Introducing Starling – bytearray.org – revision 1.2.6

Starling 帮助手册 中文版

第一部分 整理自 Adobe 官方网页[1]

第二部分 由 Slicol 翻译、校对并验证[2]

[1] http://www.adobe.com/cn/devnet/flashplayer/articles/introducing_Starling.html

[2] 由于原文过于随意和口语化，故在翻译时，少数地方使用了意译

目录

译者序

Starling 是一个优雅的框架。

第一次听说 Starling，是在 Flash 开发者大会上。

今年有幸参与了一次 Flash 开发者大会，并在会上发言，向大家介绍了 VFS 在 Flash 开发中的应用。我很关注传统游戏开发技术在 Flash 项目中的应用。Adobe 官方开始推出像 Alchemy 这样的技术，用于将传统游戏开发技术应用于 Flash 项目，Unity3D 也能够将传统 3D 游戏直接导出为 Flash 程序，后来 Stage3D 推出，使 Flash 一跃拥有了驾驭 3D 项目的能力。从而可以看到，传统游戏开发技术与 Flash 技术之间的隔阂越来越薄。Flash 越来越能胜任原本只能用 C++这样的传统技术才能完成的大型项目。

这对于 Flash 开发者来说，无疑是一个好消息。但是 Flash 开发者毕竟习惯了传统 Flash 开发的那一套方式，要想直接操作 Stage3D 接口，还是有一定门槛的。并且，大部分 Flash 开发者只有 2D 游戏的开发经验，大部分成功运营的 Flash 商业项目也都是 2D 项目。因此，如何利用 Stage3D 的 GPU 加速来提升 2D 项目的渲染能力，便比起直接去驾驭 3D 技术更加贴近我们的实际需要。

Starling 框架应运而生。

Peter 在大会上演示了用 Starling 框架开发的实际 Demo，其效果很让人惊艳。并且已经有在 AppStore 上发布了基于 Starling 的商业作品。这让我对 Starling 产生极大兴趣。于是立即下载 Starling 框架，并仔细阅读了它的帮助手册，然后惊讶地发现，很快就上手了。

我很快将 3 年前用 BitmapData 实现的粒子系统移植到 Starling 框架里，并且在 IPad 上运行，对比其效果与性能，比原来的好太多，已经基本上与文本最后面“粒子”章节的示例效果一样。这让我惊喜不已。

移植它，只花了不到 1 个小时。

所以，这绝对是一个值得去推荐和学习的框架。这是我翻译这份帮助手册的原动力。

由于 Adobe 官方网站上已经有了一部分中文译文，出于对官方文案的尊重，我没有重复翻译这一部分。而是将手册拆为 2 大部分：第一部分直接引用官方原文，只作重新排版和校对；第二部分，则根据我自己的理解进行直译或者意译，并且进行校对。

Slicol/腾讯/深圳
slicol@qq.com
http://www.slicol.com

第一部分

1、 Starling 是什么？

Starling 是在 Stage3D APIs 基础上开发的一种 ActionScript 3 2D 框架（可用于 Flash Player 11 和 Adobe AIR 3 的桌面）。 Starling 是为游戏开发设计的，但是你可以将它应用于很多其它的应用程序。 在不必涉及低级 Stage3D APIs 情况下，Starling 使得编写具有快速 GPU 加速功能的应用程序成为可能。

1、 为什么会有 Starling?

大多数 Flash 开发人员希望利用这种能力提高 GPU 的加速功能（通过使用 Stage3D 技术），而不必编写如此高级的框架和深入研究低级的 Stage3D APIs。 Starling 是完全基于 Flash Player APIs 而设计，并将 Stage3D（Molehill）复杂性抽象化。 因此每个人都能看到直观的程序。

Starling 是为 ActionScript 3 开发人员而设计，尤其是这些涉及 2D 游戏开发的人员。 在使用 ActionScript 3 之前，你必须基本了解它。由于 Starling 轻便、灵活并易于使用，你也可以将它应用于其它项目需求，例如 UI 编程。 这种框架要求设计得越直观越好，因此任何 Java™ 或者.Net™ 开发人员都可以马上开始使用它。

2、 设计理念

（1）简洁

Starling 直观并易于使用。Flash 和 Flex 开发人员能够快速地了解它，因为它遵循大多数 ActionScript 规则并将低级 Stage3D APIs 的复杂性抽象化。Starling 使用熟知的概念，例如 DOM 显示列表、事件模型以及熟知的如 MovieClip、Sprite、TextField 等 APIs，而不是依靠诸如顶点缓冲（vertices buffer）、透视矩阵（perspective matrices）、着色程序（shader programs）和组合字节码（assembly bytecode）进行编码。

（2）轻量

Starling 在很多领域都很轻便。 类的数量是有限的（大概有 80k 的代码）。 除了 Flash Player 11 或者 AIR 3（以及在未来的版本中使用的移动支持）之外，它没有外部依赖。 这些因素使得你的应用程序很小并使你的工作流程简单。

（3）免费

Starling 能够免费使用并富有朝气。 它根据 Simplified BSD 许可获得授权，因此你可以免费

地使用它，即便是在商业应用程序中也是如此。 我们每天都在使用它并且我们依靠一个活跃的团队不断地完善它。

3、 使用概述

在后台操作中，Starling 使用 Stage3D APIs —它们是在桌面上基于 OpenGL 和 DirectX ，在移动设备上基于 OpenGL ES2 而运行的低级的 GPU APIs。 需要重点注意的是，Starling 是 Sparrow 的 ActionScript 3 端口，它等同于基于 OpenGL ES2 APIs 的 ISO 库（参见图 1）

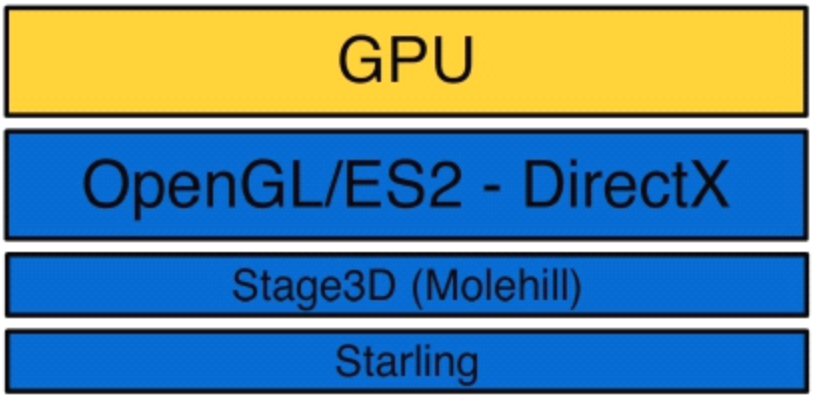

图 1.1 Stage3D (Molehill) 分层位于 Starling 之上（即显示在 Starling 之下）

Starling 重新创建了很多 Flash 开发人员熟知的 APIs 。下图列举了通过 Starling 暴露的图形元素 APIs（参见图 2）。

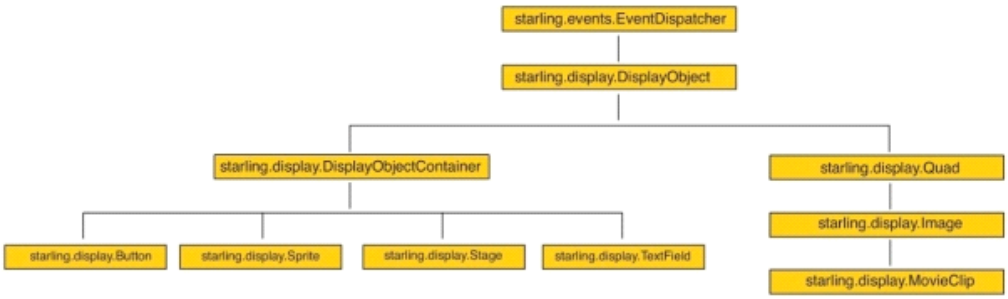

图 2. Starling 支持 DisplayObject 继承

在 3D GPU APIs 基础上可以创建 2D 内容，这看起来有点奇怪。 当涉及 Stage3D APIs 时，很多人认为这些 APIs 是严格地限制在 3D 内容中的。 实际上这是名称造成的误解：如果它叫做 Stage3D ，那么你怎么可以使用它创建 2D 元素呢？下图说明了关于使用 drawTriangles API 绘制 MovieClip 能力的问题（参见图 3）。

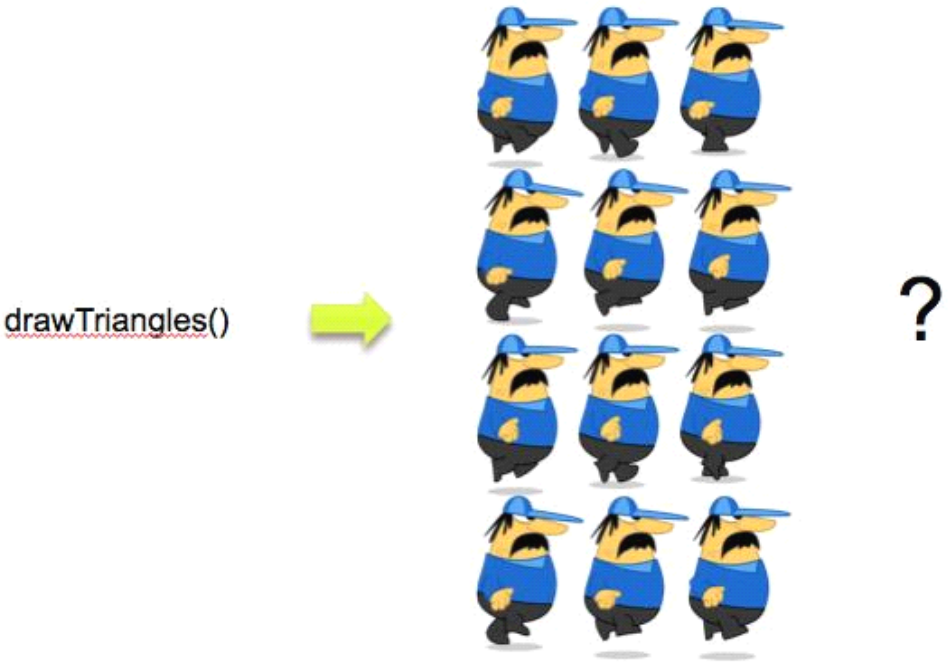

图 3. 可以使用 drawTriangles API 创建 2D 影片剪辑吗？

GPU 具有较高的效率并能快速地绘制三角形。 通过使用 drawTriangles API ，你可以绘制两个三角形，然后选取一种纹理并且使用 UV 映射将它应用到三角形中。 这样可以创建了一个带有纹理的四边形，它代表一个 sprite 。 通过更新每一个帧上的三角形的纹理，最后的结果就是一个 MovieClip 。

幸好我们没有必要通过这些细节使用 Starling 。 你只需要提供帧数，将它们提供给一个 Starling MovieClip ，这就是所有需要做的（参见图 4）。

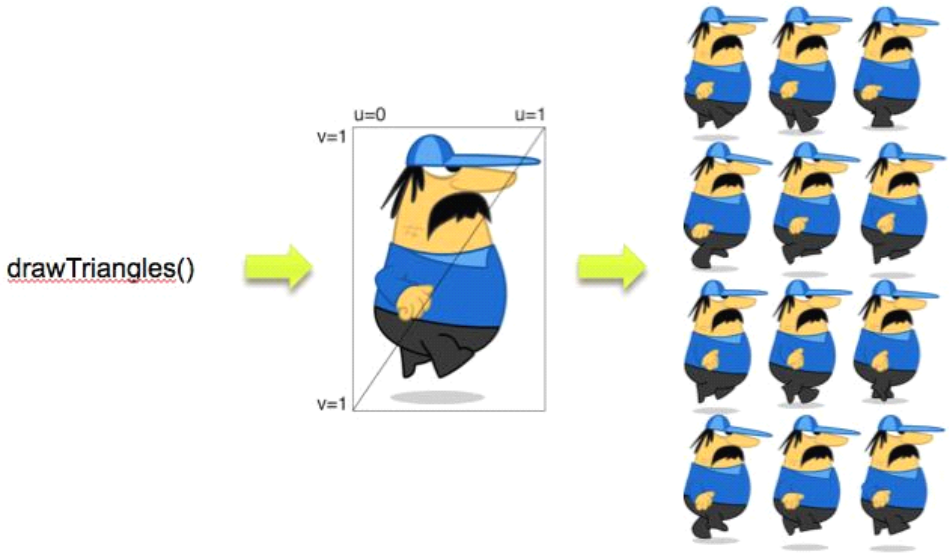

图 4. 使用 drawTriangles API 和一个带有纹理的四边形，你可以创建一个 2D 图形

为了更好地了解 Starling 如何降低复杂性，检查你必须写入的代码以便于用低级的 Stage3D APIs 显示简单的带有纹理的四边形。

```
// 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(Context3DProgramType.VERTEX, 0, m, true);

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

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

上述范例中的代码创建了一个正方形的 2D 实例（参见图 5）：

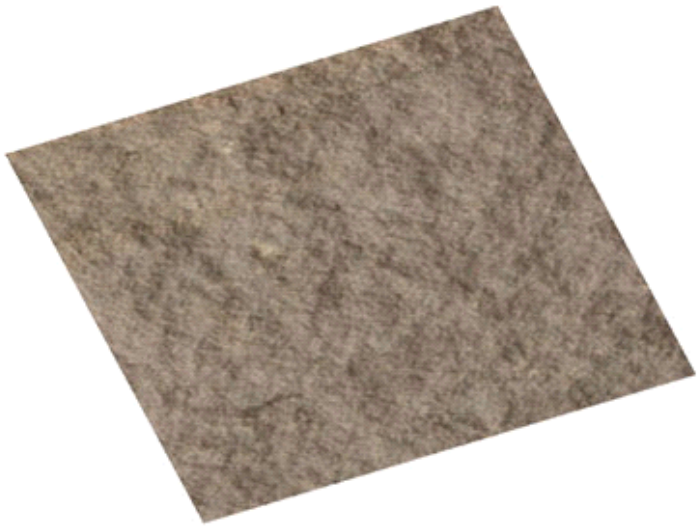

图 5. 使用 drawTriangles API 和一个带有纹理的四边形创建一个 2D 对象的结果

上述范例中所示的代码无疑是非常复杂的。 那是访问低级 APIs 需要付出的代价。 从另一方面说，你可以控制很多方面，但是它需要大量的代码来设置一切。

通过 Starling，你可以编写如下代码替换上述代码：

```
// 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
quad.pivotX = 50;
quad.pivotY = 50;
quad.x = 300;
quad.y = 150;
quad.rotation = Math.PI/4;

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

作为一个熟知如何使用 Flash APIs 的 ActionScript 3 开发人员，你可以立即使用这些已暴露的 APIs 开始工作，与此同时 Stage3D APIs 的所有复杂部分都可以在后台进行处理。

如果你使用重绘区域（redraw regions）功能进行试验，在 Starling 将在 Stage3D 上，而不是在预期的传统显示列表上渲染一切。 如下的截图说明了这种行为。 该四边形在每一帧上旋转，重绘区域（redraw regions）只显示 FTP 计数器，而不是 Stage3D 的内容（参见图 6）：

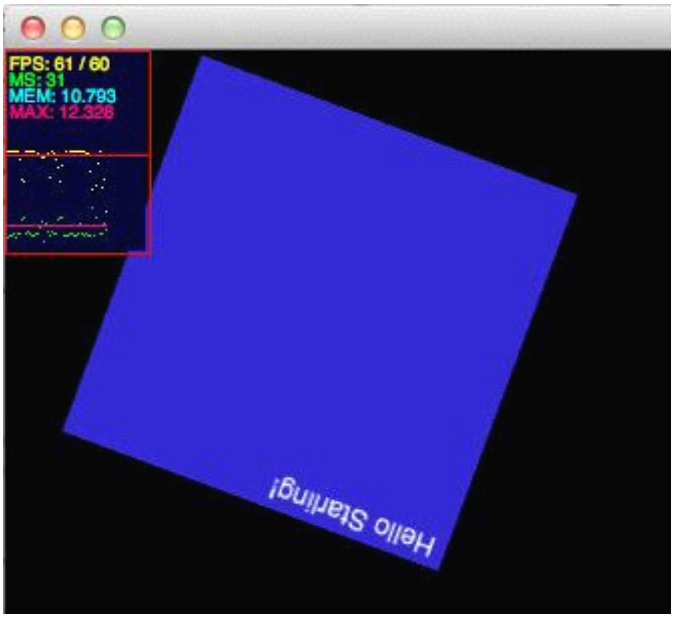

图 6. 使用 Stage3D 渲染内容的范例

需要记住在使用 Stage 3D 构架时，通过 GPU 可以完全地渲染并合成相应的内容。 因此，在 GPU 上运行的用于显示列表的重绘区域（redraw regions）功能不能使用。

4、 分层限制

当你使用 Starling（以及 Stage 3D）时，记住开发内容有一个限制。 正如之前所述，Stage3D 完全是嵌入在 Flash Player 中的全新的渲染构架。 GPU 表层放置在显示列表之下，这意味着任何在显示列表中运行的内容将放置到 Stage3D 内容之上。 在编写这篇文章时，运行在显示列表里的内容仍然不能放置在 Stage3D 分层之下（参见图 7）。

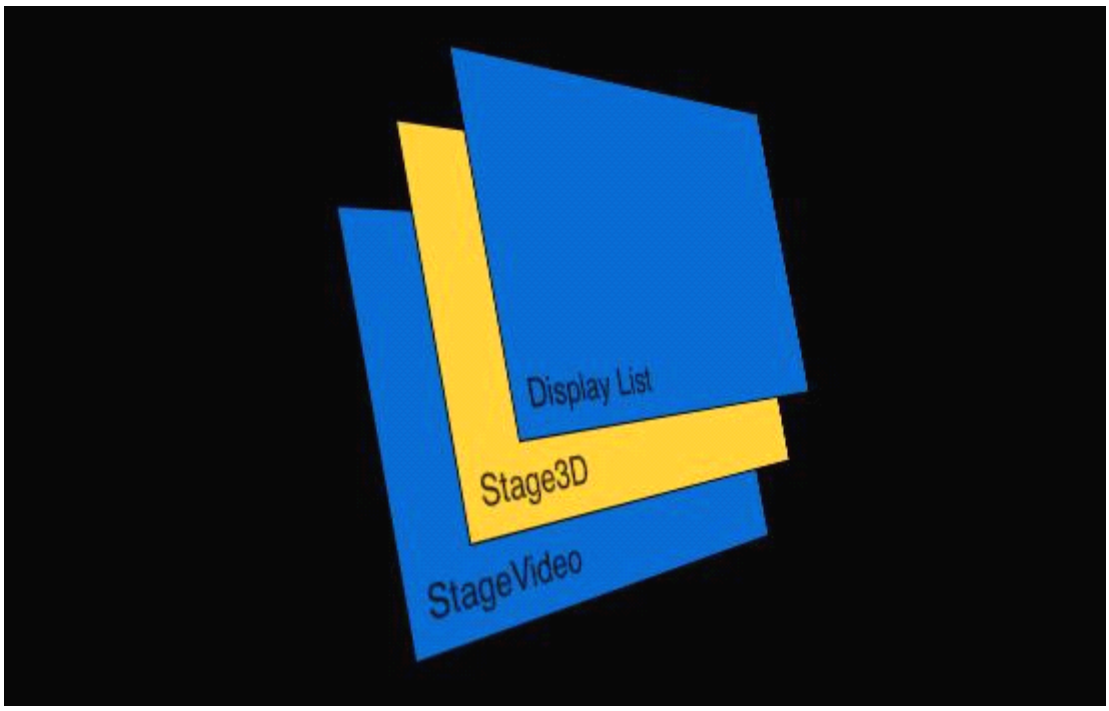

图 7. 使用 Stage3D 渲染内容的堆叠顺序

此外，还要注意 Stage3D 对象不是透明的。 如果这是可能的，那么你可以使用 Stage Video 技术（Flash Player 10.2 引入的功能）播放视频，同时可以用通过 Stage3D 渲染的内容覆盖视频。 希望未来的 Flash Player 版本支持这一功能。

5、 设置项目

你可以访问官方的 Github 页面下载 Starling。 此外，你可能发现访问 Starling 网站也会受益匪浅。

Starling 根据 Simplified BSD 许可获得授权，所以你可以在任何类型的商业或者非商业项目上使用 Starling。 如果你需要更多的信息，你可以联系 Starling 框架团队。

在你下载 Starling 之后，你可以像引用其它 AS3 库一样引用 Starling 库。为了使用 Flash Player 11 beta 的新版本，你必须把 SWF 版本 13 作为目标，这是通过将额外的编译器参数即 -swf-version=13 传递给 Flex 编译器实现的。如果你正在使用 Adobe Flex SDK，那么请按照如下步骤操作：

1) 为 Flash Player 11 下载新的 playerglobal.swc。
2) 从 Flex 4.5 SDK 表中下载 Flex 4.5 SDK (4.5.0.20967)。
3) 将相应的版本安装到你的开发环境中。
4) 在 Flash Builder 中，通过选中 File > New > ActionScript project 创建一个新的 ActionScript 项目。
5) 打开 Property inspector（右击并选中 Properties 选项）。 在左边的菜单列表中，选中 ActionScript Compiler。

6) 使用右上角的 Configure Flex SDK 选项将项目指向 Flex build 20967。单击 OK。
7) 设置你的项目目标为 SWF 版本 13。
8) 打开 Property inspector 并从左侧菜单列表选中 ActionScript Compiler。
9) 将 -swf-version=13 添加至 'Additional compiler arguments' 输入。这就保证了输出的 SWF 把 SWF 版本 13 当做目标版本。如果你在命令行而不在 Flash Builder 中进行编译，那么你必须添加相同的编译器参数。
10) 核查你已经在你的浏览器中安装了新的 Flash Player 11 版本。

6、 设置场景

在你已经准备好了你的开发环境之后，你就可以深入研究相应的代码，并且看看你如何能够充分利用这一框架。 使用 Starling 非常简单，你只需创建一个 Starling 对象并添加到你的主类即可。 在本文中，当涉及到诸如 MovieClip, Sprite 以及其它对象时，我所指的都是 Starling APIs，而不是来源于 Flash Player 的本地对象。

首先，Starling 构造器（constructor）需要多重参数。下面是签名：

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

事实上，只有前面 3 个经常使用。 相关的 rootClass 参数需要一个至扩展 starling.display.Sprite 的类的引用，而第二个参数是我们的 stage，然后是一个 Stage3D 对象：

```
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 is better quality but slower    performance)
               mStarling.antiAliasing = 1;

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

```
        }
    }
}
```

在下面的代码中，Game 类在被添加到 Stage 时可以创建一个简单的四边形：

```
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 );
        }
    }
}
```

上述代码将一个侦听器添加到 Event.ADDED_TO_STAGE 事件中，并在事件处理程序中对应用程序进行初始化。 这样你就可以安全地访问 Stage。

注意： 关注一下这个微妙的细节：上面描述的 Game 类从 starling.display 程序包中，而不是从 flash.display 程序包中扩展了 Sprite 类。必须检查你的导入语句并确保你不是使用本地 API 来替代 Starling API。

正如在 Flash 中所预期的，Starling 中的对象有一个默认的位置 0,0。因此添加几行命令使四边形位于 Stage 的中央：

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

现在，测试一下项目以便于观察相应的结果（参见图 8）：

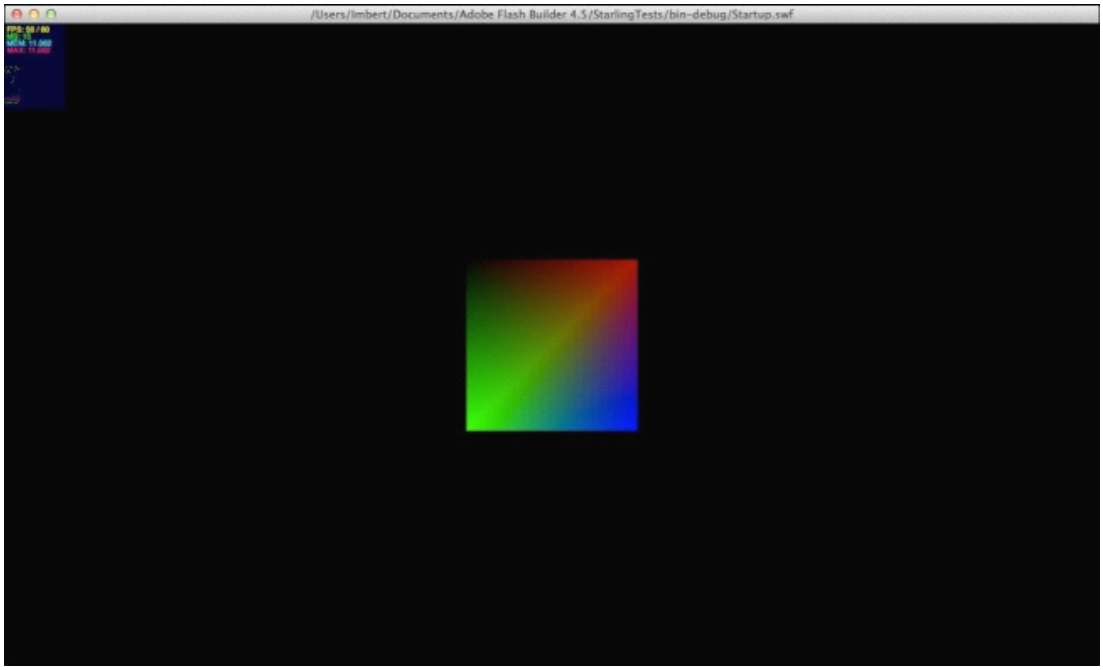

图 8. 四边形位于 Stage 的中央

注意锯齿消除功能（anti-aliasing）值允许你设置锯齿消除功能所需的类型。 一般来说，值为 1 就基本上可以接受，但是你可以选择其它值。该框架支持的锯齿消除功能（anti-aliasing）值的变化范围是 0 到 16，但是，下面的列表给出了最常用的值：

- **0**: 无锯齿消除（anti-aliasing）
- **2**: 最低程度的锯齿消除（anti-aliasing）
- **4**: 高质量的锯齿消除（anti-aliasing）
- **16**: 极高质量的锯齿消除（anti-aliasing）

你很少需要用到超过 2 的设置，尤其是对 2D 内容。 然而，根据你的项目要求，你需要针对具体情况作出相应的决定。 在图 9 中，比较一下两个截图，观察两个锯齿消除（anti-aliasing）值（1 和 4）之间的细微差别。

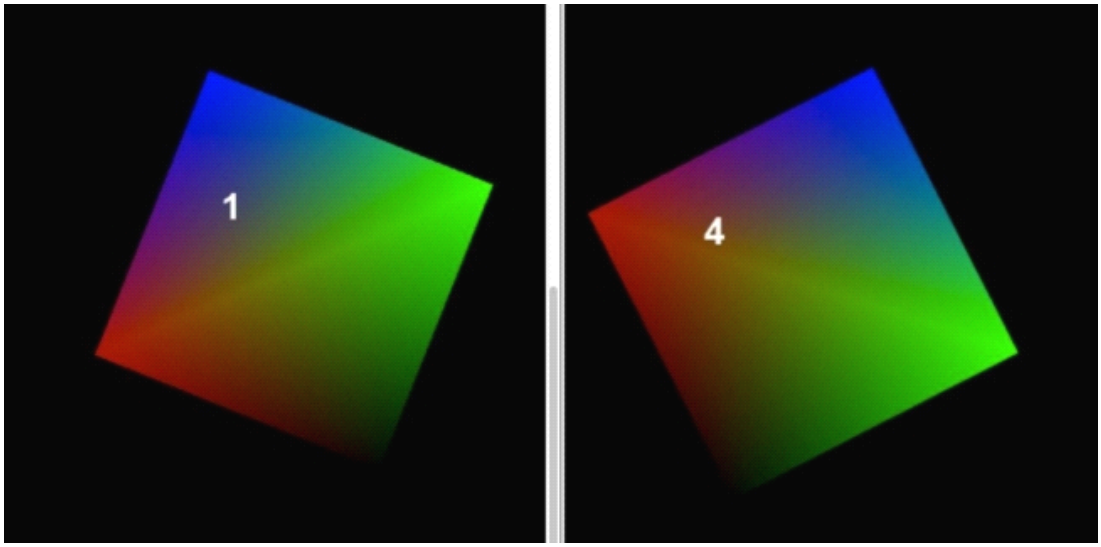

图 9. 比较一下锯齿消除（anti-aliasing）值为 1（左）和 4（右）之间的视觉差别

试验一下使用 2 以上的值为你的项目设置所需的质量。当然，选择较高的值会影响性能。注意 Stage3D 不会受到 SWF 文件的 Stage 质量影响。

下面给出能够与 Starling 对象一起使用的其它 API 的描述：

- **enableErrorChecking**: 允许你启用或者禁止启用错误检查。 指定是否将渲染器遇到的问题报告给应用程序。当 enableErrorChecking 设置为ture时，Starling 内部调用的 clear() 和 drawTriangles() 方法是同步的并可以抛出错误。 当 enableErrorChecking 设置为 false 时，clear() 和 drawTriangles() 方法是异步的且不报告错误。 启用错误检查将会减弱渲染性能。 只有当调试项目时启用错误检查功能，而在部署最终版本前禁止启用该功能
- **isStarted**: 指示是否调用了 start
- **juggler**: juggler 是一个简单对象。它仅保存了一列执行 IAnimatable 的对象，并且在被要求这样做的情形下提前了它们的时间（通过调用它自己的 advanceTime:方法）。当一个动画完成时，它将会将其抛弃
- **start**: 开始渲染和进行事件处理
- **stop**: 停止渲染和进行事件处理。 当游戏进入后台运行状态以节约资源时，使用这个方法可以停止渲染
- **dispose**: 当你希望处理当前 GPU 内存上已渲染的全部内容时，调用这个方法。 该API能够在其内部处理了一切事务（例如着色程序（shader programs）、纹理和其它一切事务）

一旦创建了你的 Starling 对象，调试记录会自动地输出，显示关于渲染的信息。 在默认情形下，当 SWF 文件正确地嵌入到页面或者当在独立的 Flash Player 中进行测试时，Starling 会输出如下代码：

```
[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，特定的硬件细节将会随着你的配置而变化。 上述信息表明已经使用了 GPU 加速功能，因为它包括驱动版本的细节。 为了便于调试，你可能希望能够强迫 Flash Player 内部使用的软件回退，以便了解当你的内容在软件上运行时它的表现如何。

添加如下的代码以便于通知 Starling 你希望使用软件回退功能（software rasterizer）：

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

当你使用软件时，输出的信息会确认你正在使用 software 模式：

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

确保你也在 software 模式下测试了你的内容，以便于更好地了解它在这种模式下的性能。如果用户的配置使用旧版本的驱动（为了保持一致性，所有 2009 年之前的驱动都包含于黑名单中），那么你的内容可能回退到软件。

在下一节中，当你将你的 SWF 文件嵌入到页面时，你需要看一下 Stage3D 的要求。

7、 Wmode 要求

你必须记住为了启用 Stage 3D 和 GPU 加速功能，在页面中你必须使用 wmode=direct 作为嵌入模式。 如果你没有指定任何值或者选择除“direct”之外其它值，例如 “transparent”、“opaque” 或 “window”，则 Stage 3D 将均不可用。相反，当 requestContext3D on Stage3D 被调用时，你会得到一个运行时异常的提示，告知你 Context3D 对象的创建失败。

下图列举了一个运行时异常的对话框：

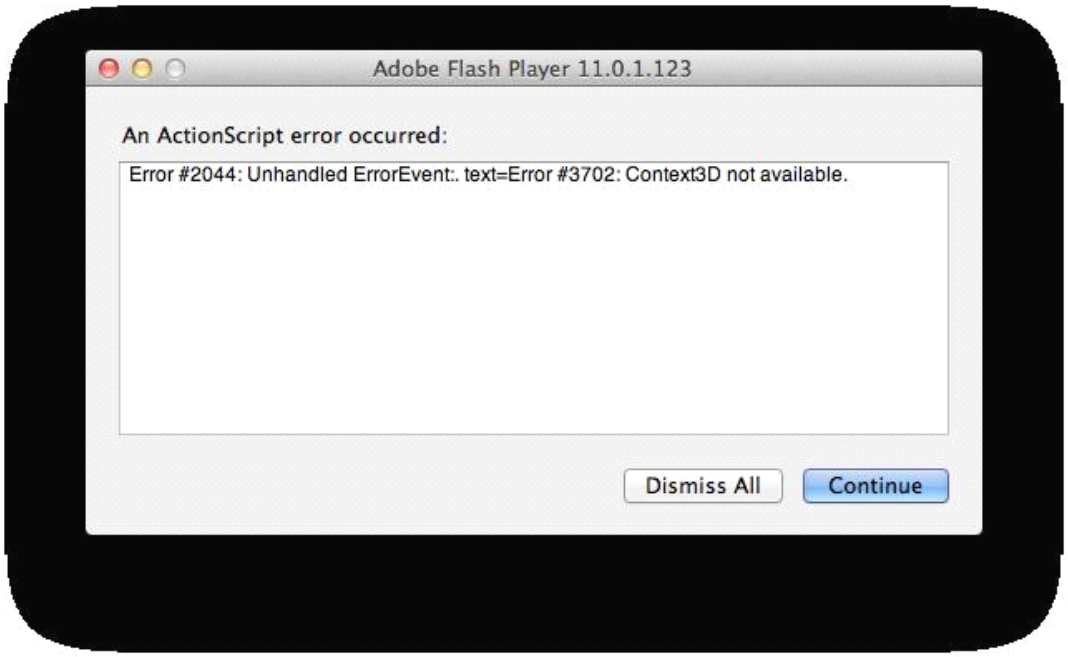

图 10. 在 Context3D 不可用的情形下，运行时异常对话框

如果你的应用程序嵌入时使用了错误的 wmode，那么必须小心处理这种情形。 你需要通过显示一条解释这一问题的信息以便给出合理的响应。 幸运的是，Starling 为你自动地处理了这一问题并显示如下信息：

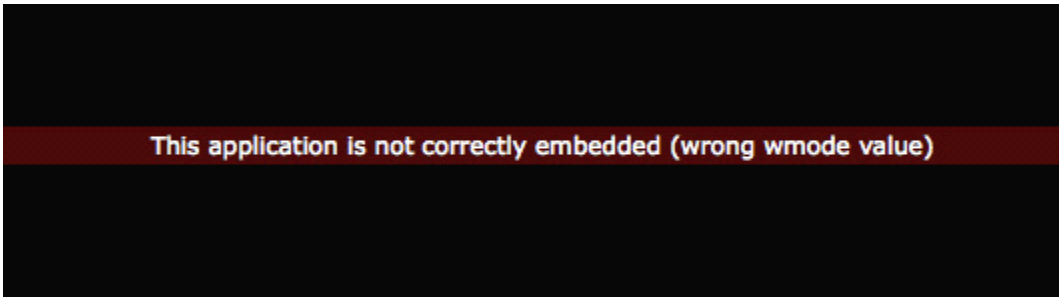

图 11. 当应用程序没有正确嵌入时显示的警告信息

8、 Stage 质量

作为一个 Flash 开发人员，stage 质量的概念对你来说并不陌生。 记住当使用 Stage3D 以及作为结果的 Starling 时，stage 质量不会影响相应的性能。

9、 渐进的增强功能

当 GPU 加速功能不起作用时，Stage3D 将回退到软件中，并且将在内部使用一个名称为 SwiftShader (Transgaming)的软件回退引擎。 为了保证你的内容在此种情形下运行正常，你需要检测什么时候你应该在 software 模式下运行，并且移除在 software 模式下可能会减慢运行速度的潜在影响。

在 2D 内容环境下，软件回退功能能够处理很多对象并提供良好的性能，但是，为了检测这一点，你仍然可以使用静态的属性环境从 Starling 对象中读取 Context3D 对象：

```
// are we running hardware or software?
var isHW:Boolean = Starling.context.driverInfo.toLowerCase().indexOf("software") ==
-1;
```

记住使用软件回退功能设计你的内容是一种很好的做法，它将提供一种渐进式的体验，从而确保在任何情形下都能获得最佳体验效果。

第二部分

10、 显示列表（The Display List）

Starling 像原生 Flash 一样有一个自己的显示列表，并且 Starling 的显示对象（DisplayObject）也有 stage 属性。当显示对象被添加到显示列表后，它的 stage 属性才是可用的。为了能够成功获取到 stage 属性，需要确保显示对象被添加到显示列表。这个时候，需要依赖一些重要的事件：

- **Event.ADDED**：当显示对象被添加到另一个显示对象上时抛出。
- **Event.ADDED_TO_STAGE**：当显示对象被添加到一个已经存在于显示列表中的显示对象时抛出。
- **Event.REMOVED**：当显示对象从它的父级对象移除时抛出。
- **Event.REMOVED_FROM_STAGE**：当对显示象从一个存在于显示列表中的父级对象移除时抛出。

通过监听这些事情，可以帮忙我们在适当的时候初始化或者释放对象，从而优化性能和资源。

下面是 DisplayObject 类所定义的一些方法：

- **removeFromParent：**安全地将显示对象从它的父级对象（如果有的话）移除；
- **getTransformationMatrixToSpace：** 创建一个用于表示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；

下面是 DisplayObject 类定义的一些属性，有些属性是原生 DisplayObject 所没有的，比如 pivotX 与 pivotY 属性可以让我们动态改变 DisplayObject 的注册点（Registration Point）。

- **transformationMatrix**：当前显示对象位置相对其父容器的转换矩阵；
- **bounds**：同原生DisplayObject；
- **width**：同原生DisplayObject；
- **height**：同原生DisplayObject；
- **root**：同原生DisplayObject；
- **x：**同原生DisplayObject；

- **y：** 同原生DisplayObject；
- **pivotX**：当前显示对象的注册点X坐标值，默认为0；
- **pivotY**：当前显示对象的注册点Y坐标值，默认为0；
- **scaleX**：同原生DisplayObject；
- **scaleY**：同原生DisplayObject；
- **rotation**：同原生DisplayObject，但单位是弧度，而非角度；
- **alpha**：同原生DisplayObject；
- **visible**：同原生DisplayObject；
- **touchable**：指定当前显示对象是否能够接受 Touch 事件；
- **parent**：同原生DisplayObject；
- **stage**：同原生 DisplayObject；

在 Starling 里，Sprite 类是你能使用到的最轻量的显示对象容器，它继承自 DisplayObjectContainer（DisplayObjectContainer 则继承自 DisplayObject）。Sprite 继承了 DisplayObjectContainer 和 DisplayObject 的所有方法和属性，也有自己的方法和属性。我们可以直接使用 Sprite 来嵌套其它显示对象，但是目前为此，我们都不这样做。而是定义一个自定义的类来继承 Sprite，比如 Game 类，用这个类来嵌套其它显示对象。

下面是 DisplayObjectContainer 的方法：

- **addChild**：同原生DisplayObjectContainer；
- **addChildAt**：同原生DisplayObjectContainer；
- **dispose**：完全销毁一个对象，释放其在GPU中所占显存，移除其全部侦听；
- **removeFromParent**：将它从自己的父级对象移除；
- **removeChild**：同原生DisplayObjectContainer，但如果被移除者不是它的子级对象，什么也不会发现（不会像原生DisplayObjectContainer那样报错）；
- **removeChildAt**：同原生DisplayObjectContainer；
- **removeChildren**：移除所有的子级对象；
- **getChildAt**：同原生DisplayObjectContainer；
- **getChildByName**：根据对象的名字来返回一个子级对象（不会递归到其子级对象的子级对象）；
- **getChildIndex**：同原生DisplayObjectContainer；
- **setChildIndex：**同原生DisplayObjectContainer；
- **swapChildren：**根据两个传入的子级对象交换其索引；
- **swapChildrenAt：**直接传入两个索引，交换其对应子级对象的索引；
- **contains**：同原生DisplayObjectContainer (会递归到其子级对象的子级对象)；

由于 stage 是继承 DisplayObjectContainer 的，你可以在 stage 上调用大部分的 DisplayObjectContainer 的方法。比如也可以给 stage 设置颜色。如果没有给 stage 设置颜色，则 Starling 默认使用原生 Flash 的背景色。你可以使用如下 SWF 标签给 Flash 定义背景色。

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

你可以定义一个自定义类，将它添加到显示列表，然后在这个类里获取到 stage 属性，通过它将颜色设置给 stage：

```
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
         {
              // 将背景色设为蓝色
              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 );
         }
    }
}
```

上面的例子中，我们没有使用任何纹理，简单地使用 2 个三角形组成一个四边形的平面。该平面的每一个顶点都设置不同的颜色，然后让 GPU 进行插值填充，以得到我们想要的渐变色。

当然，如果你只需要纯色的话，直接使用 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 );
            }
        }
    }
```

运行结果如下：

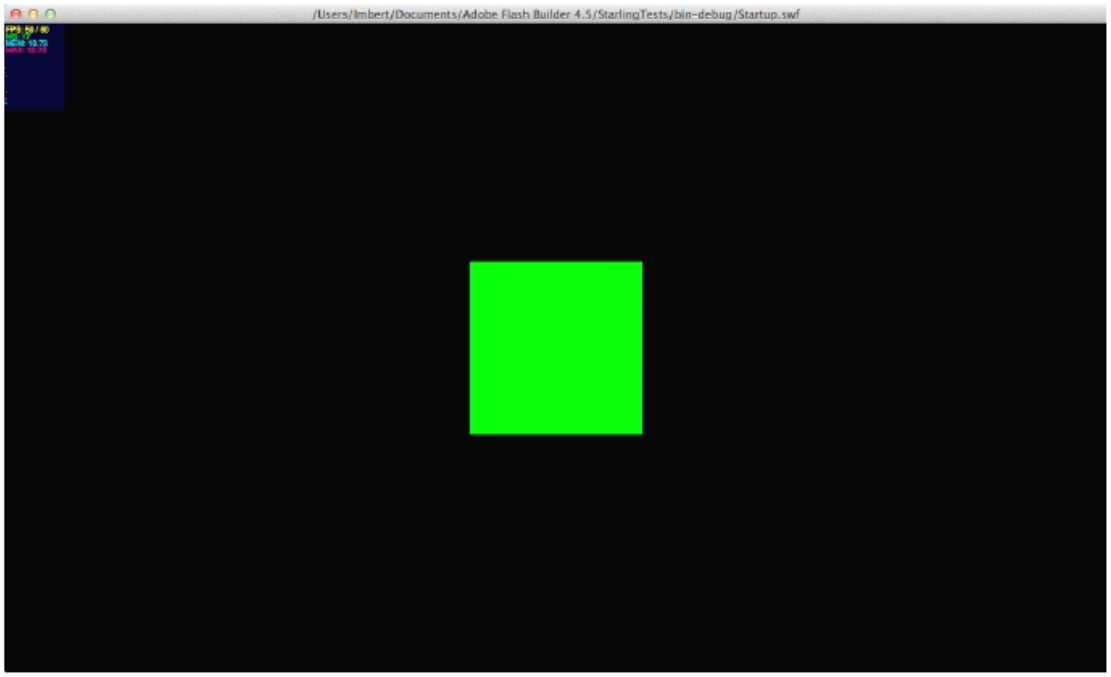

图 1.12　一个绿色的四边形

现在我们来利用 Event.ENTER_FRAME 事件对它来进行插值，从而在两个随机的颜色之间实现一个简单的 Easing 效果。

```
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;
        }
    }
}
```

然后我们可以使用 rotation 属性来旋转这个 Quad。注意：Starling 使用的单位是“弧度”（这也是与 Sparrow 一致的），而原生 Flash 使用的是“角度”。假如想使用“角度”来进行旋转的话，可以用到一个转换函数 starling.utils.deg2rad：

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

Quad 类继承自 DisplayObject，DisplayObject 是以注册点（Registration Point）为中心进行旋转的。而这个注册点，即 DisplayObject 的 pivotX 和 pivotY 属性，是可以在运行时改变的。

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

Quad 还可以与 TextField 一起被嵌入到一个 Sprite 里，这样 ActionScript 开发者就可以很自然地通过移动 Sprite 来同时移动这两个元素：

```
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;
 }
```

```
        }
    }
```

运行结果如下：

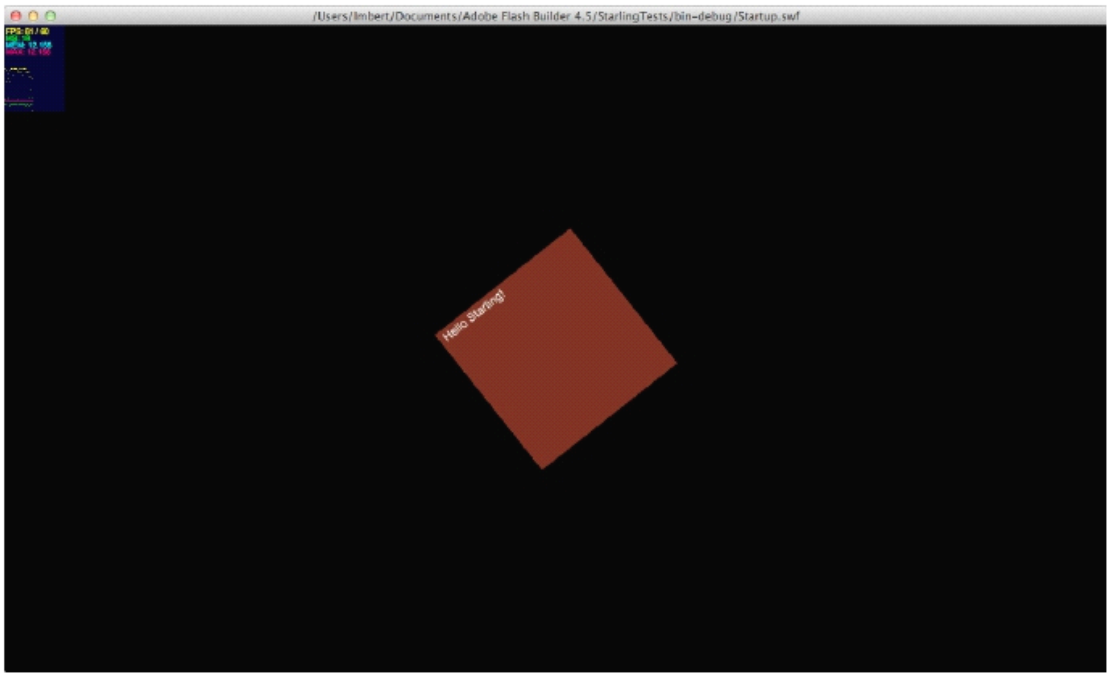

图 1.13 一个 Sprite 包含一个 Quad 和 TextField

上面的代码写得比较随意，现在让我们自定义一个类 CustomSprite 来对其进行封装，从而使 Game 类显得更加简洁。下面是 CustomSprite 类：

```
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;
        }
    }
}
```

下面是新的 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();
         }
    }
}
```

下面我们将实现更多的交互，让这个四边形跟随鼠标移动：

```
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;
        }
    }
}
```

在这里我们没有使用任何鼠标相关的方法，实事上在 Starling 里也没有鼠标的概念。后面我们将很快介绍到这一点。

我们实际上是通过监听 TouchEvent.TOUCH 事件来监听鼠标和手指的移动的，它等同于原生 Flash 的 MouseEvent.MOUSE_MOVE 事件。在每一帧，我们都可以通过使用 TouchEvent 对象提供的方法（比如 getTouch 和 getLocation）来获取当前的鼠标位置。一旦鼠标位置被获取后，我们在 onFrame 处理函数里使用一个简单的 Easing 方程来四处移动我们的四边形。

Starling 不只使我们对 GPU 的编程变得简单，也使我们能很容易地对资源进行释放。如果我们需要在点击四边形的时候将它从场景中移除，可以这样写：

```
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);
            }
        }
    }
}
```

注意上面的例子中，我们将显示对象从场景移除了，但却没有移除对 TouchEvent.TOUCH 事件的监听。我们可以用 hasEventListener 这个方法来测试该事件是否被移除：

```
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) );
    }
}
```

那么，为了安全地移除一个对象，应该使用 removeChild 这个方法的第 2 个参数 dispose。使用这个参数可以让我们在将一个对象从场景移除的同时，自动地移除所有的事件监听。

```
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) );
        }
    }
```

如果这个显示对象还有子级对象，所有的这些子级对象也会被释放。其它方法，比如 removeChildren 或者 removeChildAt 也有这个参数。注意，使用 dispose 参数也会清除对象在 GPU 里所占的显存，不过不会清除与其关联的纹理。为了把纹理也释放，你要在 Texture 或者 TextureAtlas 对象上调用 dispose 方法。

你也可以在任何 DisplayObject 上直接地调用 dispose 方法来移除所有的监听。

```
clicked.dispose()
```

下面我们将了解 Starling 的事件模型，它非常像原生 Flash 的事件模型。让我们花一些时间从 EventDispatcher 开始来了解 Starling 的事件模型吧。

11、 事件模型（Event Model）

如图 1.2 所示，所有 Starling 对象都是 EventDispatcher 的子类。像原生的 EventDispatcher 一样，所有 Starling 对象都提供如下方法：

- **addEventListener** :同原生EventDispatcher；
- **hasEventListener**： 同原生EventDispatcher；
- **removeEventListener**： 同原生EventDispatcher；
- **removeEventListeners**： 移除指定事件的所有侦听；

请注意，Starling 提供了一个非常有用的方法：removeEventListeners。在很多时候，当我们需要移除所有对某个事件的监听者时，我们可以将该事件的类型传给 removeEventListeners：

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

并且当你需要移除所有事件的所有监听者时，则可以参照如下示例：

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

上面我们使用过的 removeChild，它带有一个参数 dispose 来释放所有的事件监听者，它的内部实现就是调用上面的方法。

事件传递（Event Propagation）

到目前为止，我们已经得知 Starling 重建了一套完整的基于 Stage3D 的显示列表。而现在，我们又知道 Starling 还重建了一套完整的事件传递机制。

假如你不熟悉事件传递的概念，可以在如下网页上获取详细的教程。

Starling 所实现的事件传递，在一些细节上与原生的有一些有趣的差异。Starling 只支持事件传递过程中的冒泡阶段（bubbling phase），而没有捕获阶段（capture phase）。在下面章节的例子里，我们可以学到事件传递机制是如何运作的。

Touch 事件

正如我们前面所提到的，Starling 是 Sparrow（Sparrow 是一套用于 IOS 游戏开发的基础框架）的孪生兄弟。因此，Starling 的触摸事件机制非常适合移动应用开发。也正因为如此，当我们第一次用 Starling 在桌面应用程序中实现鼠标交互功能时，会感觉有些怪异（因为在 Starling 里没有鼠标事件）。

首先，如果你看过图 1.2，就会注意到，相对于原生的显示列表，在 Starling 的承继关系里看不到 `InteractiveObject` 类，因此所有的显示对象都默认具有交互功能。也就是说 DisplayObject 是定义了交互行为的。

在前面的例子里，我们已经简单地使用过了触摸事件：

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

你也许会认为这个事件的功能有限？而实际上，它的功能非常强大。你可以在事件中检测到许多不同的状态。任何时候，只要鼠标或者手指与一个图形对象发生了交互，一个 TouchEvent.TOUCH 事件就会抛出。

让我们仔细看看，在下面的代码中，我们 Trace 出 Touch 对象上的 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;
}
```

当我们开始与四边形进行交互时，我们能看到不同的阶段被触发。下面列出了在 TouchEvent

里的所有阶段，它们以常量的形式定义在 TouchPhase 类里。

- **began**：鼠标或者手指开始交互时（类似于鼠标按下状态）；
- **ended**：鼠标或者手指停止交互时（类似于鼠标点击状态）；
- **hover**：鼠标或者手指从一个对象上移过时（类似于鼠标移过状态）；
- **moved**：鼠标或者手指正在移动一个对象时（类似于鼠标按下状态+移动状态）；
- **stationary**：鼠标或者手指停在一个对象上时；

除以上面的阶段信息，TouchEvent 还提供一些其它的方法：

- **ctrlKey**：标志Ctrl键是否按下；
- **getTouch**:获取鼠标从指定对象上移过时产生的Touch对象；.
- **getTouches**：获取鼠标从指定对象上移过时产生的Touch对象列表（用于多点触摸）；
- **shiftKey**: 标志Shift键是否按下；.
- **timestamp**：事件触发的时间（相对于程序开始时的秒数）；
- **touches**：得到同一时间发生的所有Touch对象；

使用 `shiftKey` 和 `ctrlKey` 属性可以方便地实现对组合键的检测。每产生一次交互，都会产生一个 Touch 对象。

下面让我们看看 Touch 对象的方法：

- **clone**：复制一个Touch对象；
- **getLocation**: 得到TouchEvent触发时相对当前对象的坐标；
- **getPreviousLocation**: 得到上一个TouchEvent触发时相对当前对象的坐标；
- **globalX**: 同原生的MouseEvent；
- **globalY**：同原生的MouseEvent；
- **id**: 一个独一无二的Touch对象编号；
- **phase**：TouchEvent所处的阶段；
- **previousGlobalX**：得到上一个TouchEvent触发时相对舞台的坐标的X值；
- **previousGlobalY**：得到上一个TouchEvent触发时相对舞台的坐标的Y值；
- **tapCount**：手指快速触碰显示对象的次数。可以用于检测手指的双击；
- **target**：触发TouchEvent的显示对象；
- **timestamp**：事件触发的时间（相对于程序开始时的秒数）；

模拟多点触摸（multi-touch）

当我们在为移动设备开发应用时，经常需要处理多点触摸的交互，比如缩放操作。当我们在 PC 上进行开发时，无法像在移动设备上那样去测试多点触摸，于是 Starling 内建了一套模拟多点触摸的机制。

你可以使用 Starling 对象 simulateMultiTouch 属性来开启这个功能。

```
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();
        }
    }
}
```

一旦开启这个功能，便可以使用 Ctrl 键来以模拟多点触摸的输入。运行结果如下（Starling 用两个小圆点来表示两只手指）：

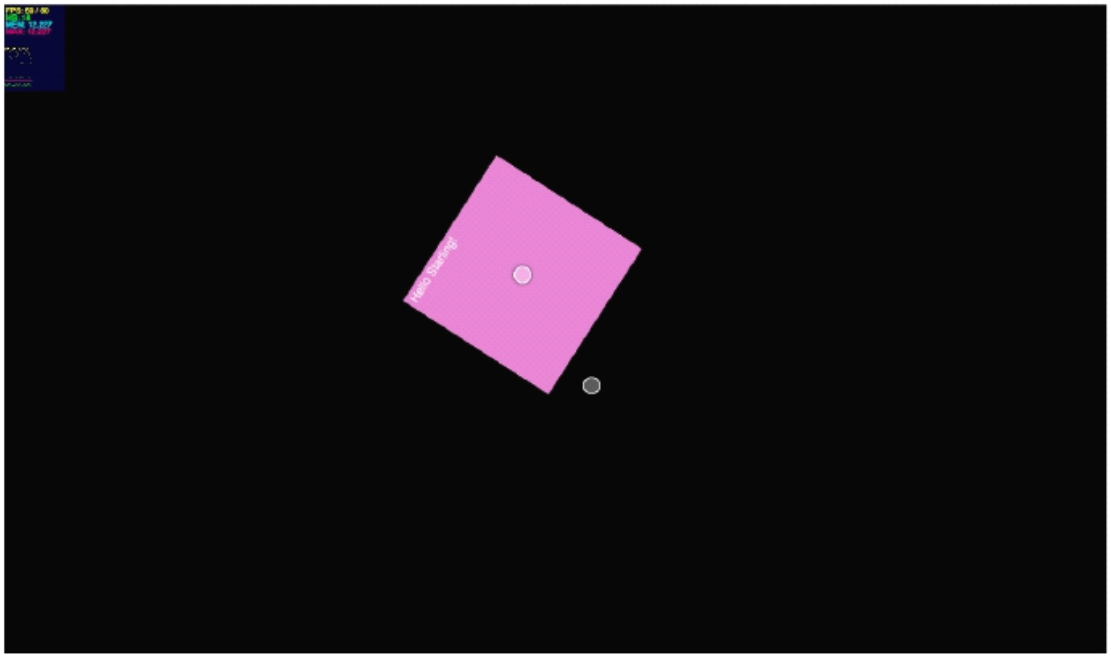

图 1.14 Multi-touch 模拟.

在下面的代码中，我们通过检测两个接触点，并计算它们之间的距离来缩放一个四边形。

```
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 );
               }
           }
        }
    }
}
```

下面我们将来学习 Starling 的纹理（Texture）。

12、 纹理

在 Starling 中，纹理是用来填充 Image 对象的。这很像原生 Bitmap 与 BitmapData 之间的关系。Texture 具有如下方法或者属性：

- **base**：Starling纹理对象所依赖的Stage3D纹理对象；
- **dispose**：销毁纹理的数据；
- **empty**：获取一个指定尺寸和颜色的空纹理，它是静态函数；
- **frame**：获取纹理的框架矩形；
- **fromBitmap**：根据指定的Bitmap对象生成纹理；
- **fromBitmapData**：根据指定的BitmapData对象生成纹理；
- **fromAtfData**：使用一个ATF（Adobe Texture Format）文件来生成被压缩过的纹理，压缩过的纹理可以节省大量内存，特别适用于像移动设备这样的特殊环境；
- **fromTexture**：根据指定的纹理对象生成一个新的纹理；
- **height**：纹理的高；
- **mipMapping**：标志该纹理是否包含MipMaps；
- **premultipliedAlpha**：标志该纹理的Alpha值是否已经被乘入到RGB值里；
- **repeat**：标志该纹理是否支持平铺和拉伸；
- **width**：纹理的宽；

纹理可以支持不同的素材格式，下面列出常用的一些格式：

- **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格式也可以被使用。但有一点需要注意，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；
- **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。它主要使用了两种相似的技术来实现它的有损压缩：JPEG-XR1压缩技术和基于块的压缩技术（简称块压缩技术）。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；而基于块的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；

MipMapping 是一项材质贴图的技术，是依据不同精度的要求，而使用不同版本的材质图样进行贴图。例如：当物体移近观察者时，程序会在物体表面贴上较精细、清晰度较高的材质图案，于是让物体呈现出更高层、更加真实的效果；而当物体远离观察者时，程序就会贴上较单纯、清晰度较低的材质图样，进而提升图形处理的整体效率。

Note that texture dimensions need to be of a power of two (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024,

> 2048), but do not require to be square. If you do not respect this rule, Starling will automatically find the nearest power of two for your image dimensions and create a texture of this size, which can result in a waste of memory. To make sure you optimize the memory used by textures, it is recommended to use texture atlases commonly known as sprite sheets. We will come back to this topic later on.

为了确保得到最好的显示质量，GPU 需要得到一张图片的所有级别的材质图样。这意味着需要从 1 到原始图片大小（这个大小需要是 2 的次方）的所有版本材质贴图。没有 Starling，你将需要使用 BitmapData.draw 和矩阵变换等技术来手动生成这些材质贴图（Miplevels）。

> It is a good practice to upload them for 2D content, this is going to make your content perform faster and look better (reduces aliasing) when elements are being scaled down.

幸运的是前面提到过，Starling 能够自动为你生成这些贴图。下面是示例代码：

```
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();
 }
```

如果使用 ATF（Adobe Texture Format）格式，你连上面的这些代码都不需要。ATF 文件格式已经包含了所有的 Miplevels。它们不是在运行时生成的，而是在之前就已经存在了，于是它可以为你节省运行时间。在处理大量纹理的情形下，它可以节省宝贵的运行时间，加快程序的初始化。

注意，在 Texture 对象上有一个 frame 属性（它返回一个 Rectangle 对象），这允许我们在将纹理赋给 Image 对象时，可以指定其在 Image 中的位置。比方说，你想使你的 Image 四周有一个边框，你可以将一张比 Image 小的纹理放在 Image 的中心：

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

既然已经说到 Image 对象，下面就让我们来看看它。

Image

在 Starling 里，一个 starling.display.Image 对象相当于原生的

flash.display.Bitmap 对象：

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

为了显示一个 Image 对象，你需要创建一个 Image 实例，并且将一个 Texture 对象传给它：

```
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，我们使用了 Texture 类的静态函数 frameBitmap 来生成 Texture 对象。

运行结果如下：

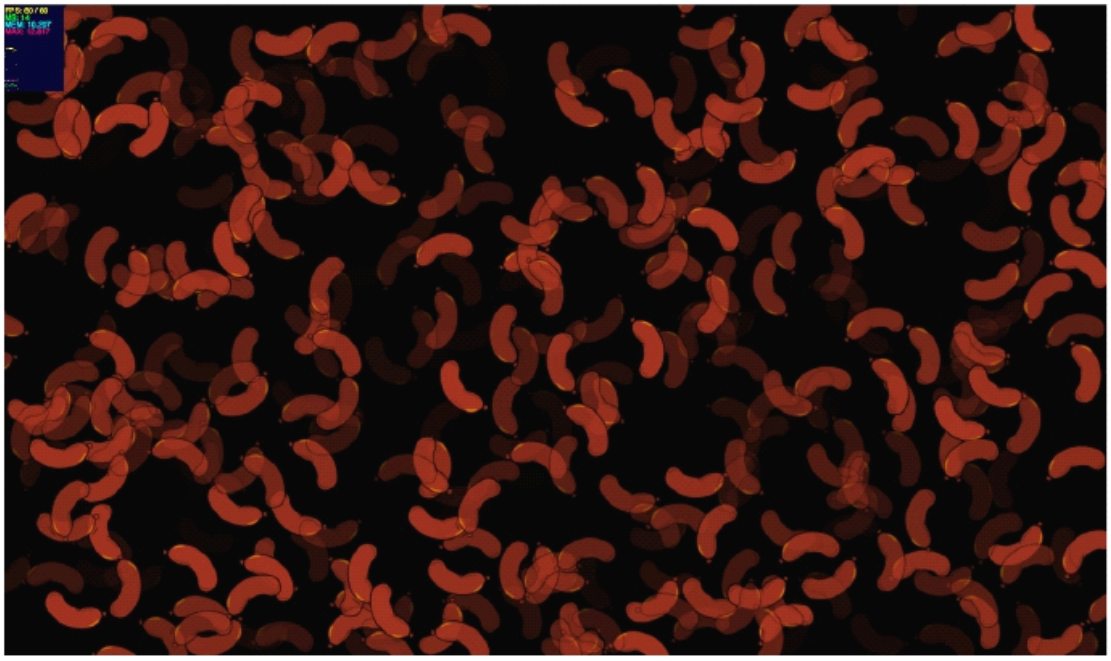

图 1.15 被随机放置的香肠

我们在上面使用了静态嵌入的位图来生成纹理，也可以用动态加载的位图生成纹理：

```
// 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 )
}
```

还有一些其它的 API 可以不用 BitmapData 也能生成纹理对象，在后面将会了解它们。

为了使屏幕上所有的 Sprite 使用相同的纹理，使用了如下代码，在每一次循环中都使用了纹理对象，但只需要在循环外创建一个纹理对象的实例：

```
// 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);
```

当然下面代码也可以实现上述效果，但是它会在每一个循环里都创建一个纹理对象，这样是非常不好的：

```
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()));
```

这将会为相同的纹理分配多份内存，同时也会对程序的性能产生不好的影响。因为多个相同的纹理被上传到 GPU，会产生多次性能消耗。最后，也会对循环本身的性能产生不良影响，因为每一个循环都会调用 fromBitmap 生成 MipMaps。

下面让我们来移动图像。让我们新建一个 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);
        }
    }
}
```

然后让我们在下面代码中使用 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.des tX ) < 1 && Math.abs
            ( sausage.y - sausage.destY ) < 1)
            {
                  sausage.destX = Math.random()*stage.stageWidth;
                  sausage.destY = Math.random()*stage.stageWidth;
                  sausage.rotation = deg2rad(Math.random()*360);
```

```
                    }
                }
            }
        }
    }
```

我们可以通过监听 Stage 的 TouchEvent.TOUCH 事件来轻松地捕获所有 Image 对象的移动事件：

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

我们测试这个事件的 target 和 currentTarget 属性，会看到正在派发这个事件的对象（currentTarget）是 Stage，而初始化这个事件的是 CustomImage 实例：

```
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 );
        }
    }
}
```

因此，我们不需要监听每一个 Image，只需要监听它们的容器。在事件的冒泡阶段，Stage 就是在这个容器里得到事件的。如果我们使用 TouchEvent 的 bubble 属性，可以看到事件正处于冒泡阶段：

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

这是 Image 的事件传递机制。让我们看看 Image 对象还提供了什么功能。正如你所料，Image 对象提供所有继承自 DisplayObject 的方法和属性，并提供了 Image 特有的用于处理图片平滑的 smoothing 属性。在 TextureSmoothing 类里定义了一些被 smoothing 属性用到的常量：

- **BILINEAR**：双线性平滑（默认值）；
- **NONE**：不平滑；
- **TRILINEAR**：三线性平滑；

用法如下：

```
//disable filtering when scaled
```

```
image.smoothing = TextureSmoothing.NONE;
```

下面图片，是进行放大并使用了双线性平滑（TextureSmoothing.BILINEAR）的效果：

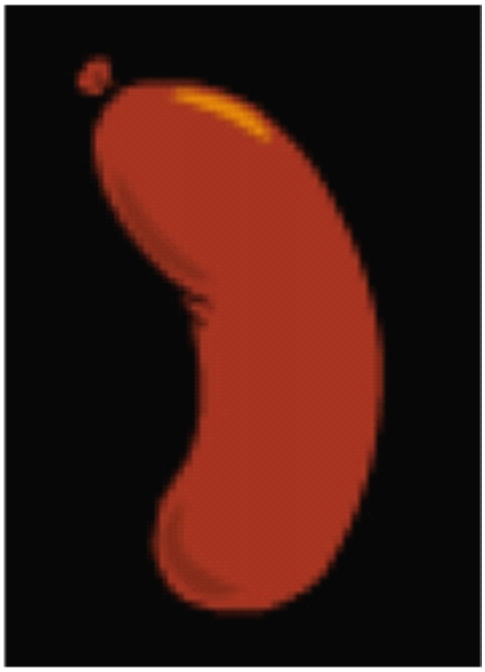

图 1.16 TextureSmoothing.BILINEAR.

下面图片，是进行放大并使用了三线性平滑（TextureSmoothing.TRILINEAR）的效果：

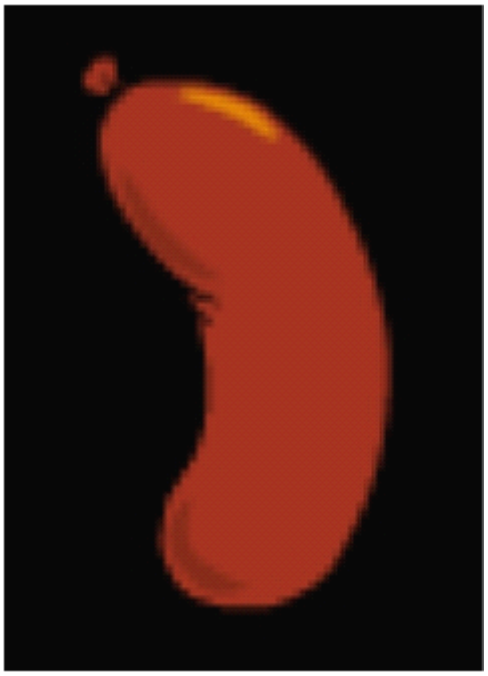

图 1.17 TextureSmoothing.TRILINEAR.

下面图片，是进行放大但没有使用任何平滑（TextureSmoothing.NONE）的效果：

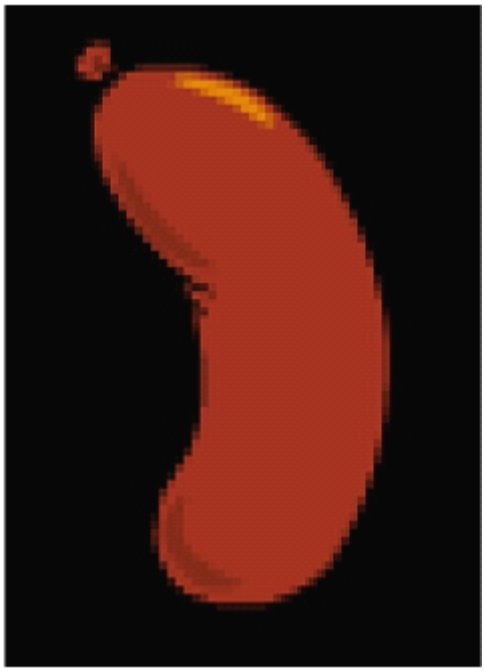

图 1.18 TextureSmoothing.NONE.

没有使用平滑与使用平滑的效果明显不同。

Image 对象还提供了 color 属性，允许你设置指定的颜色值。每一个像素的最终颜色，是 Image 的纹理颜色与你所指定的颜色值的乘积。这样可以很容易地为一张图片染上不同的颜色，而不需要额外使用不同的纹理。

从下面的图中，可以看出：

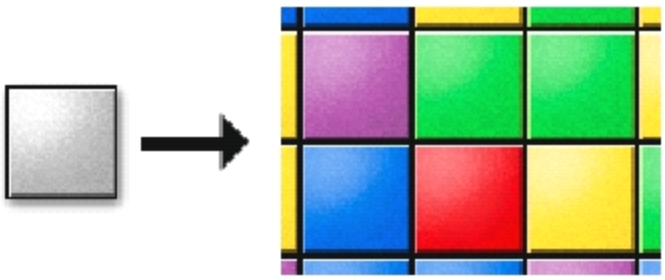

图 1.19 Texture tinted by the quad color (图片由 sparrow-framework.org 提供)

如何在 Starling 里使用动态的自定义的形状呢？也不是问题，在后面将会讲到。

碰撞检测（Collision Detection）

在大部分游戏里，如果不依赖专门的物理引擎（像 Box2D，在后面会讲到），你可能要自己处理简单的点击检测。如果是一个像圆这样的形状，只需要简单地测试 2 点之间的距离是否小于圆的半径即可。在一些其它的场合，我们也只是简单地做了两个包围盒之间的检测。那么，对于像素级别的完美的碰撞检测怎么实现呢？

下面图片描述了一个典型的情形，我们需要检测包含透明区域的两个图片之间的碰撞：

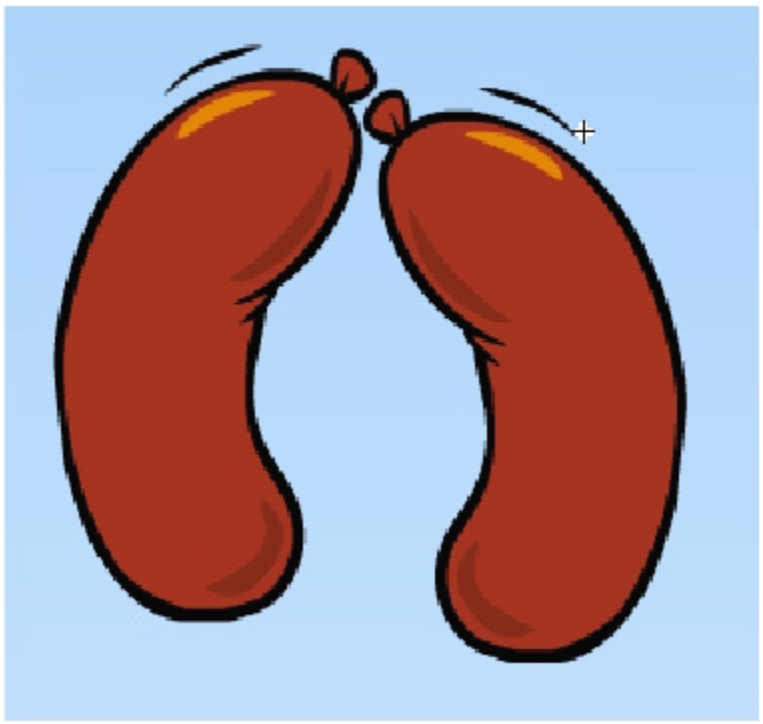

图 1.20 像素级别的碰撞

在这种有曲线轮廓的情形下，我们很想实现像素级别的碰撞检测。要实现这种像素级别的点击检测，我们要用到透明图像。

实际上，使用自己编写 ActionScript 代码来进行像素检测是相当耗费 CPU 的。但幸运的是 Starling 依赖原生的 BitmapData 来创建纹理，于是我们可以使用 BitmapData 对象上的 hitTest 函数来进行像素检测。

hitTest 函数可能看上去很复杂，使用起来却非常简单：

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

注意，secondObject 参数可以是一个 Point、Rectangle 或者 BitmapData 对象。这个方法在各种场合都非常有用。在下面的例子中，我们便用到了两个 BitmapData 对象：

```
if ( sausageBitmapData1.hitTest(new Point(sausageImage2.x, sausageImage2.y), 255,
sausageBitmapData2, new Point(sausageImage1.x, sausageImage1.y), 255))
{
   trace ("touched!")
}
```

我们首先传入一个 Point，它是进行碰撞检测的第 2 个对象的坐标。然后是 Alpha 掩码，可以指定一个特定的颜色值为透明的（大部分情况下，可以使用 255 或者 0XFF）。最后是第 2 个 Point 和 Alpha 掩码，分别是当前对象的坐标和 Alpha 掩码。

因为要在每一帧都进行点击检测，所以我们想尽可能使 GC 消耗最小。在之前的代码里，每一次点击测试，都会创建 Point 对象。而比较好的办法是，先创建好一个 Point 对象，然后在每一帧去设置它的 X 和 Y 值。

```
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, sausageBitmapData2, point1, 255))
    {
       trace("touched!");
    }
}
```

下面让我们看看怎么在 Starling 里使用原生的绘图方法。

绘图方法（Drawing API）

Starling 没有像原生的 flash.display.Graphics 对象一样拥有绘图方法。然而，很容易复用这些原生绘图方法在 BitmapData 上绘制图形，然后用于纹理。

比方说，你想要一个星形形状，并用 Starling 显示它。那么可以使用如下代码：

```
// 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);
```

以上的原理非常简单，使用原生 Graphics 提供的方法在 CPU 里绘制线条、笔触和填充，然后使之栅格化成位图并用在纹理中。

下面是另一个例子的示意图，在 Starling 中绘制圆形：

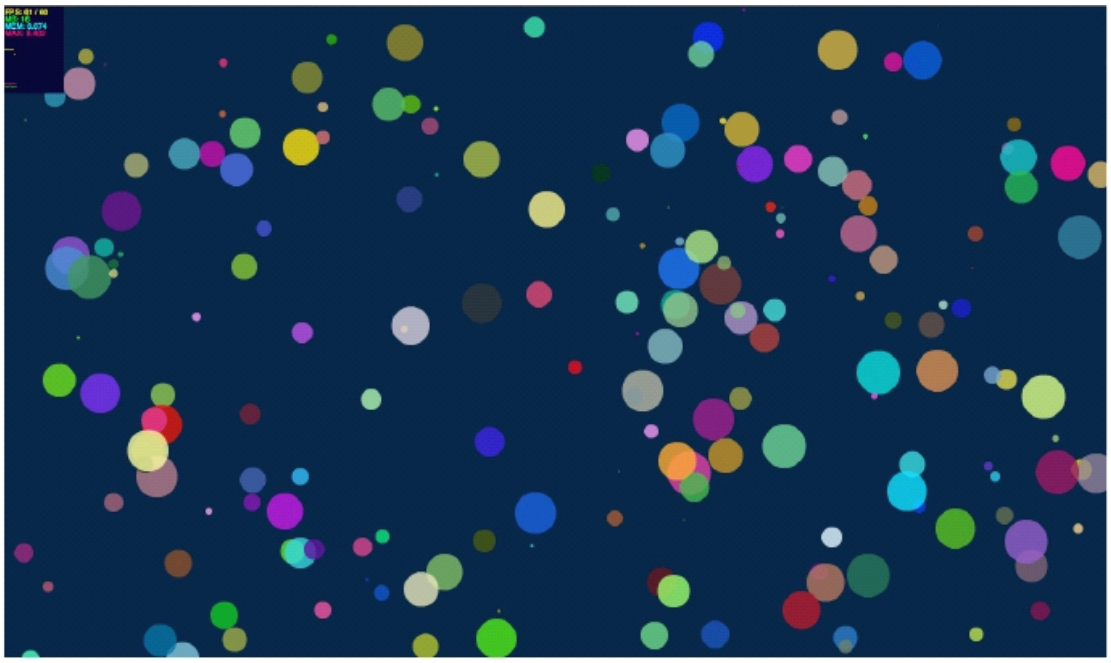

图 1.21 自定义的动态形状.

你了解过 Starling 中的 Flat Sprite 吗？那么让我们跟随下面的章节来了解怎样使用这个新的功能来提高性能。

13、 Flat Sprites

Starling 有一个非常强大的功能，被称为 Flat Sprites（在 Sparrow 里又被称为 Compiled Sprites），你可以用它来极大地提升程序的性能。

为了更好地理解显示列表是如何工作的，让我们来看一下下面这棵典型的显示列表树：

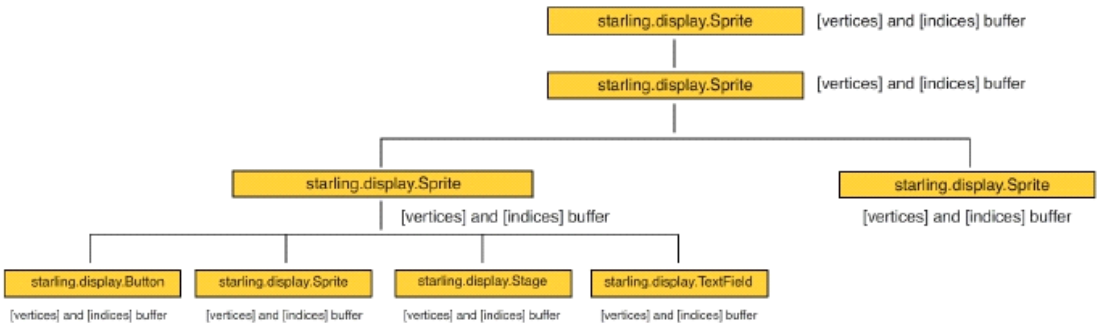

图 1.22 每个子级对象拥有各种的顶点和索引缓存

在上图中可以看出，Starling 为了处理每一个子级对象的顶点和索引缓存以及各自的行为，需要大量的计算，从而影响到性能。

那么，当所有子对象都使用同样的纹理时，Starling 会怎么做呢？它会将所有子对象的几何数据收集到一个简单的大的顶点和索引缓存中，并且在一次绘制调用中绘制所有的内容。

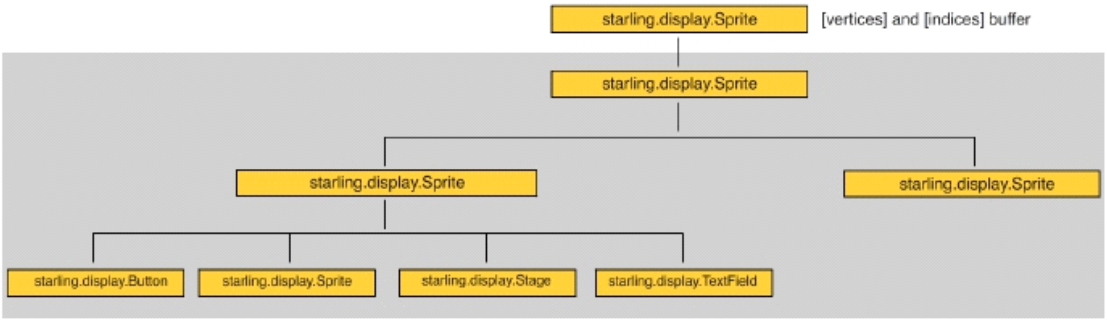

图 1.23 When flattened, children drawn in one draw call (one index/vertex buffer).

你可以认为它有点像原生显示列表所支持的一个方法： cacheAsBitmap。所不同的是（也是关键所在），当 Starling 显示列表树中的子级对象发生改变时，并不会自动进行重绘。你只有显式地调用了 flatten 方法后才能看到改变的结果。

下面是与之相关的方法：

- **flatten**: 用于尽可能快地渲染对象。一旦被调用，Starling会收集绘制显示列表树所需的所有几何数据，并将这些数据存储存于一个单一的缓存里，然后通过一次绘制调用绘制所有的内容。当然这个功能也是有限的，一旦被调用过了，新产生的变化不会被渲染，除非再次调用它；.
- **unflatten**: 关闭 flatten 功能；
- **isFlatenned**: 标志该Sprite对象当前是否处于flattened 状态；

让我们试试下面的代码，添加多个图片到一个 Sprite 里，然后逐帧旋转容器：

```
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，动画非常平滑地运行在 60FPS，但是我们还能继续优化，即通过调用 flatten 方法来减少对绘制接口的调用次数。

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

现在所有的子对象都在一次绘制调用中被绘制。你可能从 FPS 上看不到有什么区别，但是在 CPU 的占用率上会不同。做一些测试，就会看到 CPU 的占用率会有 10 倍或者更高的下降。

> Note that if the children do not share the same texture, Starling will split up the draw calls, in such scenario the benefit of the flatten behavior will be reduced.

这个功能在移动设备上也是有用的。当然，这个功能的价值还在于，它是动态的。你可以先调用 unflatten，然后执行相关的逻辑，最后再 flatten 。在任何时候，当一个子级对象被修改时，只需要显式地调用一次 flatten，修改的结果就会在屏幕上显示。

记住，flatten 方法只在 Sprite 上提供，在 MovieClip 上不支持这样的优化。将来可能会加入这个方法。

14、 影片剪辑（MovieClip）

我已经知道在 Starling 中如何使用 Sprite，但是怎么使用像 MovieClip 这样的影片剪辑呢？每一个 Flash 开发者都非常了解影片剪辑的概念。在过去的几年里，许多 AS3 开发者还自己构建了基于 BitmapData 的 MovieClip。

下图示例描述了 Starling 的 MovieClip 所用到的一份 SpriteAltas：

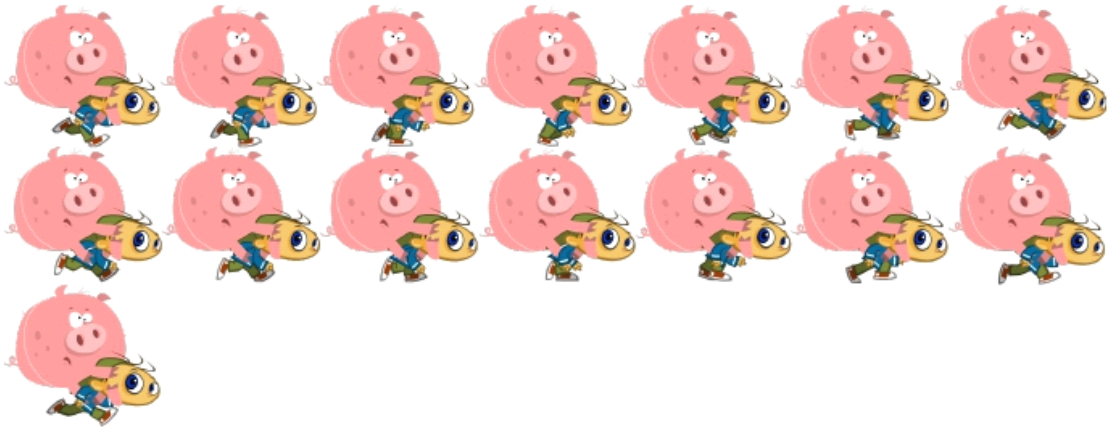

图 1.24 A sprite atlas for our running boy.

GPU 每一帧采样和显示其中 1 张贴图到屏幕上，通过这种方式，我们可以得到影片剪辑的效果。那么怎么实现它？Flash Pro 可以帮我们将动画的每一帧都导出，生成一个图片序列。然后，这些图片再被类似于 TexturePacker（http://www.texturepacker.com）的工具合入到 1 张大的纹理中。最后，该纹理被用于 MovieClip 对象。

下面图示描述了 SpriteAltas 的帧：

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

图 1.25 Frames in a sprite atlas.

如果你要将你的图集生成 MipMaps，需要确保图集中每一帧之间有 2 像素的空白。这样当 GPU 采样 1 帧的时候，不会将其它帧的像素也采样进来。

你要记住，纹理在大小上有一些限制，并且 Stage3D 在设计时也考虑到了移动设备，因此 Stage3D 会受到 OpenGL ES2 规范的严格限制，比如纹理的大小要是 2 的次方。TexturePacker 会帮你遵循这个规则，并集成一个 AutoSize 的功能，帮你确定纹理的最佳宽度和高度，还可以限制最大宽高。（Starling 的限制是 2048*2048）

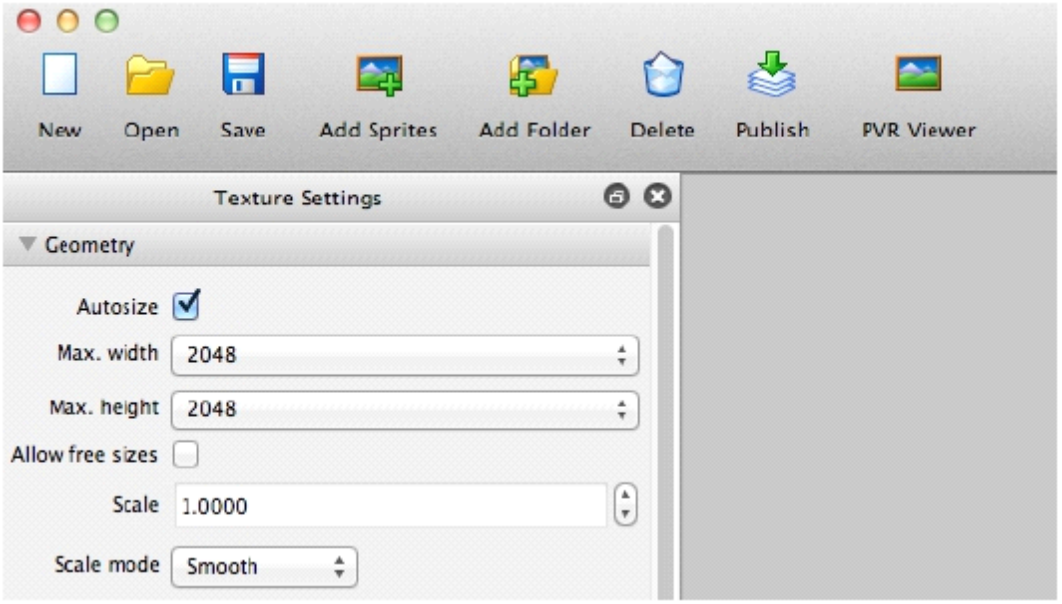

图 1.26 Autosize feature in TexturePacker.

前面提到过，Starling 将自动确保你使用的纹理是 2 的乘方大小。如果不是，Starling 也会自动为你修剪成最接近的大小。

为了让 Starling 知道怎样定位纹理中的每一帧，需要提供一个 XML 文件给 TextureAtlas 类，它描述了一份基于纹理的坐标列表。而 TexturePacker 会为我们自动生成这个列表，你可以选择不同的导出格式：XML、JSON 等等。

Starling 支持原始的 XML，下面是一份 XML 的示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<TextureAtlas imagePath="running-sheet.png">
    <!-- Created with TexturePacker -->
    <!-- http://texturepacker.com -->
    <!-- $TexturePacker:SmartUpdate:2b3f5fa2588769393bcea9b6327498826$ -->
    <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>
```

我们希望对这些帧进行全面控制，允许每一个影片剪辑都有自己独立的帧率。于是 Starling 提供了这个功能。请看下面 MovieClip 的构造函数：

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

在下面代码中，使用影片帧序列位图来创建一个纹理：

```
[Embed(source = "../media/textures/running-sheet.png")]
private const SpriteSheet:Class;

var bitmap:Bitmap = new SpriteSheet();

var texture:Texture = Texture.fromBitmap(bitmap);
```

然后我们获取 XML，它描述了每一帧在纹理中的坐标。

```
[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);
```

这时我们便可以获取运行时每一帧对应的纹理的列表：

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

我们传入参数“running”是因为我们只需要与它相关的帧纹理序列，我们也可以获取其它（比如“jump”、“fire”等）相关的帧纹理序列。

让我们看看完整的代码：

```
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 );
             }
        }
    }
```

运行后，得到如下结果：

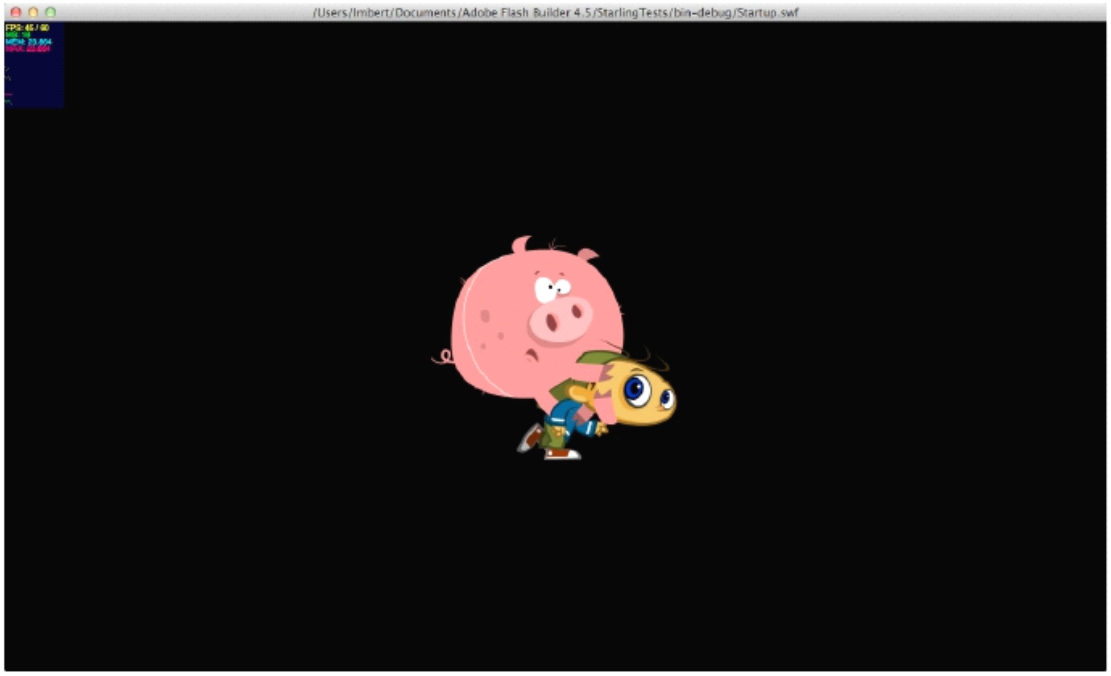

图 1.27 Our running boy rendered.

稍后，我们将介绍一个简单的缓存技术，用来重用资源。这样就可以在程序的生命周期内控制对象的个数。

如果想要翻转我们的影片，只需要使用 scaleX 属性就可以：

```
mMovie = new MovieClip(frames, 40);

mMovie.scaleX = -1;

mMovie.x = (stage.stageWidth - mMovie.width >> 1) + mMovie.width;
mMovie.y = stage.stageHeight - mMovie.height >> 1;
```

然后得到如下结果：

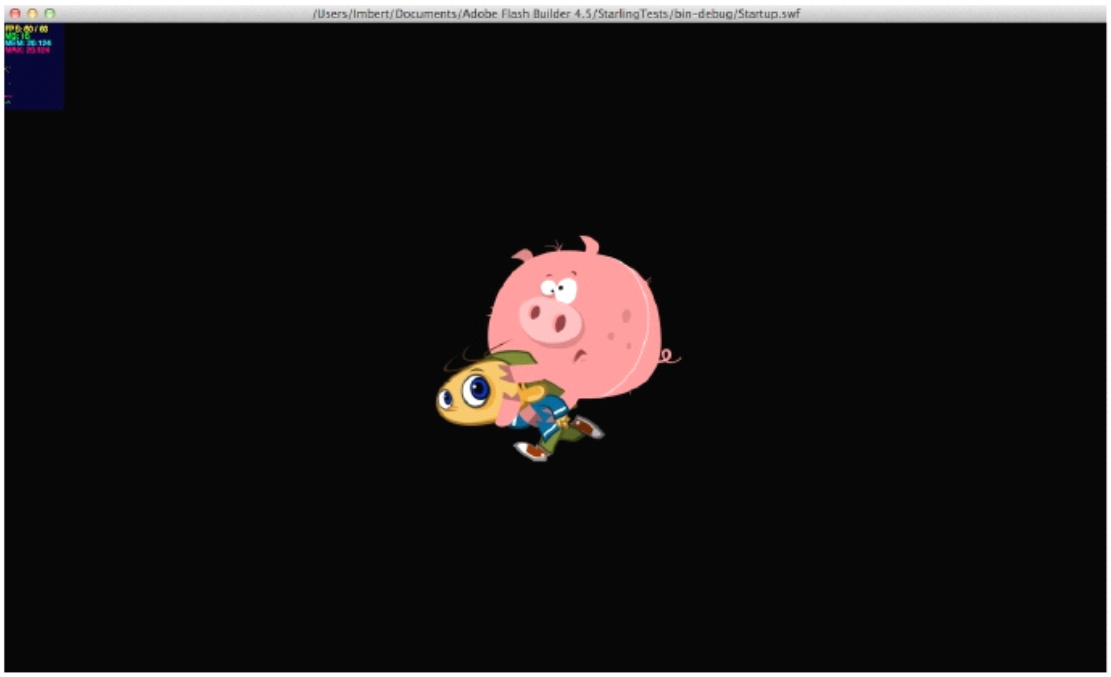

图 1.28 被翻转的 MovieClip

如前面所述，我们通常将多个资源放到同一个纹理文件里。为什么这样做？

首先很方便，游戏中的所有资源都放在同一个纹理文件里，使传递给 GPU 的纹理数变得最小。记住，传递纹理给 GPU 是非常耗的一件事，特别对于移动设备来说。所以，越少次数向 GPU 传递数据越好。另外，从一个纹理切换到另一个也是非常耗费的，所以将所有的资源放到一个纹理里比分别放在不同的纹理里要好。

纹理贴图集（Texture Altas）

在前面，我们只是介绍了 SpriteAltas 的概念，现在，我将介绍 TextureAltas。在一个 TextureAltas

里，所有的资源都包含在一个纹理贴图里。

在下图中，我们将另一个帧序列贴图添加到一张已经存在的贴图里。

图 1.29 Texture Atlas containing all our assets.

我们的 XML 文件现在也包含了一个名为 butcher（屠夫）的贴图。

```
<?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"/>
```

我们现在可以在 TextureAltas 对象的 getTextures 方法里引用这些帧：

```
// 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 );
```

运行的结果是，你会看到一个屠夫在小男孩边上：

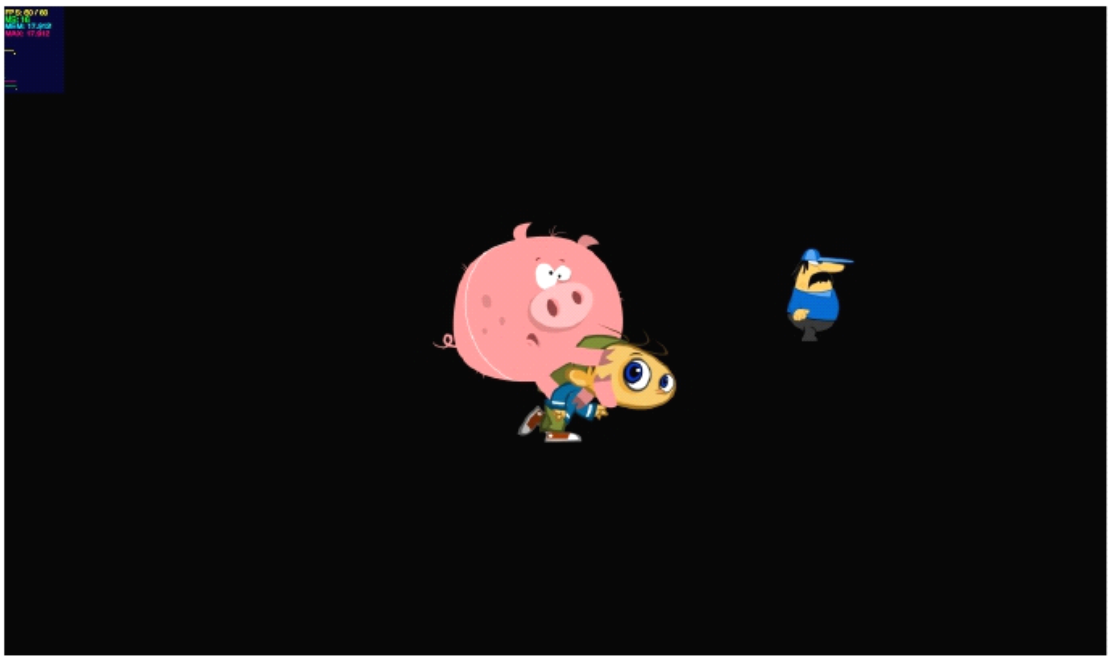

图 1.30 Both MovieClips sampled from the same sprite sheet (texture atlas).

到目前为止，我们已经创建了一个影片剪辑在屏幕上，但是他们没有播放。为了播放他们，我们需要使用一个 Juggler 对象。

Starling 对象提供一个默认的 Juggler 对象，用来处理你的动画。为了使上面的男孩和屠夫动起来，我们要这样写：

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

然后，我们的影片剪辑就可以动起来了。当然，我们可以随时暂停或者停止它。

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

要注意这两者之间的细微区别。pause 是用来暂停当前动画，使动画停留在当前所播放的帧，而 stop 会使动画回到第 1 帧。

下面看看 MovieClip 的所有方法，其中一些对于 Flash 开发者来说很眼熟，另一些则非常有用，比如设置帧率、运行时替换或者添加新的帧等等：

- **currentFrame**：同原生MovieClip；
- **fps**：默认帧率。当在MovieClip中增加1帧却未指定帧的时长时，将使用它来计算默认的时长；
- **isPlaying**：标志影片是否正在播放；
- **loop**：标志影片是否循环播放；
- **numFrames**：得到影片剪辑的帧数；
- **totalTime**：得到所有帧的累积时间；
- **addFrame**：增加1帧，并指定其时长；
- **addFrameAt**：在指定帧处插入1帧；
- **getFrameDuration**：得到指定帧的时长（秒）；
- **getFrameSound**：得到指定帧所关联的声音对象；
- **getFrameTexture**：得到指定帧的纹理；
- **pause**：同原生MovieClip；
- **play**：同原生MovieClip，需要同时确保它被添加进1个Juggler；
- **removeFrameAt**：移除指定帧；
- **setFrameDuration**：设置指定帧的时长（秒）；
- **setFrameSound**：设置指定帧的声音；
- **setFrameTexture**：设置指定帧的纹理；

我们不会全部介绍它们，只介绍一些非常有用的方法。比如 addFrameAt 和 removeFrameAt，它们允许你在运行时添加或者删除帧。或者 setFrameDuration，它可以让你为某一帧指定特

定的时长。当然还有其它的一些辅助方法，像 isPlaying 和 loop。

在下面的代码中，我们想让第 5 帧播放 2 秒的时间。

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

我们也可以动态地为某一帧添加声音效果：

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

多亏了这些方法，可以让我们通过动态加载或者嵌入资源，在运行时生成影片剪辑。这是一个非常强大的功能。

下面，让我们用键盘来控制这个小男孩：

```
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;
        }
    }
}
```

如果你需要监听动画的结束，可以监听 Event.COMPLETE 事件。

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

我们在这个例子中正好使用了 Juggler 对象，下面让我们来看看它是怎么工作的，以及它还能做什么？

15、 动画系统（Juggler）

Juggler 类允许你使任何实现了 IAnimatable 接口的对象动起来。MovieClip 对象就实现了这个接口，但是你也能为 Starling 定义自己的动画类，所有你要做的只是实现这个 IAnimatable 接口并重写 advanceTime 方法。在教程的最后，我们构建粒子系统时，将会定义自己的动画类。

下面我们可以看看在 MovieClip 里实现动画播放的主要代码。在每一帧，纹理都会被切换。它的原理很像在 Bitmap 对象里每一帧去改变它的 bitmapData 属性：

```
// 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);
}
```

下面是 Juggler 所有可用的 API 列表：

- **add**：添加一个对象到Juggler里；
- **advanceTime**：API intended to be called if needed to manually handle the Juggler main loop.
- **delayCall**：延时调用一个指定的方法，返回一个实际调用该方法的代理对象；

- **elapsedTime**：该Juggler累积运行的全部时长；
- **purge**：一次性移除所有对象；
- **remove**：从Juggler里移除一个对象；.
- **removeTweens**：从Juggler里移除与指定对象关联的所有Tween对象（详见源码）；

Juggler 有一个有趣的功能，就是支持延时调用。在下面的代码里，我们使用这个功能来延时将一个对象从它的父级对象里移除：

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

当游戏主体逻辑被暂停时，你可能需要创建另一个 Juggler 对象来独立播放某一个动画。比如，当游戏被暂停时，你需要表现菜单动画效果。这个时候，你需要做的就是创建一个新的 Juggler 对象，与菜单动画关联。

这样，你需要将游戏设计为每一个模块都有自己的 Juggler。当我们需要暂停时，不需要暂停整个游戏，只需要调用与当前模块相关联的 Juggler 的 stop 函数：

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

相反，游戏中每一个主要模块（菜单，背景，操作区）将被独立的 Juggler 处理。当游戏需要暂停，只需要指定的 Jugglers 被暂停，这允许你控制游戏中每一个模块的暂停和恢复。

在下面代码中，我们定义了一个 BattleScene 类，用来实现我们的战斗场景。

```
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();
        }
    }
}
```

在这个类的外边，我们在 EnterFrameEvent.EVENT 事件的处理函数里调用 advanceTime 函数，

并传入这一帧所流逝的时间。

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

当游戏被暂停时，我们停止调用 advanceTime 函数。当然，我们希望菜单能够被显示，菜单上面动画效果能够被播放，只需要加上：

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

    dashboard.advanceTime ( event.passedTime );
}
```

上面代码中只是根据 paused 的值切换了一下 advanceTime 的调用，就可以实现上述功能。

让我们再来看看在 Starling 实现交互的另一个重要的功能：Button。

16、 Button

Starling 支持按钮功能。下面是 Button 类的构造函数：

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

默认情况下，Button 类创建一个内部的 TextField 来实现按钮的标签，并使之在按钮内居中对齐。在下面代码中，我们创建一个简单的使用了位图皮肤的按钮。

```
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);
            }

        }
    }
```

在这里我们用 fromBitmap 函数来创建用于按钮皮肤的纹理：

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

让我们来创建一个简单的菜单：

```
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);
            }
        }
    }
```

下面是运行结果：

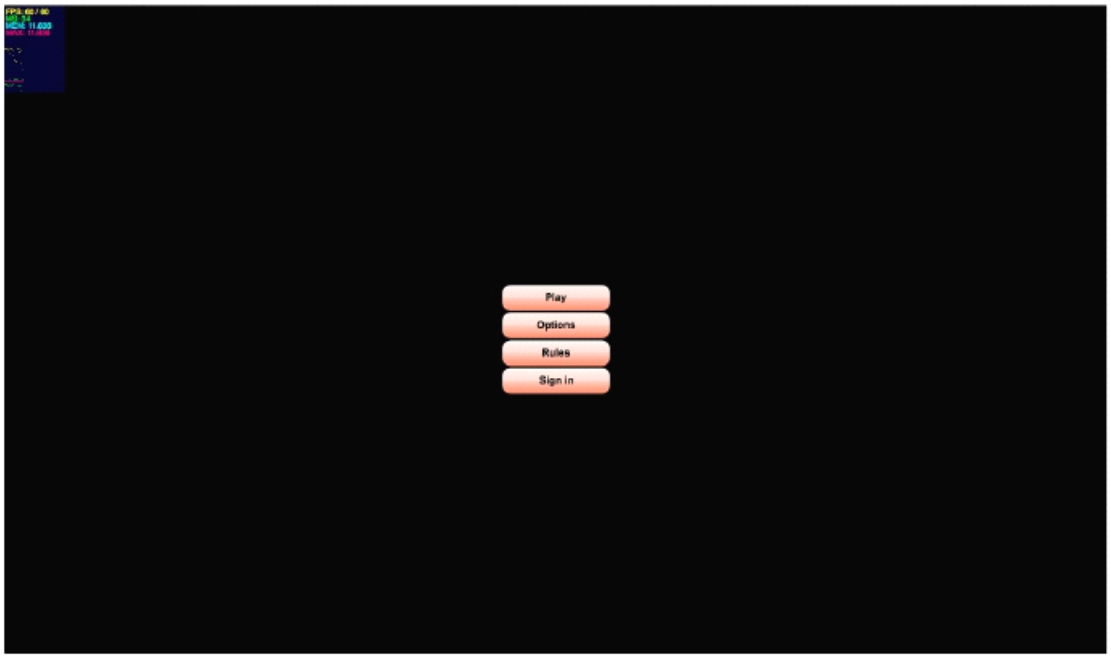

图 1.31 A simple menu, made out of buttons.

在这里，我们没有使用 SpriteSheet。我们只是简单地使用了 Texture 对象。如果你只打算使用单一的皮肤，这种方法是可以的。最好的办法是将你所有的皮肤都放到一个 TextureAtlas 里。

下面来看看 Button 对象提供的属性：

- **alphaWhenDisabled**：当按钮处于Disabled状态时的Alpha值；
- **downState**：当按钮处于按下状态时的纹理；
- **enabled**：决定按钮是否能被点击；
- **fontBold**：决定按钮标签的字体是否加粗；
- **fontColor**：字体的颜色；
- **fontName**：按钮标签的字体名，可以是系统字体或者被注册的位图字体；
- **fontSize**:按钮标签的字体大小；
- **scaleWhenDown**：当按钮被触摸时的缩放倍数，如果downState属性有效，则按钮不会被缩放；
- **text**：按钮的标签文本；
- **textBounds**：按钮的标签包围盒；
- **upState**：按钮未被按下时纹理；

与原生 Flash 的相反，Button 对象是 DisplayObjectContainer 的子类。意味着，你可以有很多种其它的方式来美化它。

Button 对象也会派发一个特别的事件：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 事件会往上冒泡，利用这个特性，你可以在它的窗口上监听到事件：

```
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!");
            }
      }
}
```

下面给菜单加一些背景：

```
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;
          }
     }
}
```

我们现在使背景具有卷动效果，下面是运行结果：

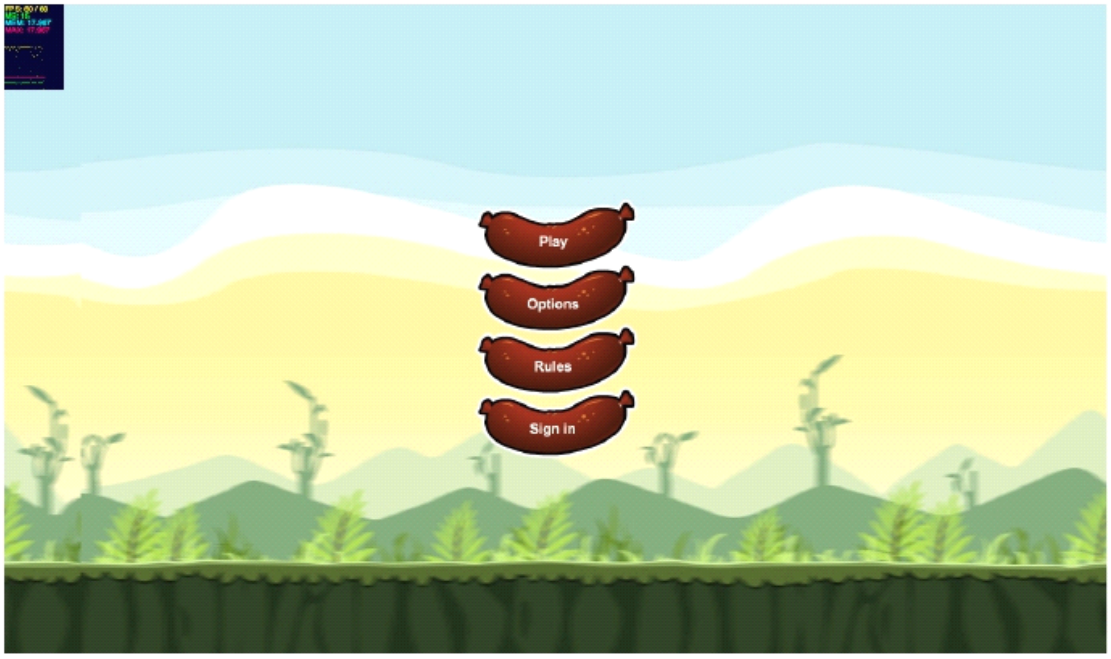

图 1.32 Our menu and a scrolled background.

我们在 PhotoShop 里让背景有一点点模糊，这是为了让卷动的效果更加明显。

17、 TextField

在教程一开始，我们在一个四边形里简单地使用过 starling.text.TextField 类。让我们花一点时间来介绍它在 Starling 里是怎么工作的。你可能不知道 GPU 怎么去渲染一个字体，这里有一个诀窍。Starling 创建一个原生的 TextField 对象，让它在屏幕后面用 CPU 渲染字体。让渲染后的字体栅格化，再上传给 GPU，从而可以显示出来。

在下面代码中，我们创建一个 TextField 对象，使用系统字体 Verdana 显示文本：

```
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);
            }
        }
    }
```

下面是运行结果：

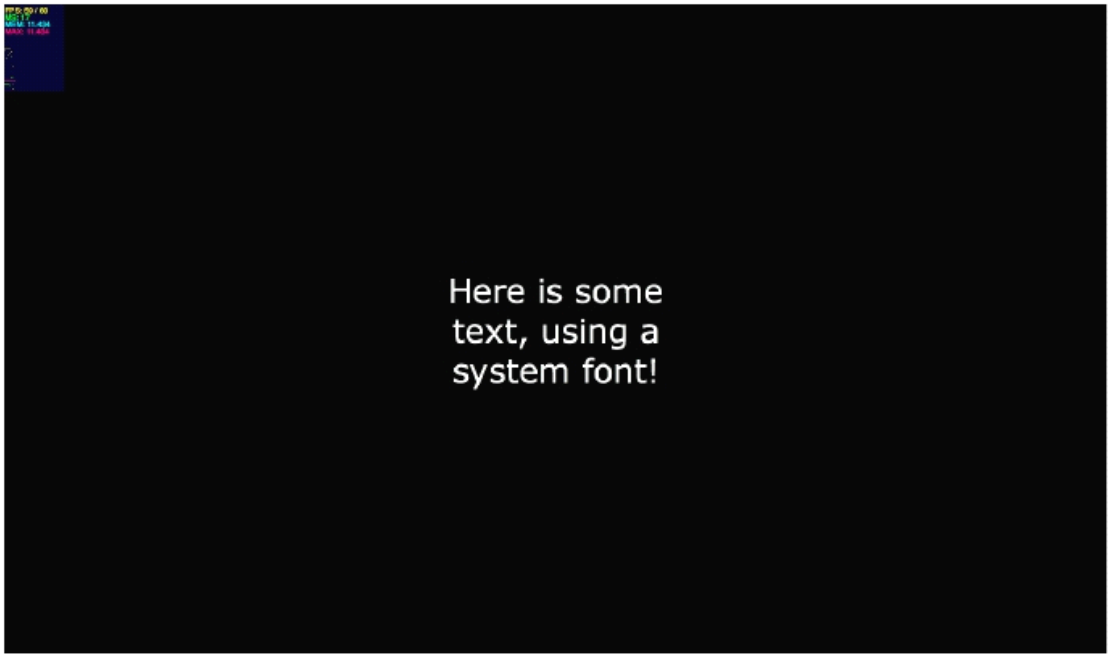

图 1.33 Some simple text.

记住，你在屏幕上看到的并不是一个真正的文本框，只是文本框的一个快照，它被绘制成一张位图纹理，并上传到 GPU 显示出来。

让我们仔细看看 TextField 类的属性：

- **alpha**：同原生TextField；
- **autoScale**：类似于原生TextField的autoSize；
- **bold**：决定是否使用粗体；
- **border**：是否显示边框，在调试时非常有用；
- **bounds**:文本框的包围盒；
- **color**：同原生TextField；
- **fontName**：字体名；
- **fontSize**：字体大小；
- **hAlign**：文本的水平对齐；
- **italic**：决定是否使用斜体；

- **kerning**：当使用位图字体时，是否使用字距调整信息，默认为true；
- **text**：同原生TextField；
- **textBounds**：文本框里字符的实际包围盒；
- **underline**：决定是否使用下划线；
- **vAlign**：文本的垂直对齐；

最 Cool 的属性之一是 autoScale，下面很快会讲到。让我们在下面代码中先来了解一下这些属性。在下面代码中，我们在文本四周添加一个边框，把字体加粗，并且改变它的颜色。

```
// 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;
```

结果如下：

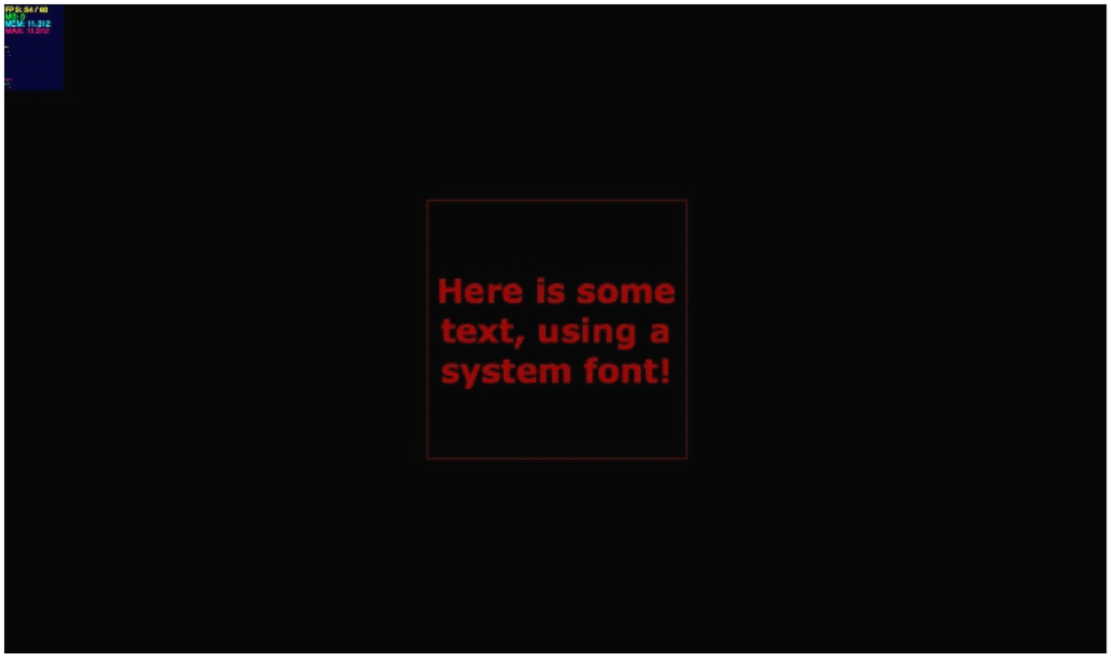

图 1.34 一些简单的有颜色的文本.

注意 TextField 不支持 HTML 格式的文本。将来可能会支持这个功能。如果你想要这个功能，也可以在 Starling 论坛上提出来。记住 Starling 是开源的，你也能自己实现这样一个功能，把它合入到 Starling 里去。

上面讲到怎么使用系统字体，这是非常有用的。但是在很多项目里，你会需要使用嵌入字体。在游戏开发中，你为了提高用户体验，很有可能需要使用嵌入字体。

嵌入字体（Embedded Fonts）

Starling 没有提供像原生 TextField 那样的 embedFonts 属性。但是不要担心，在 Starling 里非常容易嵌入字体。你所需要做的就是嵌入字体或者在运行时加载字体，然后将字体的名字传入给 TextField 的构造函数。

在下面代码中，我们来嵌入一个字体：

```
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 对象会通过字体名自动找到嵌入的字体，然后使用它。下图显示我们使用嵌入的 Abduction 字体的效果：

图 1.35 一些简单的使用了嵌入字体的文本

非常简单对不对？但是在你的程序中，你可能需要输入文本，比如输入名字、Email 和其它信息。你能想像得出，在 GPU 上进行文本编辑是一件多么棘手的事。大部分平台，包括移动平台，都会遇到这个问题。有一个简单的办法可以解决它，就是使用原生的显示列表。

在之前我们介绍过当 wmode 的设置不正确时，Starling 如何显示一条错误信息。Starling 是依赖其原生的叠加功能（nativeOverlay）来实现这一点的。Starling 工作在 Stage3D 的顶层，原生叠加功能允许你从 Starling 对象里获取显示列表，并且将这些对象添加到原生的显示列表中。这些对象包括：Video 或者可输入的 TextField。

当 Starling 检测到错误的 wmode 设置后，内部的 showFatalError 函数会在 Stage3D 的顶层显示警告信息。在这种场合下，无法使用 GPU 来渲染这些信息，便使用原生显示列表来显示信息。

```
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);
}
```

在下面代码中，我们创建一个输入 TextField 对象，把它叠加到 Starling 之上：

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

这个功能在需要输入文本的时候特别有用。注意，我们也可以在任何时候从 Starling 的 nativeStage 属性获取到原生的 stage 对象。

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

为了让我们的 Starling 达到最佳性能，让我们来看看位图字体。

位图字体（Bitmap Fonts）

为了在资源释放和 GC 的时候，达到最佳的性能和最少的消耗，我们可以使用基于位图的 TextField。原理是一样的，所有的字形信息都会被导出到 SpriteSheet，然后被用来渲染所需要的字形。

以下就是一个字形编辑器工具（GlyphDesigner，来自 71 squared-收费的），图中显示的是，正在编辑 BritannicBold 字体的字形 SpriteSheet。

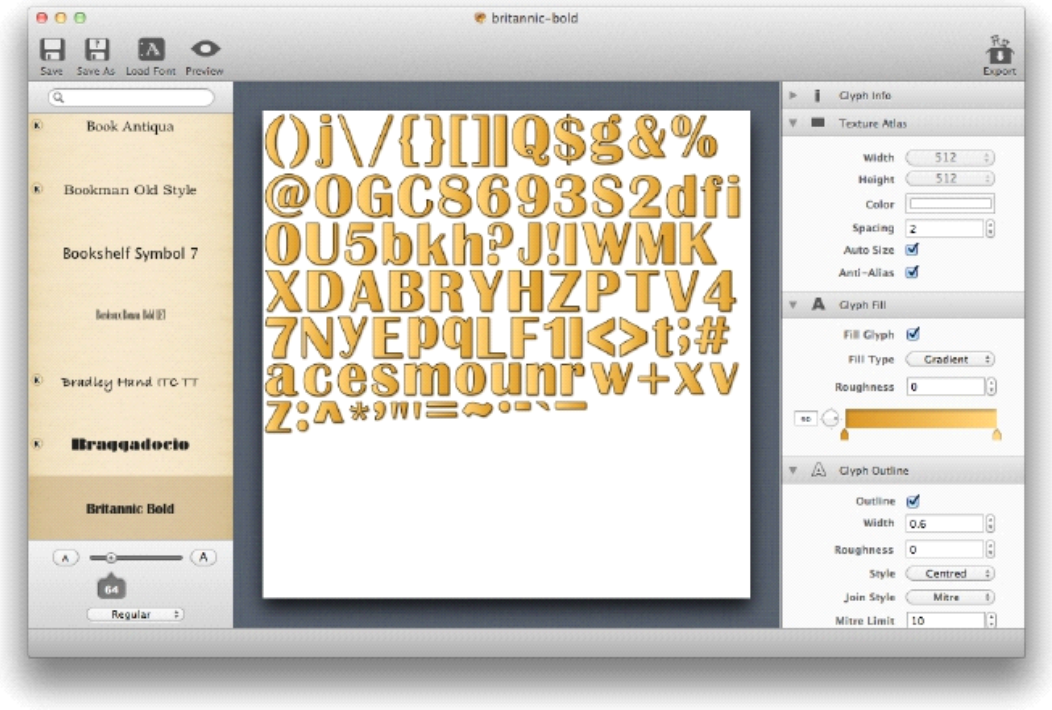

图 1.36 GlyphDesigner on MacOS.

在 Windows 下也有一个类似的工具叫 Bitmap Font Generator（来自 Angel Code -免费的）：

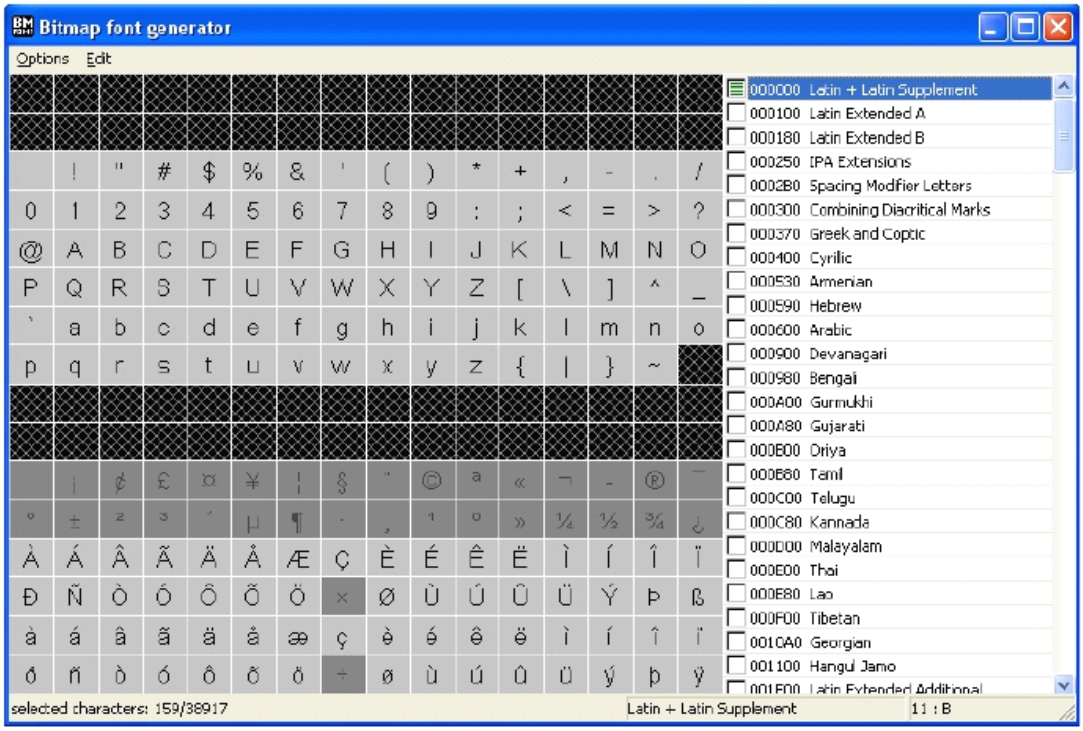

图 1.37 Bitmap Font Generator on Windows.

但是，请相信我，你会更喜欢 MacOS 上的这个。

当然，你也可以先用一个包含所有字形的 TextField，在运行时使用 BitmapData 的绘制函数来生成字形 SpriteSheet。根据你的运行平台（桌面或者移动），你可以进行灵活地选择。如果在运行时生成，可以减少程序的大小，如果嵌入字形则会加快程序的启动速度。你可以自己决定。

在导出的时候，纹理会被存为图像，而决定字形在图像上的坐标等信息的描述则被存为.fnt 文件（XML 文件或者文本文件），其格式如下 ：

```
<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>
```

这两个文件被导出后，通常可以像下面这样使用它们。

```
[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;
```

我们将通过 TextField 的静态函数来使用这些文件：

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

在下面，我们用 registerBitmapFont 方法将字体纹理和描述传入给 BitmapFont 对象：

```
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);
            }
        }
    }
```

一旦注册了字体，可以使用它的字体名来创建 TextField 对象。运行结果如下：

图 1.38 使用位图字体的一些文本

现在，让我们使文本内容变得更长一点，看看会发生什么：

```
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)
```

如我们所想到的那样，我们的文本框太少了，无法显示完整的文本内容，多余的内容被裁剪了：

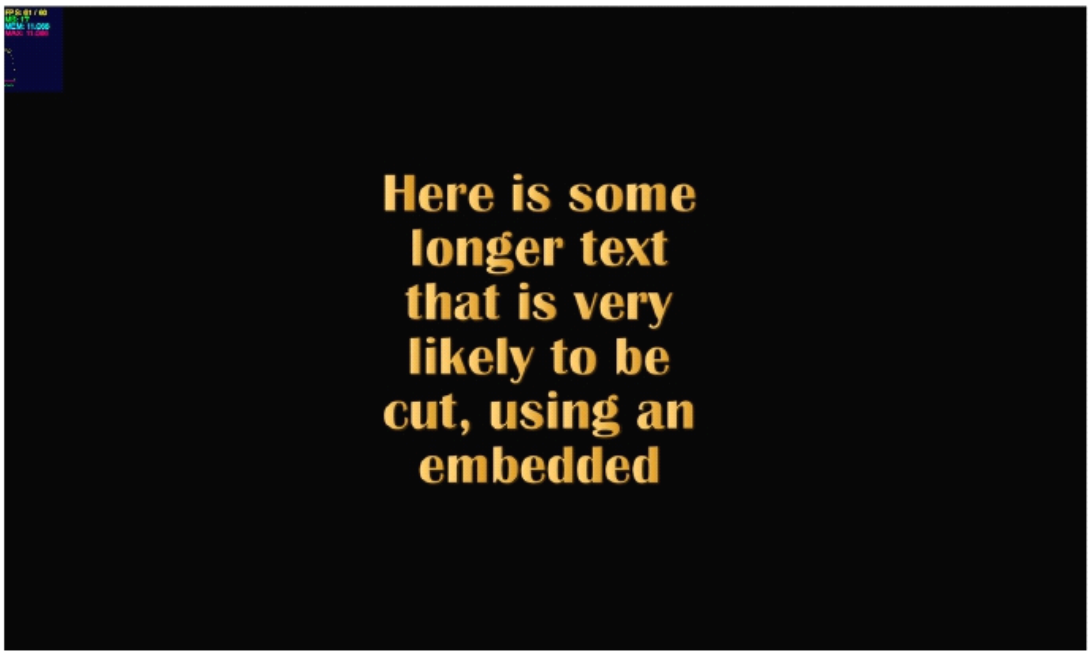

图 1.39 Some large bitmp text being cut off.

不过，Starling 给 TextField 提供了 autoScale 属性：

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

当你需要使你的文本本地化或者你需要文本适应一个指定的框框时，这个功能就非常有用了。大部分时候，因为设计的原因，你都希望文本被缩放一点点，以使得无论文本内容有多长，你的布局都可以保持不变。

下面就是使用了 autoScale 属性的运行结果：

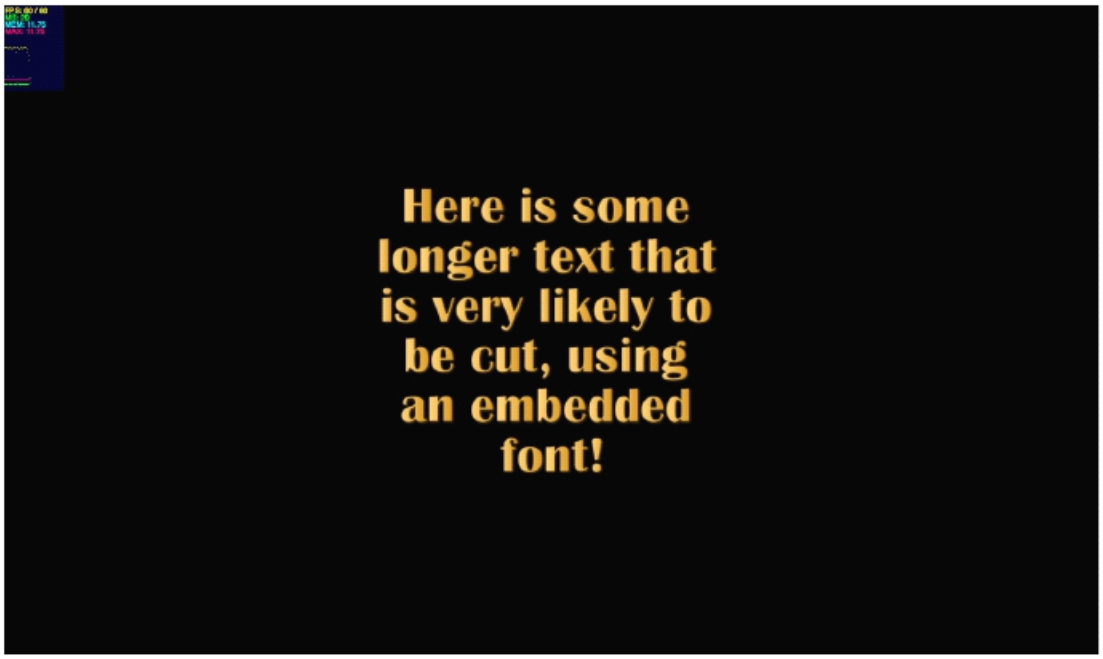

图 1.40 Large text being scaled down to fit the bounding box

我们使用位图字体来优化我们之前关于菜单和背景卷动的例子：

```
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：

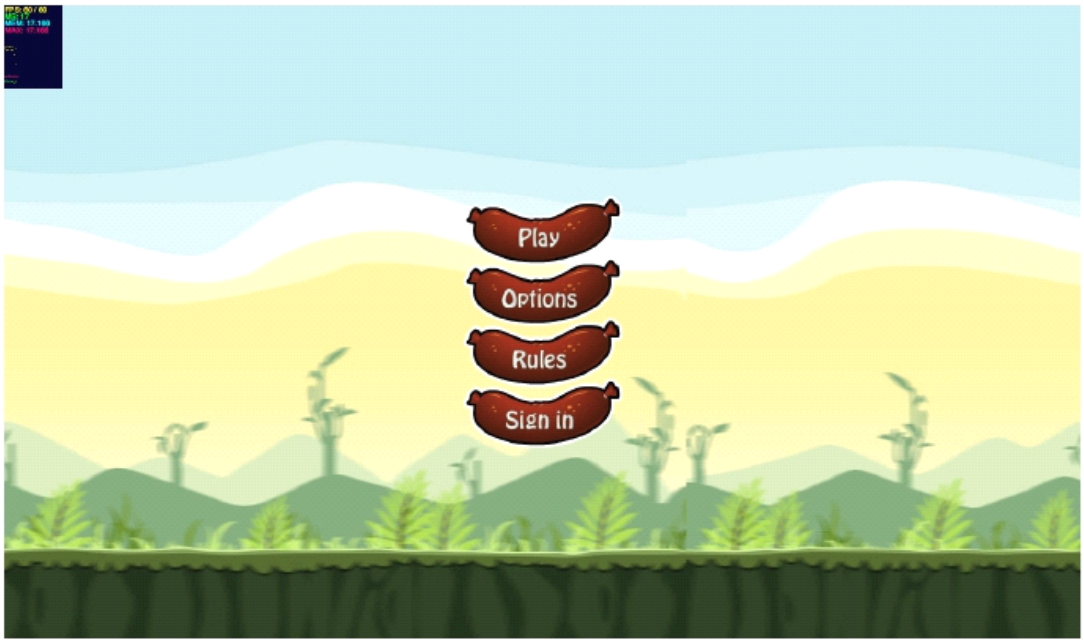

图 1.41 Custom font used in our menu.

下面来讨论 Starling 的另一个功能，RenderTexture。

18、 RenderTexture

starling.textures.RenderTexture 允许开发者创建一个非破坏性（non-destructive）绘图。作为 Flash 开发者，可以联想到 BitmapData 的功能。这个功能在开发类似绘图工具的应用来说是非常有用的，因为你需要在连续绘制新的绘图时依然保留之前的绘图。

在下面的代码里，我们在 Starling 里重现了 BitmapData.draw 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);
            }
        }
    }
}
```

只要按下鼠标并且移动它，就可以看到如下效果：

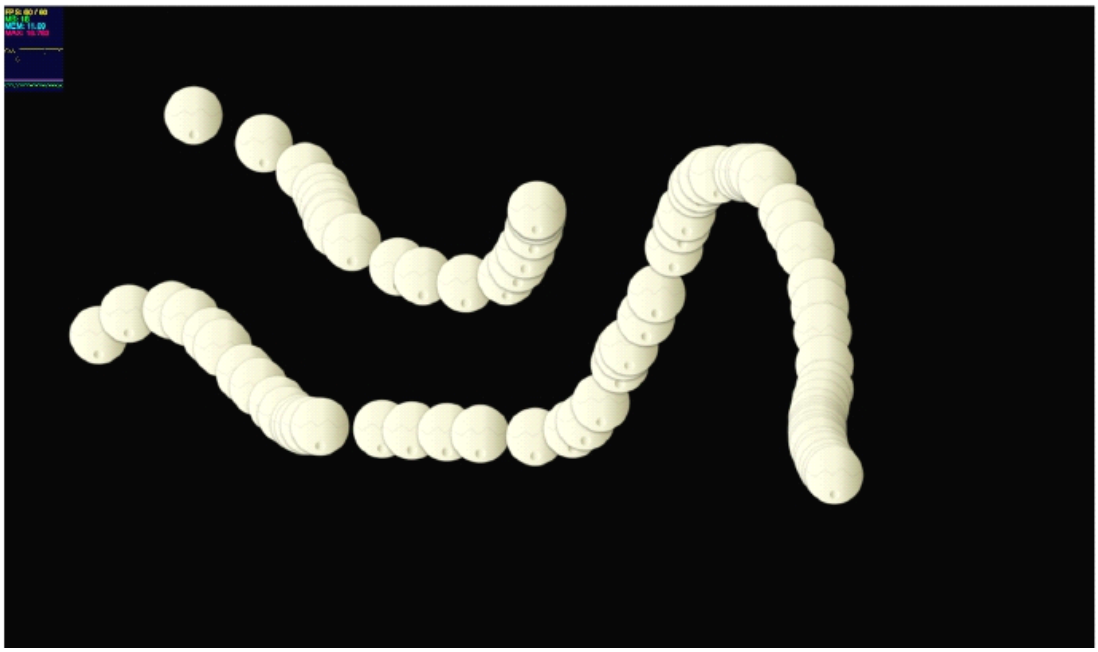

图 1.42 Non destructive drawing.

下面我们将讨论一个对于 Flash 开发者来说非常熟悉的一个功能：Tweening！

19、 Tweens

Starling 用自己的 Tween 引擎实现了 Tween 功能，支持大部分最常用的效果，如下图所示：

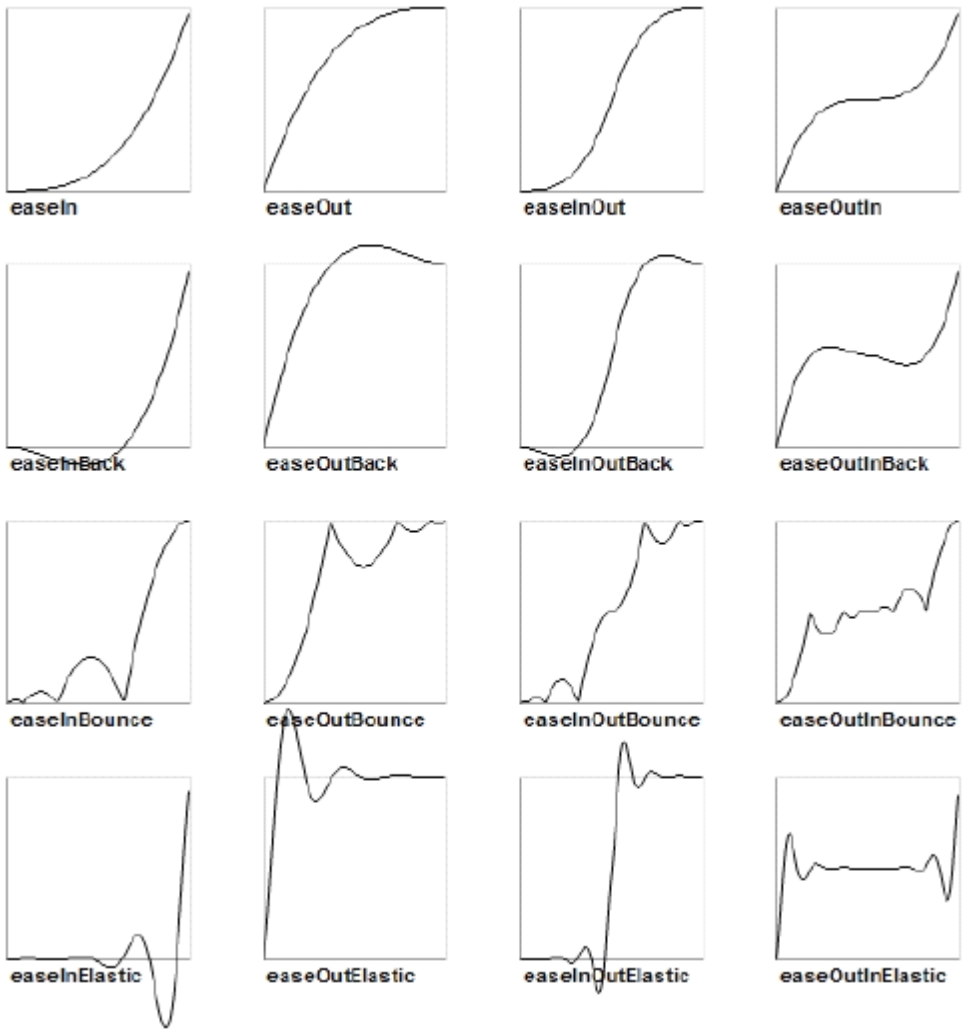

图 1.43 Easing equations available in Starling (figure courtesy of sparrow-framework.org).

在下面代码中，我们在一个 TextField 对象的 x 和 y 属性上应用 Tween 效果：

```
// 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);
```

下面是 Tween 所提供的方法：

animate : 使目标对象的指定属性补间到指定值，你可以重复多次调用这个方法；
currentTime：该补间动画自创建以来已经经历过的时间；
delay：设置补间动画开始前的延时；
fadeTo：使目标对象的Alpha值补间到指定值；
isComplete：用于判断补间是否完成；
moveTo：使目标对象的X和Y值同时补间到指定值；
onComplete：当补间动画结束时回调的函数；
onCompleteArgs：传递给补间动画结束回调函数的参数；
onStart：当补间动画开始时的回调函数；
onStartArgs：传递给补间动画开始回调函数的参数；
onUpdate：补间动画播放过程中的进度回调函数；
onUpdateArgs：传递给进度回调函数的参数；
roundToInt:在补间计算过程中，浮点型是否需要转换为整型，默认为false；
scaleTo：使目标对象的scaleX 和scaleY同时补间到指定值；
target：应用补间动画的目标对象；
totalTime：补间动画所花费的总体时间（秒）；
transition：补间动画所使用的过渡函数（如图 1.43 所示）；

在下面的例子中，我们通过 onComplete 属性来监听 Tween 的结束，并且通过 onCompleteArgs 功能传递参数给监听函数，以决定在 Tween 结束后是否需要释放该对象。

```
// 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];
```

在下面代码中，我们通过 onStart、onUpdate 和 onComplete 监听 Tween 的进度：

```
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")
            }
        }
    }
```

下面让我们看看更多的组织资源的方法。之前我们已经简单地使用 Embed 标签来嵌入资源。

在大型项目里，你可能希望在一个统一的地方获取所有资源。下面章节将会用到一些简单的技巧来组织和重用你的资源。

20、 资源管理（Asset management）

到目前为止，我们已经简单地用过资源。为了优化使用资源的方法，非常建议使用一个 Assets 对象来统一管理资源。这个 Assets 对象将负责集中管理资源，保证它们能够被重用，而不是被丢弃和多次实例化。

在下面代码中，我们在 Assets 上定义了一个函数 getTexture，我们用它来获取一个被嵌入的纹理。

```
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.");
}
```

注意，这里我们使用了一个 Dictionary 来存储资源，所以在下次需要资源时，只需要从这个资源池里直接获取，而不需要再次创建它。

通常地，纹理通过 Embed 标签来嵌入：

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

注意，Starling 并不要求你一定使用嵌入纹理，你也能使用 Loader 对象来动态地加载一个纹理。

```
// 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);
    }
```

使用 BitmapData 也可以动态地生成纹理，到目前为止，我们已经在 Starling 中使用过现成的位图来生成纹理。但如果我们想动态地创建一个 BitmapData 并用作纹理呢？

我们将要使用到 Texture 类的 fromBitmapData 函数：

```
// 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 应用可能运行在屏幕尺寸不同的移动设备或者桌面浏览器里。Starling 可以让你很容易地处理屏幕尺寸的问题。下面让我们看看是怎么做到的。

21、 处理屏幕尺寸（Handling screen resizes）

作为一个 Flash 开发者，你通常会依赖一个非常简单的事件来处理屏幕尺寸的改变，这个事件就叫作：Event.RESIZE，每当屏幕尺寸发生变化时都会触发这个事件。使用 stage.stageWidth 和 stage.stageHeight 属性，你能够对你的界面进行合理布局，而不需要担心屏幕的大小。

为了实现这个，Starling 使用 starling.display.Stage 来派发类似的事件，叫作：ResizeEvent.RESIZE，它可以让你轻松处理屏幕尺寸的动态变化。

在下面代码中，我们来修改我们的第一个例子，使方块始终处于屏幕的中心。

```
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;
    }
}
}
```

只要屏幕尺寸发生了改变，ResizeEvent.RESIZE 事件就会被触发，该事件会带上最新的尺寸数据，我们再手动地将它们赋给 stageWidth 和 stageHeight 属性。然后，我们重新设置方块的坐标，让它处于屏幕的中央。

22、 Plugging Starling with Box2D

最惊奇的是 Starling 的显示列表可以非常容易与现有的框架融合。下面举一个例子，在我们的游戏中使用 Boxe2D 框架来实现物理效果。

你可以在 Box2D 的开源网站上学习更多的关于 Box2D 的知识：
http://box2dflash.sourceforge.net/

在下面代码中，我们让一些方块在重力的作用下掉落到地面上。而且这一切都是在 GPU 的渲染下完成的。

```
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;


        Private var m_world:b2World;
        Private var m_velocityIterations:int = 10;
        Private var m_positionIterations:int = 10;
        Private 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);
          }
     }
}
```

运行结果如下：

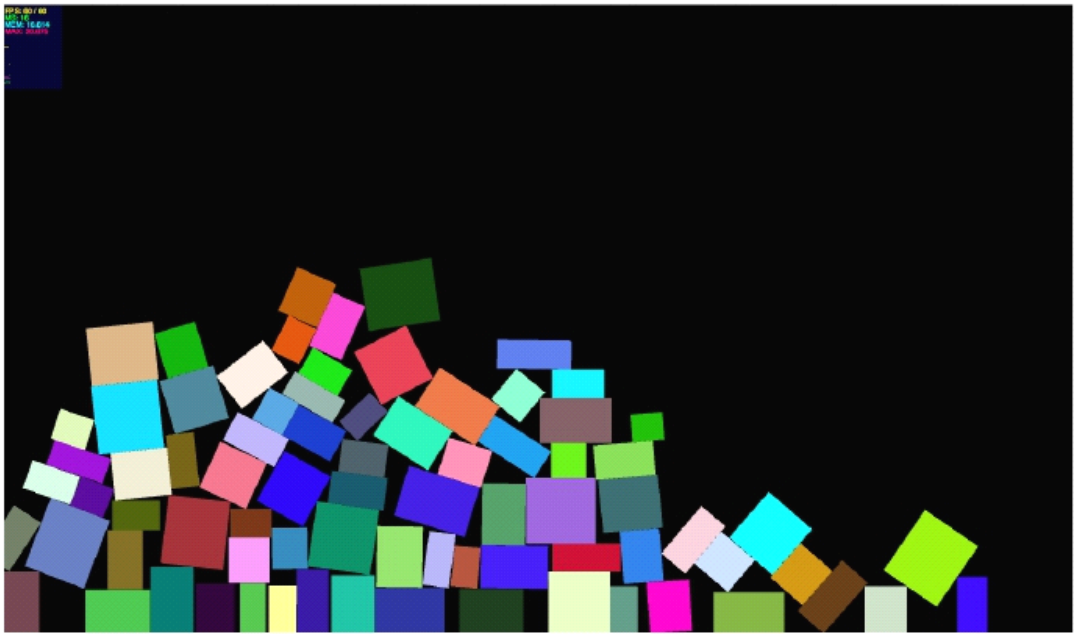

图 1.44 Box2D used with Starling.

下面是一个运行在移动设备上的 AIR 中的版本：

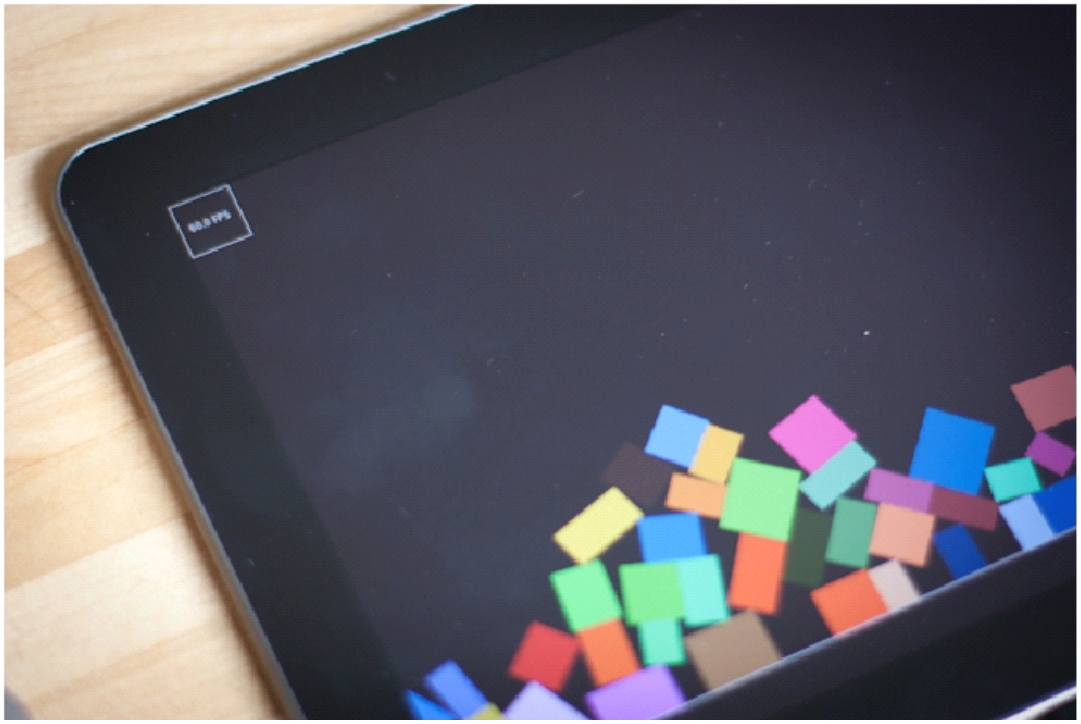

图 1.45 The same applicaton running on a tablet at 60fps.

当然，我们可以很容易地用纹理来替代这些方块，从而得到更丰富的图形效果。现在你已经知道怎么创建让人惊叹的基于 GPU 的 2D 游戏了。

23、 Profiling Starling

现在到了要去分析它的性能的时候了。在所有的游戏中，最需要做的一件事就是显示 FPS。通常，开发者需要在整个游戏的调试期间显示 FPS，以监控性能瓶颈并改进它。

从本教程一开始，我们一直在使用一个经典的 Stats 类，它来自于 mr.doob：
https://github.com/mrdoob/Hi-ReS-Stats

到目前为止，我们一直在使用这个统计类。当然也有例外，特别是在移动设备中。在将来的移动 AIR 版本中，也能使用 Stage3D。运行在移动设备，比如 Android 平台上，使用原生显示列表的同时使用 Stage3D，将会遇到很大的性能问题。可以告诉你，在一些 GPU 上，使用原生的显示列表，将会降低一半的性能。所以，原本应该是 60FPS 的结果变成 30FPS 消耗在显示列表上，30FPS 消耗在 Stage3D 上。因此，即便只是为了调试，也应该将所有的游戏内容都在 Stage3D 中渲染。

在之前正好讲到位图字体的概念，所以我们将开发一个小小的 FPS 类来显示帧率，使之能够用 Stage3D 渲染。

下图描述了我们正要导出的字形 SpriteSheet。注意，我们只选择了用于 FPS 显示的字形，这样可以将 SpriteSheet 尽可能的轻量。

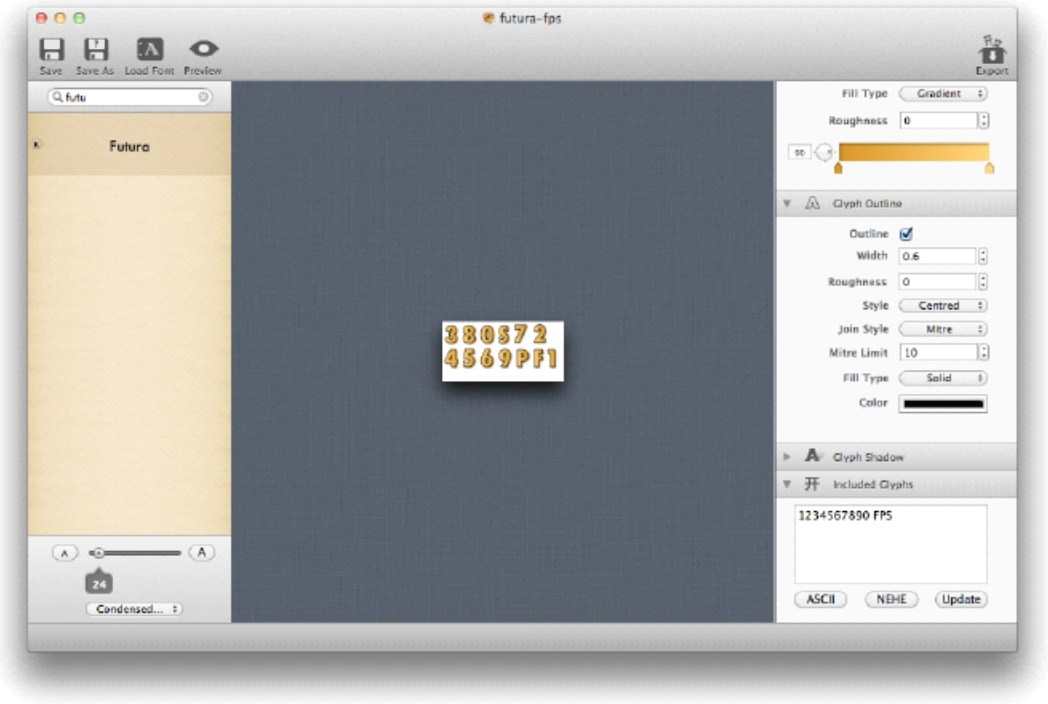

Figure 1.46 The text texture for our FPS counter.

字形导出后，我们使用基于 BitmapFont 的 TextField 对象来显示 FPS：

```
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;
            }
        }
    }
```

要使用它非常简单，在游戏中加上如下代码即可：

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

下面就是运行的结果：

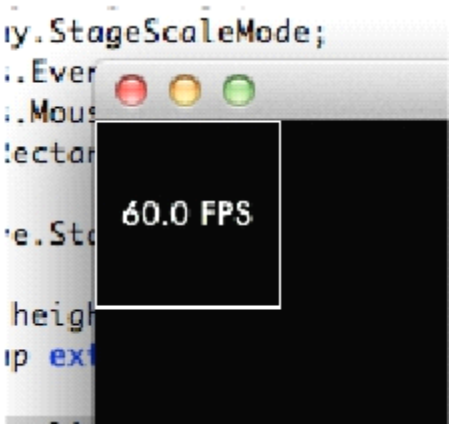

图 1.47 Our FPS counter rendered through the GPU.

现在可以在上面那个物理 Demo 中使用它了：

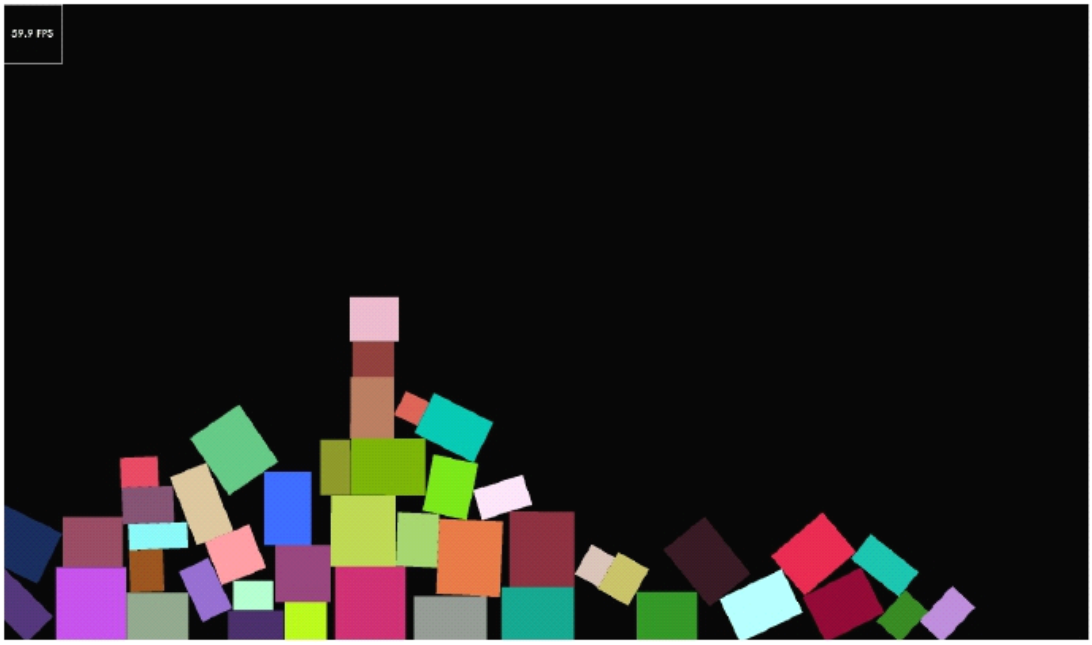

图 1.48 Our FPS counter integrated in our application.

我们现在有了一个可以在移动设备分析性能的方法了。如前面所说，我们可以完全摆脱原生的显示列表了。

注意，这个大名鼎鼎的由 mr.doob 创建的 Stats 类最近已经被移植进 Starling。你能在下面这个网址找到他的其它贡献：
http://forum.starling-framework.org/topic/list-of-user-contributions#post-181

当然，我们可以很容易地用纹理来替代这些方块，从而得到更丰富的图形效果。现在你已经知道怎么创建让人惊叹的基于 GPU 的 2D 游戏了。

24、 粒子（Particles ）

粒子可用来实现非常炫丽的效果。不管你是否相信，粒子看起来很酷，但是却并不复杂。从技术上讲，粒子就是一些具有纹理的四边形，依据一定的混合模式四处移动，以产生漂亮的叠加效果。

为了在 Starling 里设计粒子，我们将使用一个非常方便的工具： ParticleDesigner。它与另一款用于设计字体的工具 GlyphDesigner 是同一家公司开发的。下面是该工具的一个快照，它正运行在仿真模式，你可以实时预览你所设计的粒子效果。

图 1.49 Particle Designer on MacOS.

它的主窗口是一个参数面板。你可以轻松地花几个小时来了解所有这些选项，用它们来创建你想要的粒子。还有一个按钮，可以用来随机地生成这些参数，用来产生不同的粒子效果。下面图中所示的是 save-as 对话框，用于导出 ParticleEmitter 文件（.pex）和纹理。这两个文件将会被 ParticleDesignerPS 对象用到。

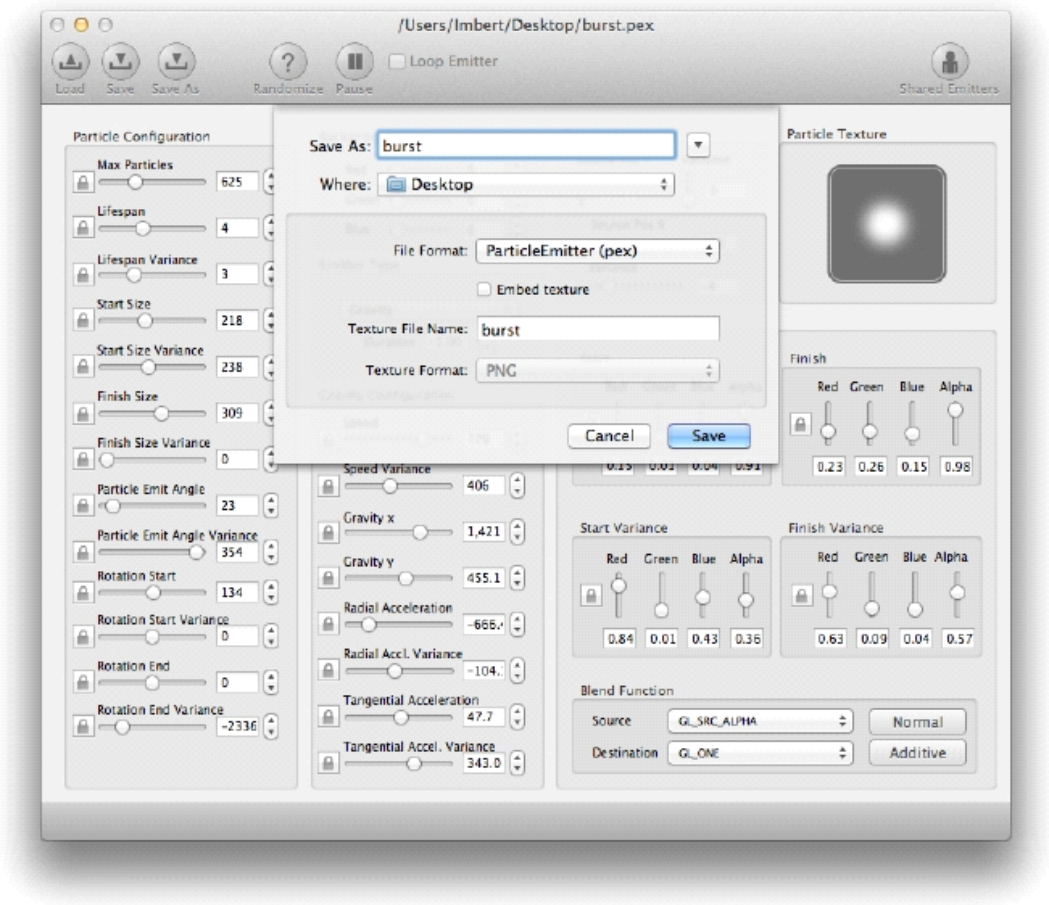

图 1.50 Particle Designer on MacOS.

下面是一个用 ParticleDesigner 创建并运行中 Starling 中的粒子效果的示例。

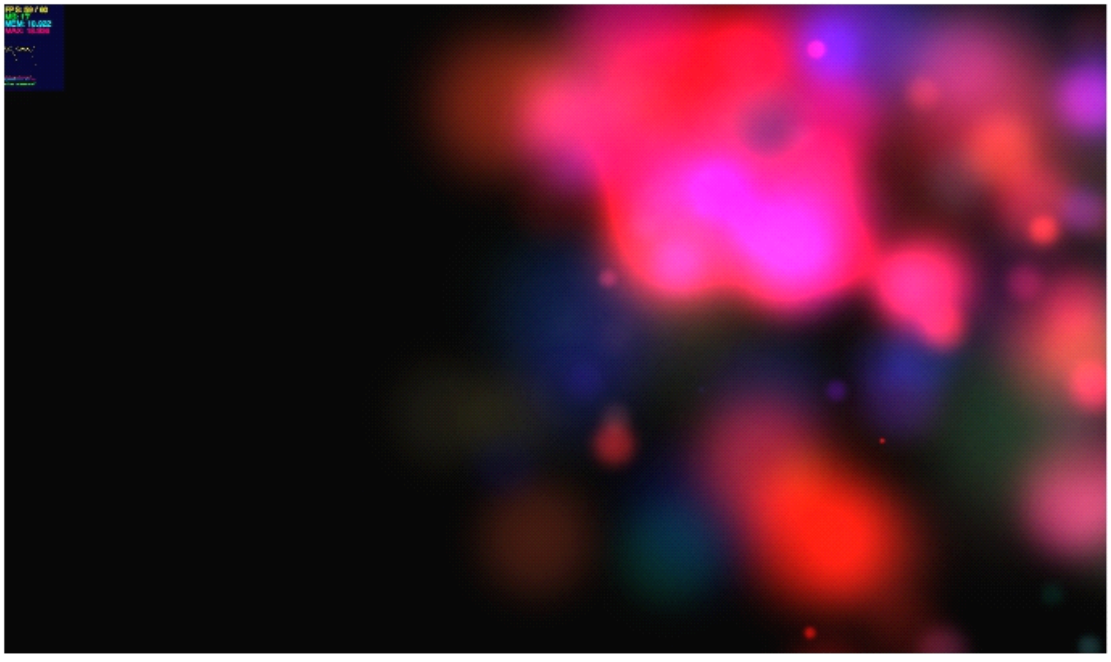

图 1.51 Custom particles running through Starling.

很漂亮不是吗？这个粒子引擎是 Starling 的一个扩展功能，可以在下面这个地址下载：
https://github.com/PrimaryFeather/Starling-Extension-Particle-System

下面来看看另一个漂亮的粒子效果：

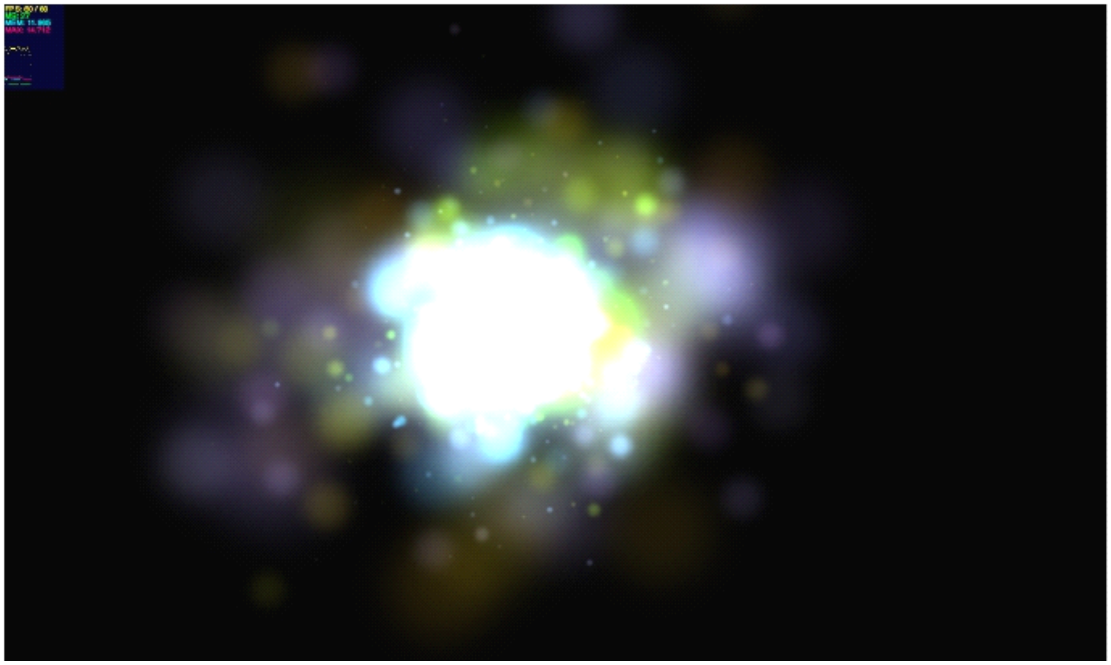

图 1.52 Custom particles running through Starling.

请记住，在 Starling 里，粒子与其它的动画对象一样，需要被添加到一个 Juggler 里才能看到粒子的动画效果。

```
// 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);
```

由于 ParticleDesignerPS 实际上是一个 DisplayObject，所以你可以将它设置在任意你想要的坐

标上，你也能使用 DisplayObject 的所有特性。当然，如果你不需要这个粒子效果了，你需要将它从 Juggler 里移除，并且 dispose 它。

下面是 Lee Brimelow (leebrimelow.com)创建的一个粒子效果的示例。这个粒子效果用来模拟一艘太空船的火焰。

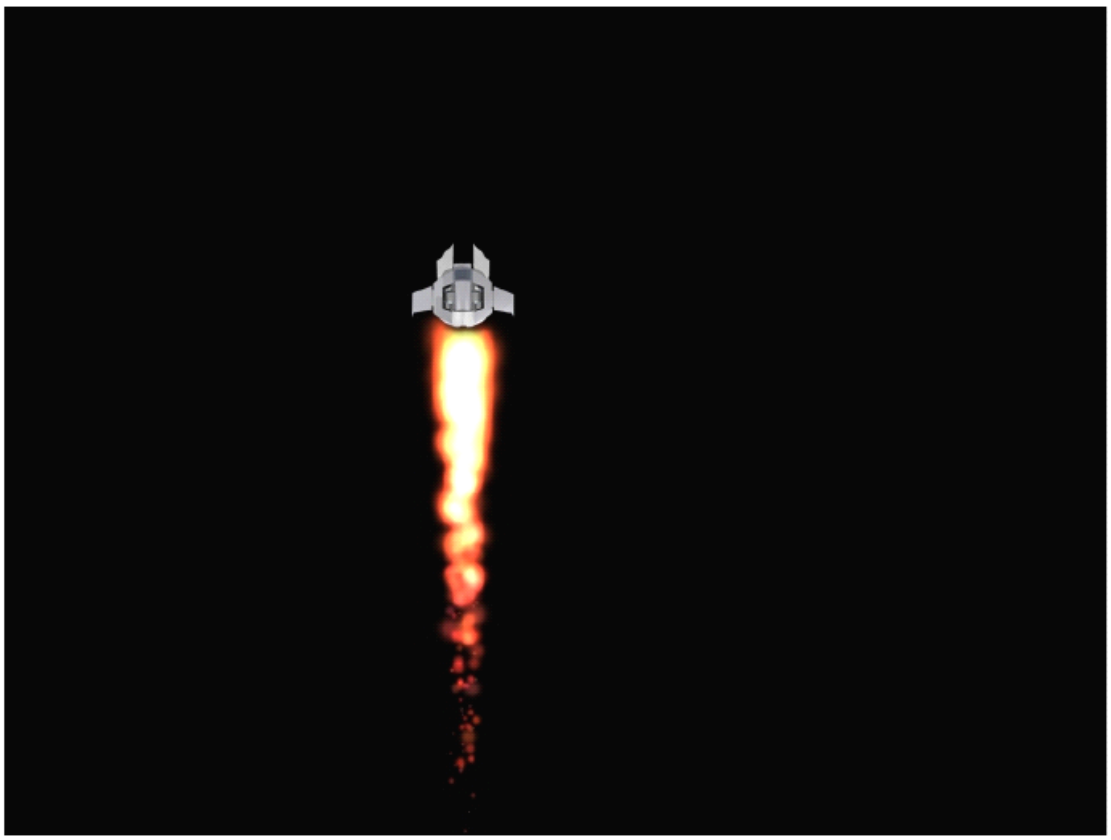

图 1.53 Particle effect integrated to our space ship.

然后我们在太空船发射的火箭上添加一些粒子。

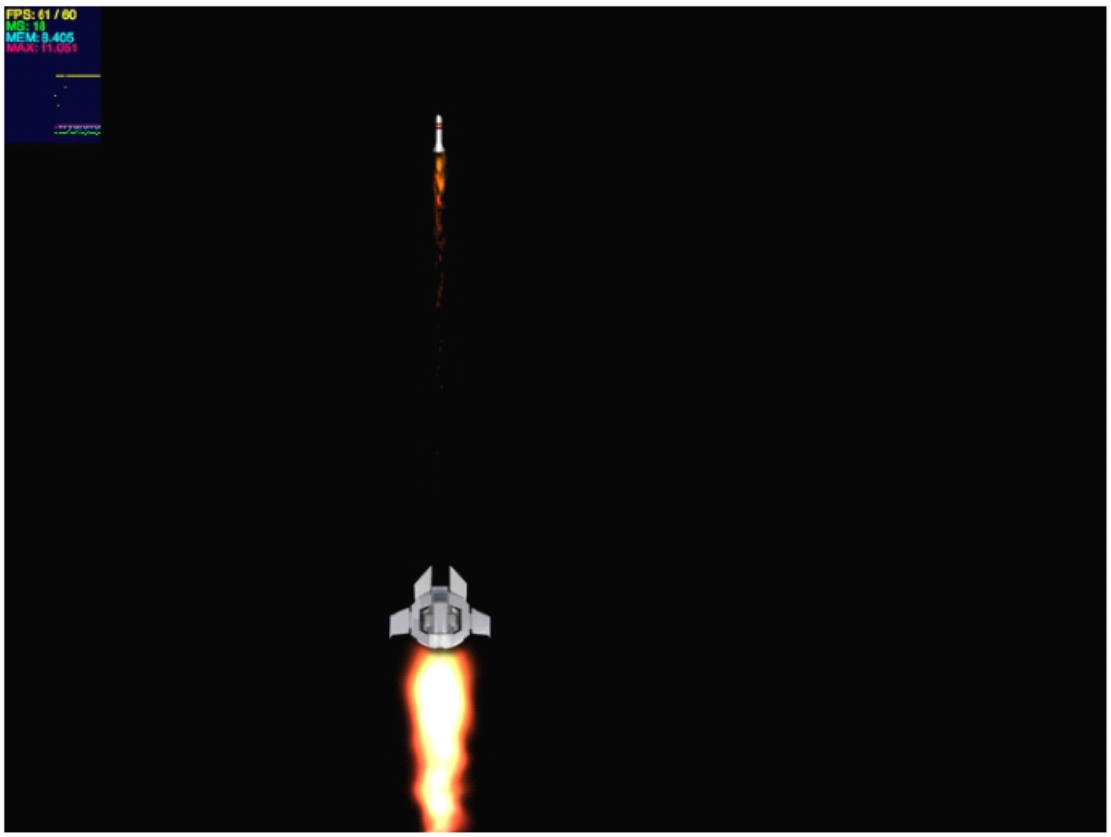

图 1.54 Rockets with fire.

一旦火箭飞出舞台，我们需要将这些粒子从场景和 Juggler 里移除。

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

但是我们忘记了一件非常重要的事情。我们忘记 dispose 这些粒子了。如果不 dispose 它们，它们不会从 GPU 缓存里移除。于是，我们的代码应该这样写：

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

我们也可以使用 removeChild 来同时移除和 dispose 这些粒子：

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

在一种情形下，我们可以容易地检测到粒子的当前坐标，如果它超出了场景的范围，就 dispose 它。另一种情形下粒子并没有超出场景，而是在一个随机的位置生命周期结束了。在这个情形下，我们就需要检测粒子的生命周期是否已经结束，如果结束了，就 dispose 它。

正好 ParticleDesignerPS 类提供 isComplete 属性，用于查询粒子的状态。我们用一个 Vector 来存储所有的粒子，然后在 GameLoop 里检测所有的粒子是否 isComplete。

```
for each (var p:ParticleDesignerPS in particlesVector)
{
```

```
        if ( p.isComplete )
            removeChild(p, true);
    }
```

每创建一个粒子，都要将它的引用添加到 particlesVector 里 ，在主游戏循环里将会自动地处理粒子的释放。

现在已经全部介绍了 Starling。希望大家能够喜欢它。现在让我们开始用 Starling 来创建一个让人惊奇的应用吧。

25、 Credits

我要感谢 Chris Georgenes （mudbubble.com）为这本书提供了丰富的素材。
我要感谢 Daniel Sperl（Starling 的作者），Starling 是一个非常漂亮的框架，与你一起工作是我的荣幸。