将粒子系统从舞台上移除及销毁对象的工作：

```
removeChild (particle, true);
```

让我们来考虑一个将来很有可能会遇到的潜在问题，就是作为导弹火焰的粒子效果中，如果有粒子移动出了屏幕范围我们需要将它移除掉。这一点通过检测粒子的当前坐标位置就能够很容易地判断出哪些粒子飞出了屏幕范围。
另一个潜在的问题是，当粒子在飞动的过程中会逐渐降低自己的透明度，我们需要把透明度

变成 0 的看不见的粒子移除掉。此时检测其当前坐标位置就不能帮助我们找出该销毁的粒子了，现在你需要知道的，仅仅是一个粒子效果何时播放完毕，并在播放完毕后进行销毁工作。幸运的是，在 ParticleDesignerPS 对象中提供了一个 isComplete 属性来指示一个粒子是否已经处于播放完结状态。我们可以在游戏每帧的主循环中遍历我们用来存储所有粒子效果的 Vector 对象，依次检测它们的 isComplete 属性：

```
for each (var p:ParticleDesignerPS in particlesVector)
{
   if ( p.isComplete )
      removeChild(p, true);
}
```

每次我们需要创建粒子效果的时候记得把它们添加进 particlesVector 对象中，这样就可以确保每帧在游戏主循环中都会自动检测粒子效果的当前状态，并在粒子效果播放完毕后自动完成销毁的工作。
好了，对 Starling 框架的介绍已经差不多了，我希望列位能够喜欢本教程，是时候让我们使用它来为我们创建丰富多彩的应用程序了！

工作人员

我首先要谢谢 Chris Georgenes(mudbubble.com)为本教程提供那么棒的素材。
其次我要感谢 Daniel Sperl(Starling 框架作者)，能和你一起工作真是寡人之荣幸也。

本书翻译：S_eVent
联系我：www.iamsevent.com