Introducing Starling

What is Starling? 2
Why Starling? 2
Philosophy 2
  Intuitive 2
  Lightweight 2
  Free 2
How 2
Layering restrictions 8
Getting Started 9
Setting up your scene 10
Wmode requirements 14
Stage quality 15
Progressive enhancements 16
The Display List 16
Event model 28
  Event propagation 29
Touch Events 29
  Simulating multi-touch 30
Texture 33
  Image 35
  Collision detection 43
  Drawing API 44
Flat Sprites 45
MovieClip 48
  Texture Atlas 54
Juggler 60
Button 62
TextField 69
  Embedded fonts 72
  Bitmap fonts 74
RenderTexture 83
Tweens 85
Asset management 89
Handling screen resizes 90
Plugging Starling with Box2D 91
Profiling Starling 95
Particles 99
Credits 106

What is Starling?

Starling is an ActionScript 3 2D framework developed on top of the Stage3D APIs (available on desktop in Flash Player 11 and Adobe AIR 3). Starling is mainly designed for game development, but could be used for many other use cases. Starling makes it possible to write fast GPU accelerated applications without having to touch the low-level Stage3D APIs.

Why Starling?

Most Flash developers want to be able to leverage GPU acceleration (through Stage3D) without the need to write such higher-level frameworks and dig into the low-level Stage3D APIs. Starling is completely designed after the Flash Player APIs and abstracts the complexity of Stage3D (Molehill) and allows easy and intuitive programming for everyone.

Obviously Starling is for ActionScript 3 developers, especially those involved in 2D game development; of course you will need to have a basic understanding of ActionScript 3. By its design (lightweight, flexible and simple), Starling can be used also be used for other use cases like UI programming. That said, everything is designed to be as intuitive as possible, so any Java™ or .Net™ developer will get the hang of it quickly as well.

Philosophy

Intuitive

Starling is easy to learn. Especially Flash/Flex developers will feel at home immediately, since it follows most of the ActionScript dogmas and abstracts the complexity of the low-level Stage3D APIs. Instead of coding against concepts like vertices buffer, perspective matrices, shader programs and assembly bytecode, you will use familiar concepts like a DOM display list, an event model, and familiar APIs like MovieClip, Sprite, TextField, etc.

Lightweight

Starling is a lightweight bird in many ways. The amount of classes is limited (around 80k of code). There are no external dependencies beside Flash Player 11 or AIR 3 (mobile support will come in a future release). This keeps your applications small and your workflow simple.

Free

Starling is free and alive. Licensed under the Simplified BSD license, you can use it freely even in commercial applications. We are working on it every day and we count on an active community to improve it even more.

How

Starling uses the Stage3D APIs behind the scenes which are low-level GPU APIs running on top of OpenGL and DirectX on desktop and OpenGL ES2 on mobile devices. As a developer you have to know that Starling is the ActionScript 3 port of Sparrow (http://www.sparrow-framework.org), the equivalent library for iOS relying on OpenGL ES2 APIs:

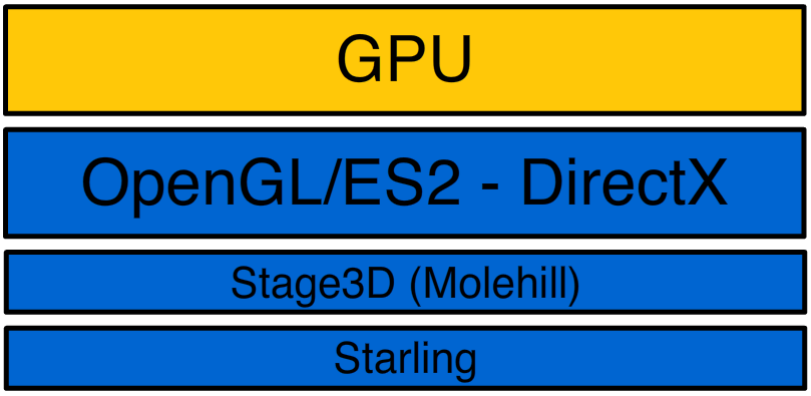

Figure 1.1

Starling layer on top of Stage3D (Molehill).

Starling recreates many APIs that Flash developers are already familiar with, below is a figure illustrating the APIs exposed by Starling when it comes to graphical elements:

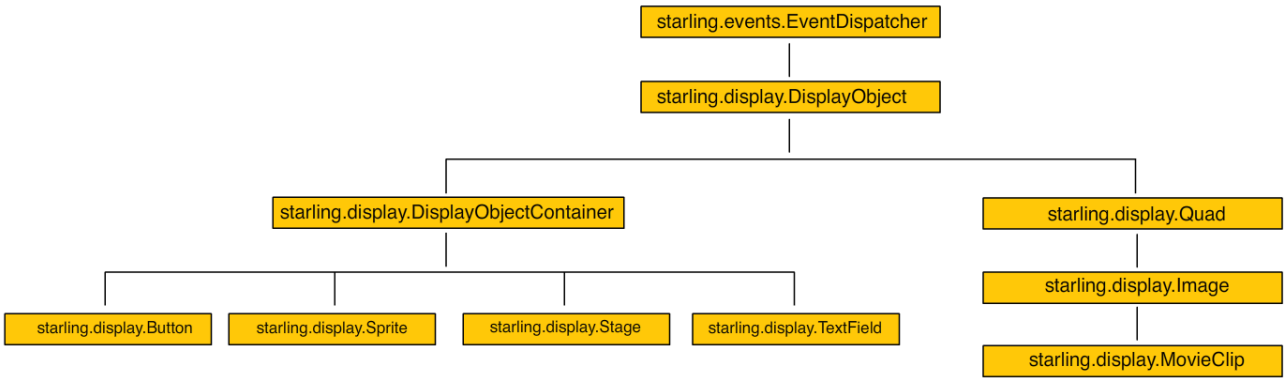

Figure 1.2

DisplayObject inheritance.

Many people think that the Stage3D APIs are strictly limited to 3D content. Actually, to be fair, the name can be confusing; cause there is the term 3D in it, so how can you do 2D with them?

The figure below, illustrates the idea, how can we draw something like a `MovieClip` with the `drawTriangles` API?

Figure 1.3
drawTriangles and 2D?

Actually, it is very simple, GPU are extremely efficient at drawing triangles, so the `drawTriangles` API will be drawing two triangles, we will then be sampling a texture and applying it to the triangles using UV mapping, we will then end up with our textured quad, which represents our sprite. By updating the texture every frame on our triangles, we would end up with a `MovieClip`, pretty cool hu?

Now, the good news is that we will not even have to go through those details when using Starling, we will just provide our frames, supply them to a Starling `MovieClip` and voila!

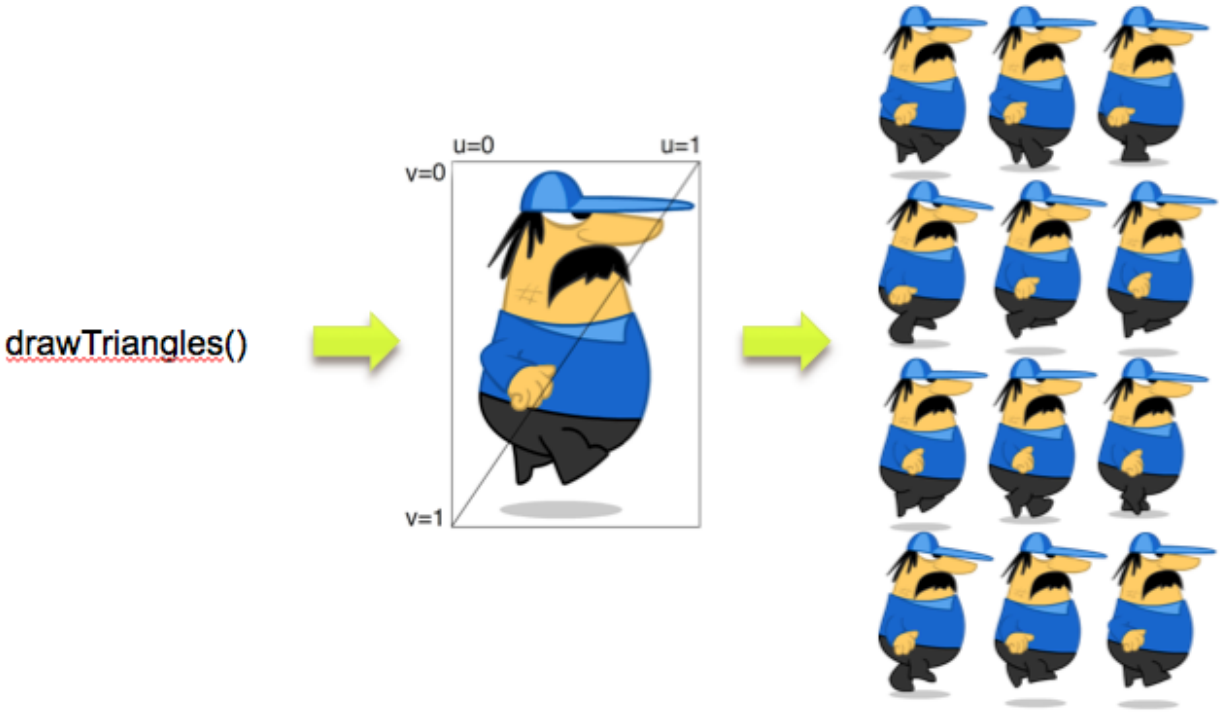

Figure 1.4
drawTriangles + textured quad = 2D.

To give you an idea of how Starling reduces the complexity, let's see what code we would have to write to display a simple textured quad using the low-level `Stage3D` APIs:

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

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

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

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

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

// create the bitmap texture
var bitmap:Bitmap = new TextureBitmap();

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

And we would end up with the following result:

Figure 1.5
A simple textured quad.

Pretty complex code for this right? That is the cost of having access to low-level APIs, you get to control a lot of things, but at the cost of low-levelness.

With Starling, you will write the following code:

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

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

// set the properties
image.pivotX = 50;
image.pivotY = 50;
image.x = 300;
image.y = 150;
image.rotation = Math.PI/4;

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

As an ActionScript 3 developer, used to the Flash APIs, you will feel pretty much at home with these APIs exposed, while all the complexity of the `Stage3D` APIs is done behind the scenes.

If you try to use the redraw regions feature, you will see that Starling as expected, renders everything on `Stage3D`, not the classic display list. The figure below illustrates the behavior. We have a quad rotating on each frame; the redraw regions only show the FPS counter sitting in the display list (running on the CPU):

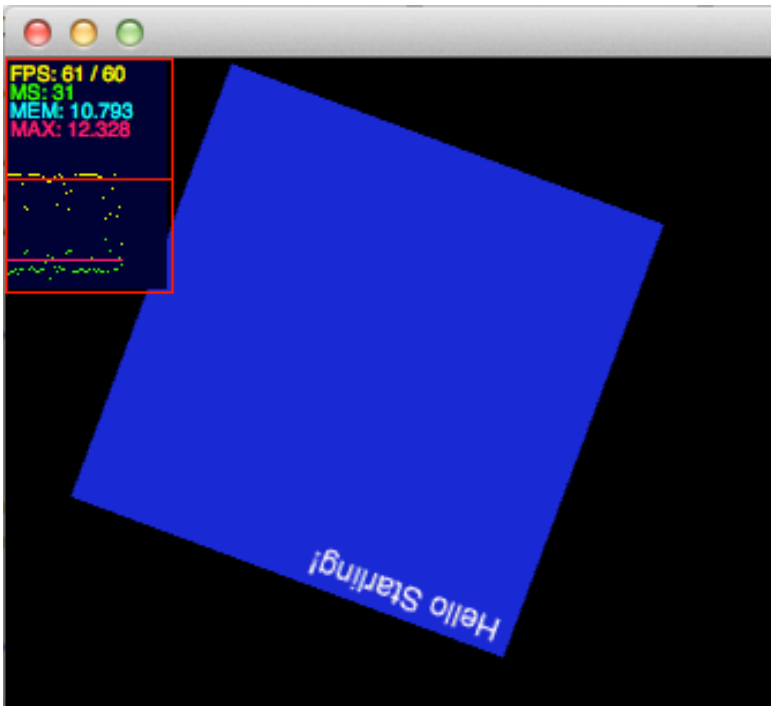

Figure 1.6

Content rendered through Stage3D.

Remember that with the Stage 3D architecture, the content is completely rendered and composited by the GPU, as a result the redraw regions feature used for the display list cannot be used.

Layering restrictions

As a developer, using Starling (and as a result Stage 3D), you need to remember one limitation will when developing content. As mentioned earlier, Stage3D is literally a new rendering architecture inside Flash Player. A GPU surface is placed under the display list, meaning that any content running inside the display list will be placed above the Stage3D content, at not time, content running in the display list can be placed under the Stage3D layer.

The figure below illustrates the layering:

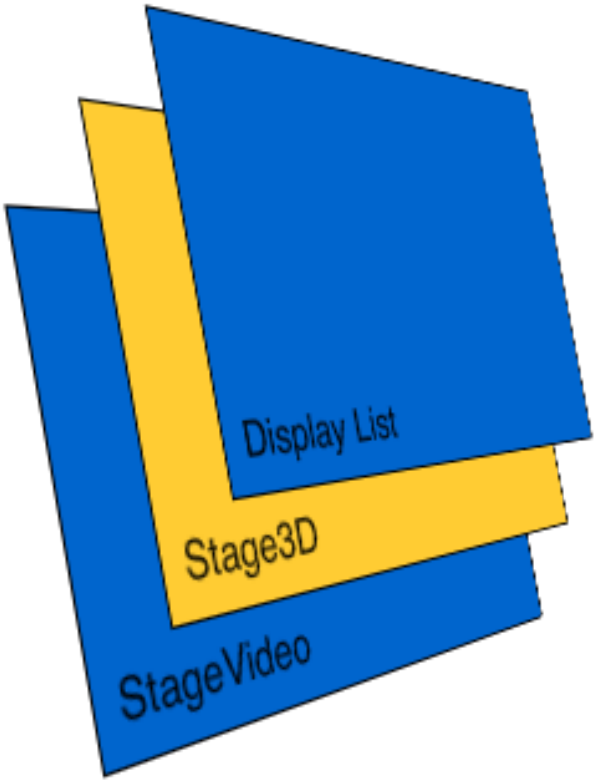

Figure 1.7
Layering with Stage3D, StageVideo and the display list.

Note that the `Stage3D` object cannot be transparent, such a capability would allow developers to play video using the StageVideo technology (introduced in Flash Player 10.2 - http://www.adobe.com/devnet/flashplayer/articles/stage_video.html) and overlay the video with content rendered through Stage3D. Such a feature may be enabled in a future release of the Flash Player and AIR.

Getting Started

In order to download Starling, check the following links:

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

Note that Starling is licensed under the simplified BSD licence, so feel free to use Starling in any kind of projects. You can contact the people behind the Starling framework at office@starling-framework.org.

Once downloaded, you will reference the Starling library as any other AS3 library. Then, to use the new Stage3D APIs required by Starling, you will need to target SWF version 13 by passing in an extra compiler argument to the Flex compiler: *-swf-version=13*. Directions are below.

If you are using the Adobe Flex SDK:

- Download the new playerglobal.swc for Flash Player 11.
- Download Flex 4.5 SDK (4.5.1.21328) from the Flex 4.5 SDK table - http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4.5
- Install the build in your development environment
- In Flash Builder, create a new ActionScript project: File -> New -> ActionScript project.
- Open the project Properties panel (right-click and chose 'Properties'). Select
- ActionScript Compiler' from the list on the left.
- Use the 'Configure Flex SDK's' option in the upper right hand corner to point the project to Flex build 21328. Click ok.
- Configure your project to target SWF version 13
- Open the project Properties panel (right-click and chose 'Properties'). Select
- ActionScript Compiler' from the list on the left.
- Add to the 'Additional compiler arguments' input: -swf-version=13. This ensures the outputted SWF targets SWF version 13. If you compile on the command-line and not in Flash Builder, you need to add the same compiler argument.
- Ensure you have installed the new Flash Player 11 build in your browser

To be able to access the Flash Player 11 APIs you will also need the updated playerglobal.swc, available at the following address: http://labs.adobe.com/downloads/flashplayer11.html

Setting up your scene

Enough of introduction, let's dig into the code and see what this little bird can do. Starling is very easy to setup, a `Starling` object needs to be created and added to your main class.

> Starting from now, when referring to objects like MovieClip, Sprite, etc, we will imply the Starling APIs and not the native ones from the Flash Player, unless specificed explicitly.

First, the Starling constructor expects multiple arguments, here is the signature:

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

Actually, the only 2 first are really used commonly. The `rootClass` argument expects a reference to a class extending `starling.display.Sprite` and as a second argument, our stage:

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

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

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

```
            addChild ( new Stats() );

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

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

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

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

Below, our `Game` class creating a simple quad when added to 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
         {
            q = new Quad(200, 200);
            q.setVertexColor(0, 0x000000);
            q.setVertexColor(1, 0xAA0000);
            q.setVertexColor(2, 0x00FF00);
            q.setVertexColor(3, 0x0000FF);
            addChild ( q );
         }
    }
}
```

Like we would do as usual, we listen to the `Event.ADDED_TO_STAGE` event and initialize our application in the event handler. That way, we can safely access the stage.

> Again, pay attention to this subtle detail. Our Game class here extends the Sprite class from the starling.display package, not the flash.display.package. A good practice is to always check your import statements and make sure you are not using the native API rather than the Starling APIs. You will get used to it quickly, but this can be confusing at the beginning.

By testing our previous code, we get the following result. Note that, as expected, objects, just like in Flash will have a default position of 0,0. Let's add a few lines to center our quad:

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

Note that shifting one bit to the right (>>1) is the equivalent of a division by 2, and here is our result:

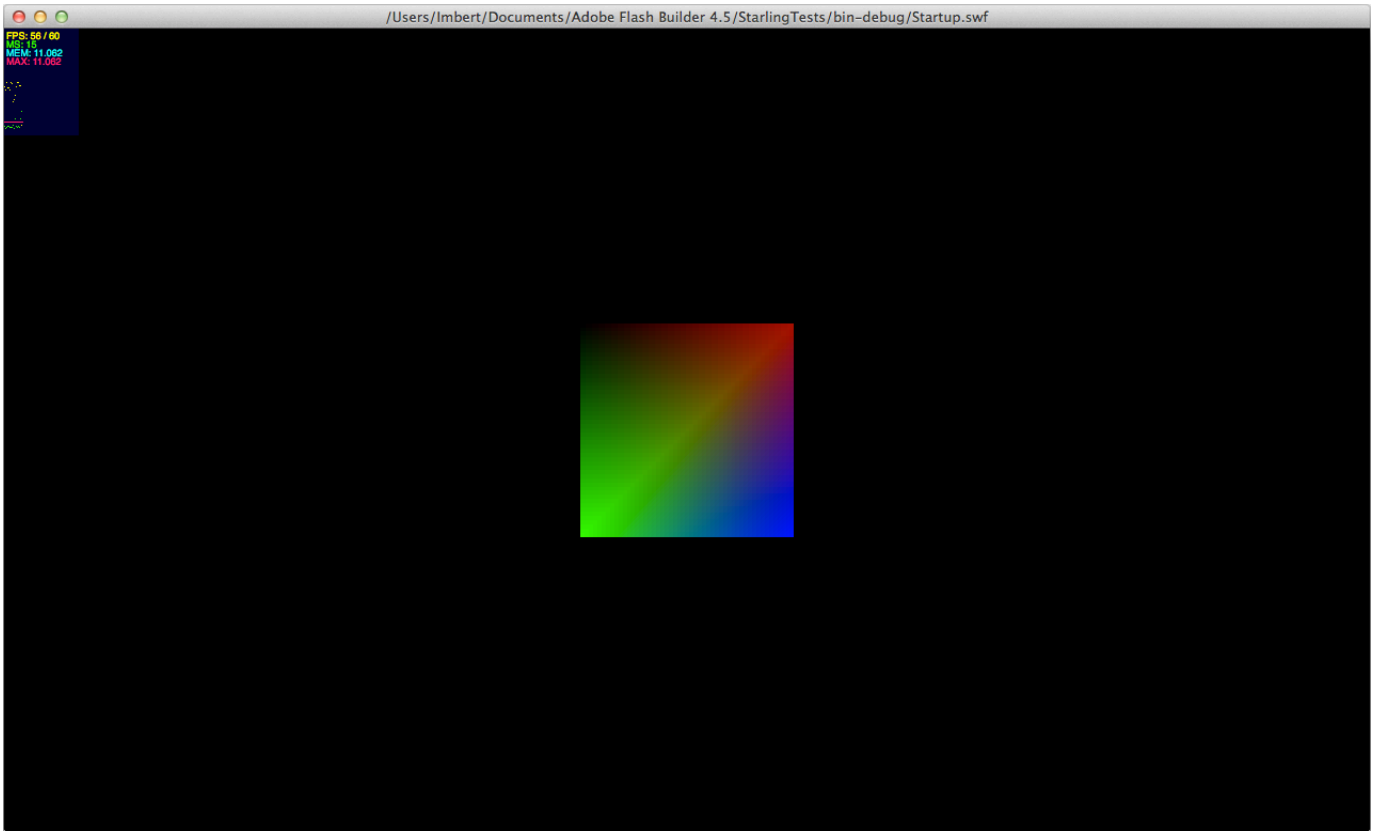

Figure 1.8

Our first quad !

Note that the antialiasing value allows you to set the anti-aliasing you want, generally a value of 1 is totally acceptable but you can go further, technically you can go from 0 to 16, but here is the list of the most common values:

- 0 : No anti-aliasing
- 2 : Minimal anti-aliasing.
- 4 : High quality anti-aliasing.
- 16 : Very high quality anti-aliasing.

We will rarely need to go above 2, especially for 2D content, but you will decide depending on your use cases. Below are some figures showing the slight difference between the 2 values (1 and 4):

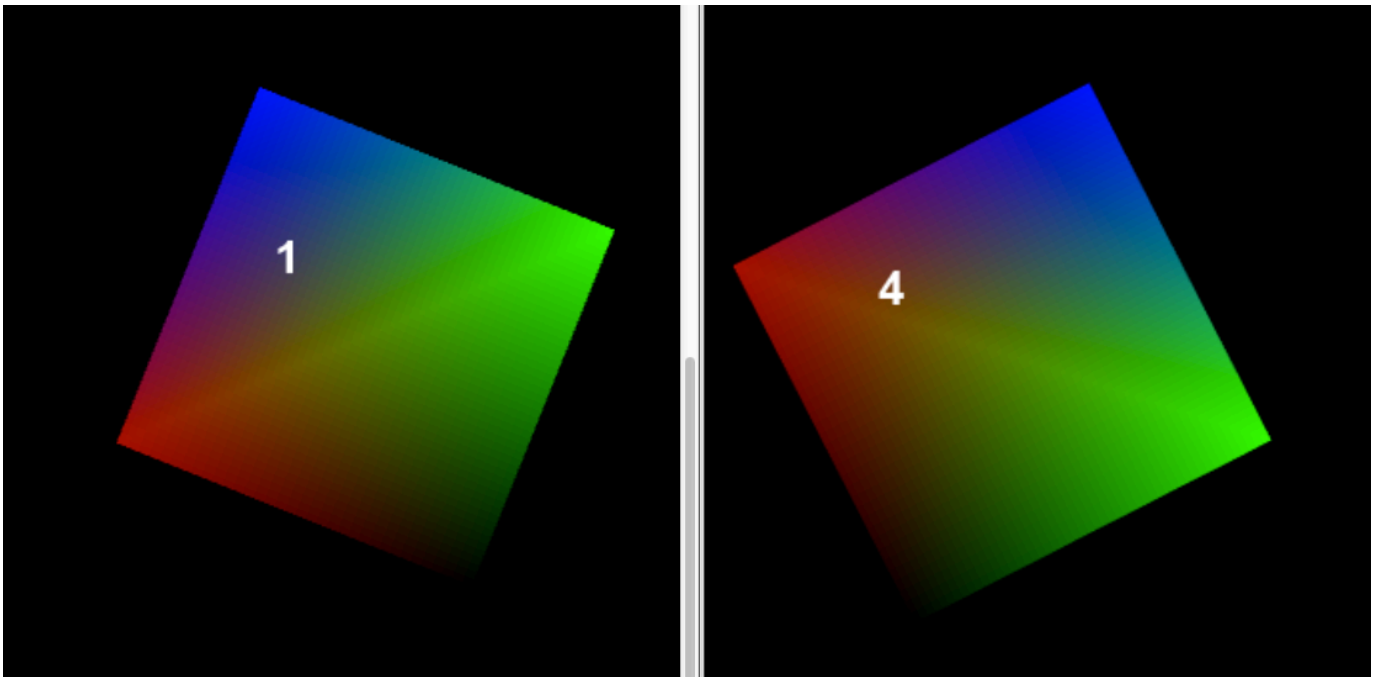

Figure 1.9

Anti-aliasing differences.

You can try values above 2 to adjust to the quality you want. Of course, choosing a high value will have an impact on the performance.

Let's have a quick look at the other APIs available on the Starling object:

- **enableErrorChecking** : Allows you to enable or disable error checking. Specifies whether errors encountered by the renderer are reported to the application. When enableErrorChecking is true, the clear(), and drawTriangles() methods called internally by Starling are synchronous and can throw errors. When enableErrorChecking is false, the default, the clear(), and drawTriangles() methods are asynchronous and errors are not reported. Enabling error checking reduces rendering performance. You should only enable error checking when debugging.
- **isStarted** : Indicates if start was called.
- **juggler**: A juggler is a simple object. It does no more than saving a list of objects implementing IAnimatable and advancing their time if he is told to do so (by calling its own advanceTime: method). When an animation is completed, it throws it away.
- **start** : Starts the rendering and event handling.
- **stop** : Stops the rendering and event handling, you would use this method to pause the rendering when the game goes into the background to save resources.
- **dispose** : Call this method, when you want to dispose the entire content being rendered and currently on the GPU memory. This API internally disposes the shader programs and listeners.

Once your `Starling` object has been created, a debug trace is outputted automatically, giving you informations about the renderer used. By default, when the SWF is correctly embedded in the page or when testing in standalone, Starling will output the following:

```
[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)
```

Of course, hardware details depend on your configuration. Note that it informs us here that we are using GPU acceleration by giving details about the drivers version. For debugging purposes, you may want to force the software fallback used internally by the Flash Player to get an idea of the performance of your content when running on software.

To do so, just inform that you want to use the software fallback (software rasterizer):

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

When using software, the message outputted confirms we are running in software mode:

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

You want to test your content in software too, to get an idea of the performance when some users will be running in this mode. Your content will fallback to software if the graphics card drivers have been issued before 1/1/2009. Now, let's have a look at the requirements for Stage3D when it comes to embedding your SWF in a page.

Wmode requirements

You have to remember that in order to enable Stage 3D and GPU acceleration you have to use `wmode=direct` as the embed mode in the page. If you do not specify any value or choose an other value than "direct", like "transparent", "opaque", or "window", Stage 3D will not be available, instead will get a runtime exception informing you that the creation of the `Context3D` object failed, when `requestContext3D` on `Stage3D` is called. Below is a figure illustrating the runtime exception dialog box:

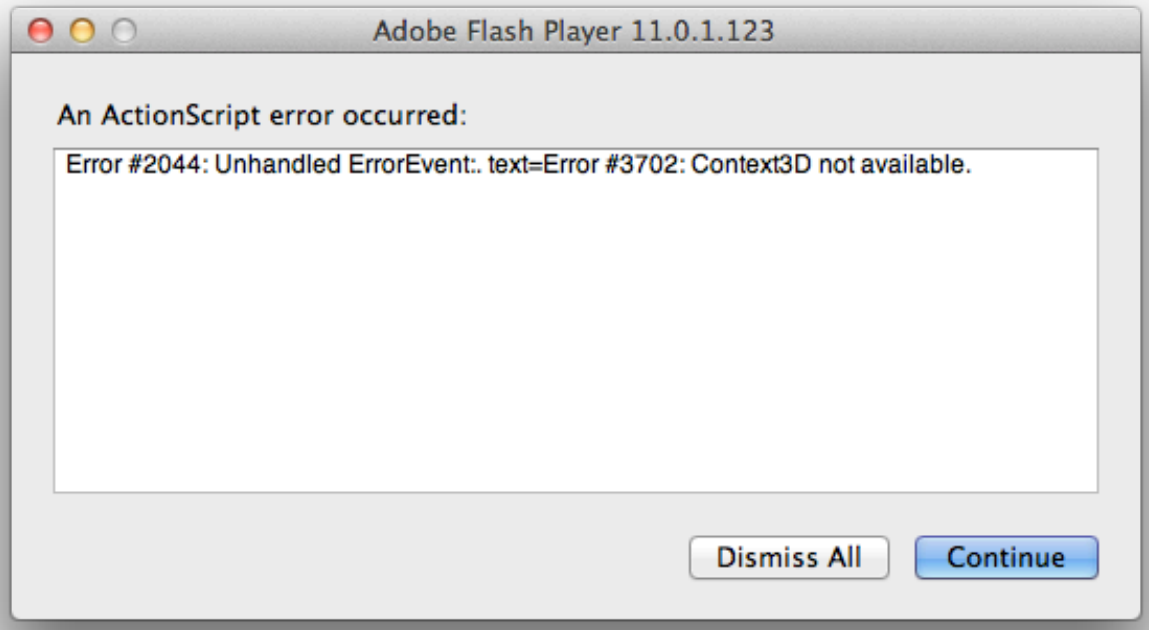

Figure 1.10

Runtime exception, when Context3D is not available.

It is important to handle this situation if your application is embedded using the wrong wmode. You need to react appropriately by displaying a message explaining the issue. Fortunately, Starling handles this automatically for you and display the message below:

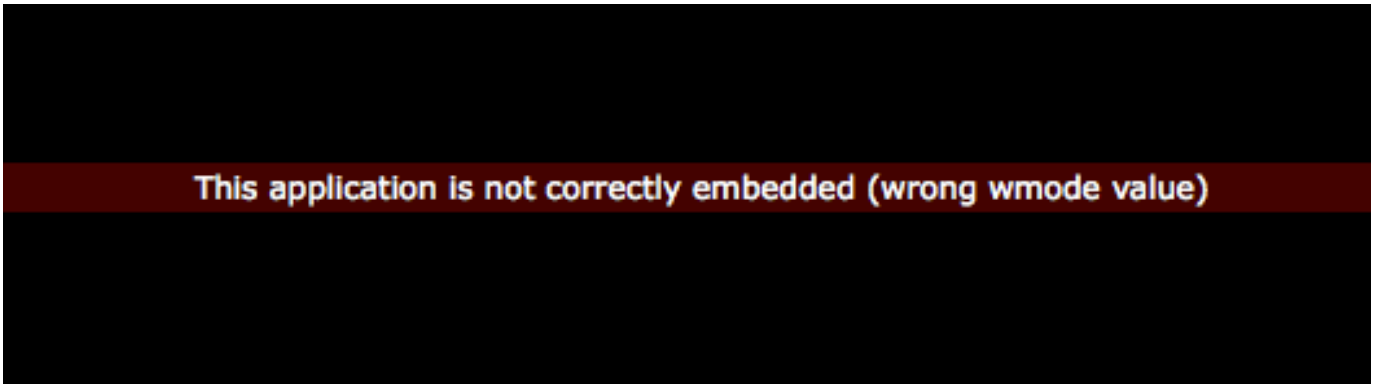

Figure 1.11

Warning message when the application is not correctly embedded.

Stage quality

As a Flash developer, the concept of stage quality is not new to you. Remember that when working with Stage3D, and as a result Starling, the stage quality has no impact on the performance.

Progressive enhancements

As mentioned previously, when GPU acceleration cannot be leveraged, Stage3D will fallback to software and use internally a software fallback engine, called SwiftShader (Transgaming). To make sure that your content runs well in such a scenario, you need to detect when you are running in software and remove potential effects that would be slow in software.

In the context of 2D content, software fallback should be able to handle many objects and provide good performance, but still, to detect this, you can access the `Context3D` object from the Starling object by using the static property `context`:

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

It is a good practice to always design your content with software fallback in mind, it will offer a progressive experience, ensuring the best experience possible on all configurations.

Let's have a look now at a pretty exciting topic, the display list in Starling!

The Display List

Just like with the native Flash display list, Starling follows the same rule, the stage is not available until objects are added to the display list. To access the stage safely, we usually rely on some of the most important events in Flash, which are also available in Starling:

- **Event.ADDED** : the object was added to a parent.
- **Event.ADDED_TO_STAGE** : the object was added to a parent that is connected to the stage, thus becoming visible now.
- **Event.REMOVED** : the object was removed from a parent.
- **Event.REMOVED_FROM_STAGE** : the object was removed from a parent that is connected to the stage, thus becoming invisible now.

We will actively rely on those events in the following examples, just like any Flash content, those events will allow us to initialize or deactivate objects and optimize performance and resources.

Below is the list of methods defined by the `DisplayObject` class:

- **removeFromParent** : Removes the object from its parent, if it has one.
- **getTransformationMatrixToSpace** : Creates a matrix that represents the transformation from the local coordinate system to another.
- **getBounds** : Returns a rectangle that completely encloses the object as it appears in another coordinate system.
- **hitTestPoint** : Returns the object that is found topmost on a point in local coordinates, or nil if the test fails.
- **globalToLocal** : Transforms a point from global (stage) coordinates to the local coordinate system.
- **localToGlobal** : Transforms a point from the local coordinate system to global (stage) coordinates.

Below is a list of the properties defined by the `DisplayObject` class, it is very pleasant to see all those properties exposed and some little improvements like `pivotX` and `pivotY` to dynamically change the registration point of a `DisplayObject`:

- **transformationMatrix** : The transformation matrix of the object relative to its parent.
- **bounds** : The bounds of the object relative to the local coordinates of the parent.
- **width** : The width of the object in points.
- **height** : The height of the object in points.
- **root** : The topmost object in the display tree the object is part of.
- **x** : The x coordinate of the object relative to the local coordinates of the parent.
- **y** : The y coordinate of the object relative to the local coordinates of the parent.
- **pivotX** : The x coordinate of the object's origin in its own coordinate space (default: 0).
- **pivotY** : The y coordinate of the object's origin in its own coordinate space (default: 0).
- **scaleX** : The horizontal scale factor. "1" means no scale, negative values flip the object.
- **scaleY** : The vertical scale factor. "1" means no scale, negative values flip the object.
- **rotation** : The rotation of the object in radians. (In Sparrow, all angles are measured in radians.)
- **alpha** : The opacity of the object.
- **visible** : The visibility of the object. An invisible object will be untouchable.
- **touchable** : Indicates if this object (and its children) will receive touch events.
- **parent** : The display object container that contains this display object.
- **stage** : The stage the display object is connected to, or null if it is not connected to a stage.

Just like with the native Flash APIs, a `Sprite` is the most lightweight container you can use. Of course as a subclass of `DisplayObject` you get all the APIs mentioned earlier plus the ability to nest content. So far we did not nest content, except for the scene, actually, remember that our `Game` class extends `Sprite`, which is a `DisplayObjectContainer`.

Below are the APIs exposed by `DisplayObjectContainer` objects:

- **addChild** : Adds a child to the container. It will be at the topmost position.
- **addChildAt** : Adds a child to the container at a certain index.
- **dispose** : Removes the GPU buffers and all the listeners registered to the object.
- **removeFromParent** : Removes the child from its parent.
- **removeChild** : Removes a child from the container. If the object is not a child, nothing happens.
- **removeChildAt** : Removes a child at a certain index. Children above the child will move down.
- **removeChildren** : Removes all children from the container.
- **getChildAt** : Returns a child object at a certain index.
- **getChildByName** : Returns a child object with a certain name (non-recursively).
- **getChildIndex** : Returns the index of a child within the container.
- **setChildIndex** : Changes the index of the specified child.

- **swapChildren** : Swaps the indexes of two children.
- **swapChildrenAt** : Swaps the indexes of two children.
- **contains** : Determines if a certain object is a child of the container (recursively).

Once you have access to the stage you can then call most of the `DisplayObjectContainer` APIs on it, but you can also pass a color to the stage. By default Starling will take the default SWF background color, that you can set using the following SWF tag:

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

You can also override this behavior and just pass a color to the `stage` object that you can access from any `DisplayObject` added to the display list:

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

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

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

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

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

Now, understand that we are not using any texture, we basically have two triangles grouped as a quad, and each vertex of our plane has a different color being interpolated on the GPU.

Of course, if you want a solid plain color, just use the `color` property of the `Quad` object:

```
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 );
         }
    }
}
```

Then you will end up with this:

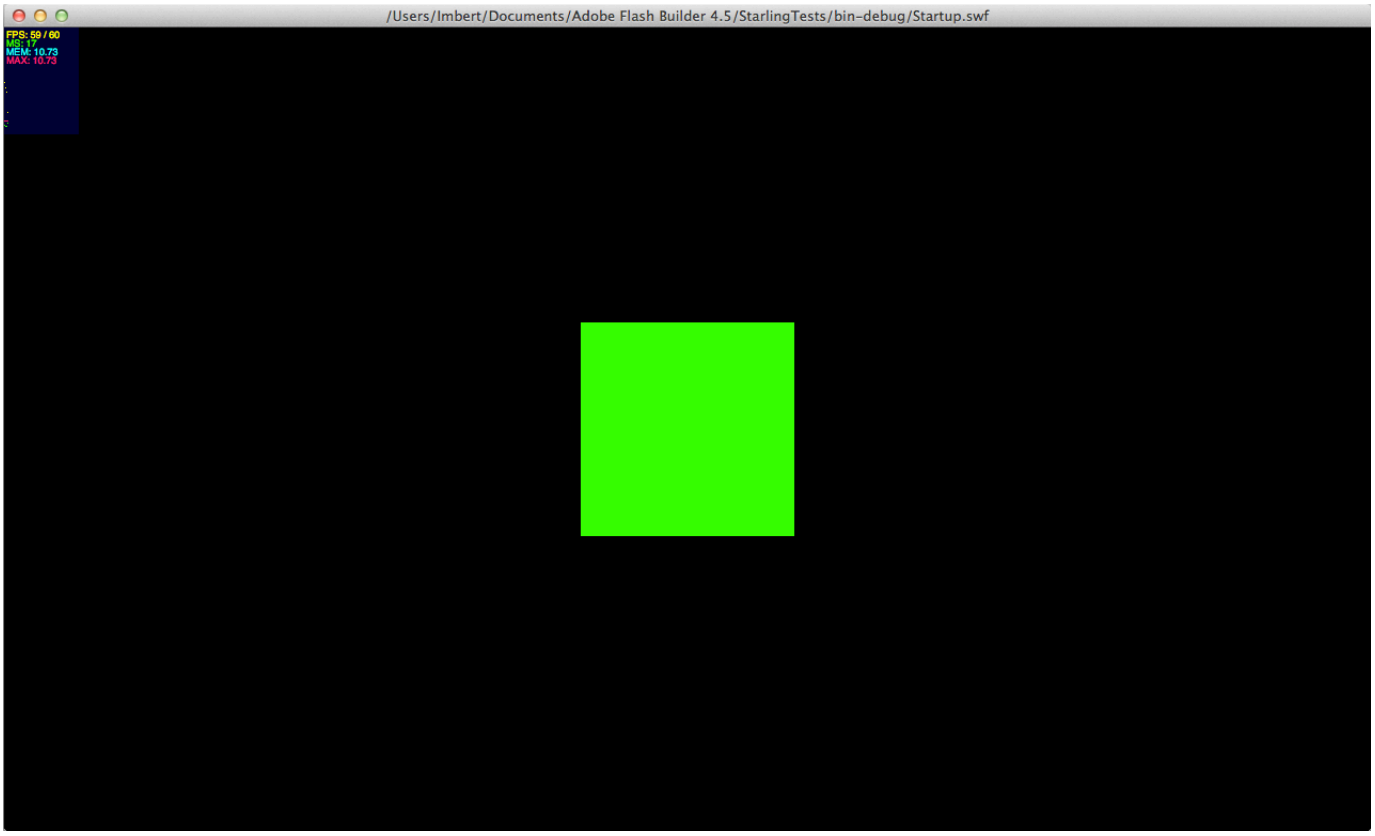

Figure 1.12

A solid green quad.

We will be now using an `Event.ENTER_FRAME` event, this handler will interpolate the quad color using a simple easing effect between random colors:

```
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;
         }
    }
}
```

To rotate this quad, we can use the `rotation` property, note that Starling works in radians whereas Flash Player works with degrees. This choice was made to preserve consistency between Sparrow and Starling. Anytime you want to apply a rotation using degrees, just use the `starling.utils.deg2rad` function or just inline the conversion:

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

One neat thing is that all `DisplayObject` have `pivotX` and `pivotY` properties allowing us to move the registration point at runtime for any object:

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

It feels very natural for an ActionScript developer to use Starling, this quad as a `DisplayObject` can then be nested with a `TextField` inside a `Sprite`, and moving this sprite will allow us to group elements together, just like what you would do with the native display list:

```
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;
           }

     }
}
```

We now rotate our sprite containing our `Quad` and a `TextField` all this rotating around the container registration point:

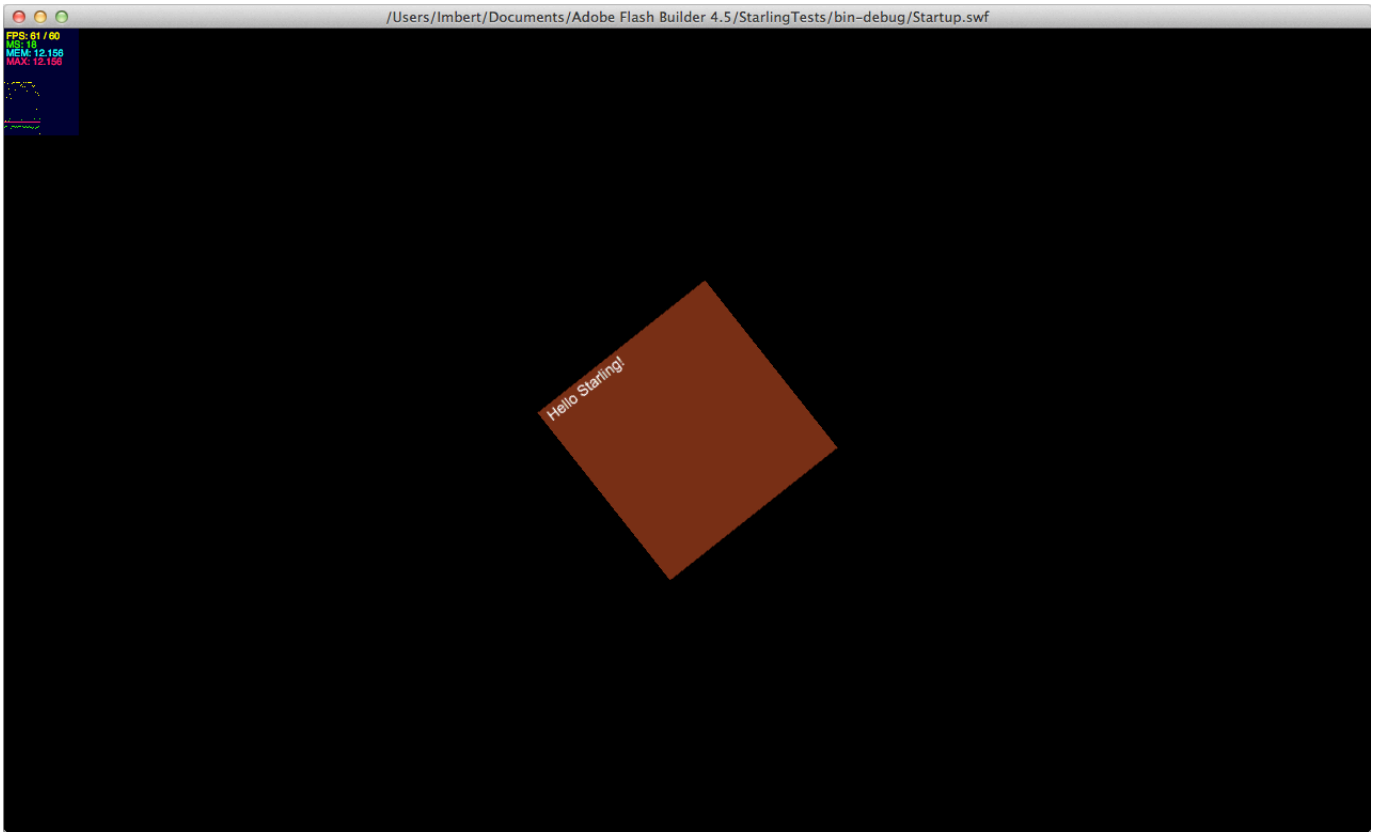

Figure 1.13

A Sprite containing a Quad and a TextField.

Our code starts to look messy, let's move some code to define a `CustomSprite`, which will encapsulate the internal color behavior and children. Here is our final `CustomSprite` class:

```
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;
        }
    }
}
```

And this is our `Game` class:

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

Note that we update our custom sprite using the update API on `CustomSprite` allowing us to have the main loop in the game class allowing us to control our object from a central place. By using this approach on our other elements, it becomes trivial then to add a pause mechanism to our entire set of content.

Let's add a little more interaction to our little test, we are going to add some more movement by having our quad to follow the mouse. In the code below we add a little piece of code to make it happen:

```
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;
            }
        }
    }
```

Note that here, we do not use any `Mouse` API, and actually there is no concept of mouse in Starling, we will come back to this very soon.

By listening to the `TouchEvent.TOUCH` event, we are actually listening to any mouse/finger movement, just like a classic `MousEvent.MOUSE_MOVE`. Every frame, we store the current mouse location by using the helper APIs from the `TouchEvent` object like `getTouch` and `getLocation`. Once the mouse position is stored, we use a little easing equation inside the `onFrame` event handler to move our quad around.

As mentioned previously, Starling not only makes your life easier to program on the GPU, but also when it comes to disposing objects for resources. Let's say we want to remove this quad from the scene when we click on it, we will then write the following:

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

Note that here we remove the child but we do not remove the `Event.ENTER_FRAME` listener. We can test if our sprite still has a listener by using the `hasEventListener` API:

```
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) );
    }
}
```

To safely remove a child, use the second parameter `dispose` of the `removeChild` API, which allows you to also remove automatically all the listeners registered to the object being removed:

```
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) );
    }
}
```

If the child has children, all of them will be disposed too. The same `dispose` argument is available also on the other APIs to remove children like `removeChildren` or `removeChildAt`. Note that the dispose behavior also clears the GPU buffers for the object, but not the texture. To dispose the texture, you can call the same `dispose` method on the `Texture` or `TextureAtlas` object.

You can also remove all the listeners by calling the `dispose` API explicitely on any `DisplayObject`:

```
clicked.dispose()
```

We have been using for the first time, the Starling event model which looks very similar to the native Flash Player's one, let's spend some more time on the `EventDispatcher` API available in Starling.

Event model

As shown in figure 1.2, all Starling objects are subclasses of the `EventDispatcher` class. Just like with the native `EventDispatcher` API, all Starling objects expose APIs to add or remove listeners to them:

- **addEventListener** : Registers a listener to a specific event.
- **hasEventListener** : Tests if there is a listener for a specific event.
- **removeEventListener** : Removes the event listener.
- **removeEventListeners** : Removes all the listeners registered to a specific event or all of them.

Note the addition of a very useful API: `removeEventListeners`. At any time, when you need to remove all listeners registered to a specific event, use `removeEventListeners` by passing the event type:

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

When you need to remove all the listeners, no matter the event, in a scenario where you are about get rid of an object or deactivate it, just call the same `removeEventListeners` API with no event type as a parameter:

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

Note that we just used recently the `removeChild` API that takes an argument for disposing listeners, which calls internally the same API internally, but on each child.

Event propagation

As we have seen since the beginning of this Starling tutorial, Starling recreates the concept of display list on top of `Stage3D`. The great news is that you will be able to also reuse the power of event propagation with Starling. Event propagation can be really useful in some scenarios to limit the number of listeners that you have to register and unregister and make your code cleaner.

For those of you not familiar with the concept of event propagation, you can get more details about it here: http://www.adobe.com/devnet/actionscript/articles/event_handling_as3.html

As an interesting detail, Starling handles event propagation, but a slightly different version than the native one in Flash. Starling only supports the bubbling phase, there is no concept of capture phase. We will be leveraging event propagation during the next examples to see how it works.

Touch Events

As we mentioned earlier, Starling is Sparrow's cousin, as a result, the touch event mechanism is Starling is really tailored for mobile and as a result for touch interactions, which can be quite confusing at first sight when using Starling on desktop applications designed for mouse interactions.

First, if you take a look at the figure 1.2, you will notice that in contrary of the native display list, with Starling there is no `InteractiveObject` class in the hierarchy, all display objects are by default interactive. To say it differently, the `DisplayObject` class defines interactive behaviors.

We have been using touch events quickly in the past examples. We started with some very basic stuff, like reacting when the mouse touches the quad. For this, we used the `TouchEvent.TOUCH` event:

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

You may think that this is pretty limited right? Actually it is very powerful cause you can detect a lot of different states through this single event. Everytime a mouse or fingers are interacting with a graphical object, a `TouchEvent.TOUCH` event is dispatched.

Let's have a closer look; in the code below, we trace the `phase` property available on the `Touch` object in our `onTouch` event handler:

```
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;
}
```

When we start interacting with the quad and click on it, we can see that different phases are triggered; here is the list of all the phases available as constants through the `TouchPhase` API:

- **began** : A mouse or finger starts interacting (similar to a mouse down state).

- **ended** : A mouse or finger stop interacting (similar to a native click state).
- **hover** : A mouse or finger is hovering an object. (similar to a native mouse over state)
- **moved** : A mouse or finger is moving an object (similar to a native mouse down state + a mouse move state).
- **stationary** : A mouse or finger stopped interactng with an object and stays over it.

Let's have a look at some others API available on the `TouchEvent` event object:

- **ctrlKey** : A boolean returning the state of the ctrl key (down or not).
- **getTouch**: Gets the first Touch object that originated over a certain target and are in a certain phase.
- **getTouches** : Gets a set of Touch objects that originated over a certain target and are in a certain phase.
- **shiftKey**: A boolean returning the state of the shift key (down or not).
- **timestamp** : The time the event occurred (in seconds since application launch).
- **touches** : All touches that are currently happening.

The use of the `shiftKey` and `ctrlKey` properties is useful to detect combination with keyboard keys. So everytime there is an interaction, the finger or the mouse has a `Touch` object related to it.

Let's see the APIs available on a `Touch` object:

- **clone** : Clones the object.
- **getLocation**: Converts the current location of a touch to the local coordinate system of a display object.
- **getPreviousLocation**: Converts the previous location of a touch to the local coordinate system of a display object.
- **globalX**: The x-position of the touch in screen coordinates.
- **globalY** : The y-position of the touch in screen coordinates.
- **id**: A unique id for the object.
- **phase** : The current phase the touch is in.
- **previousGlobalX** : The previous x-position of the touch in screen coordinates.
- **previousGlobalY** : The previous y-position of the touch in screen coordinates
- **tapCount** : The number of taps the finger made in a short amount of time. Use this to detect double-taps, etc.
- **target** : The display object at which the touch occurred.
- **timestamp** : The moment the event occurred (in seconds since application start).

Simulating multi-touch

When developing content for mobile devices, there are lof of chances that you may want to leverage multi-touch interactions, like for scaling for instance. When authoring on desktop, if you cannot test live on the device, Starling offers a great built-in mechanism to simulate multi-touch.

To enable it, you need to use the `simulateMultiTouch` property on the `Starling` object:

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

Once enabled, just use the ctrl key, automatically, two little dots will appear and simulate multi touch inputs. The figure below illustrates the idea:

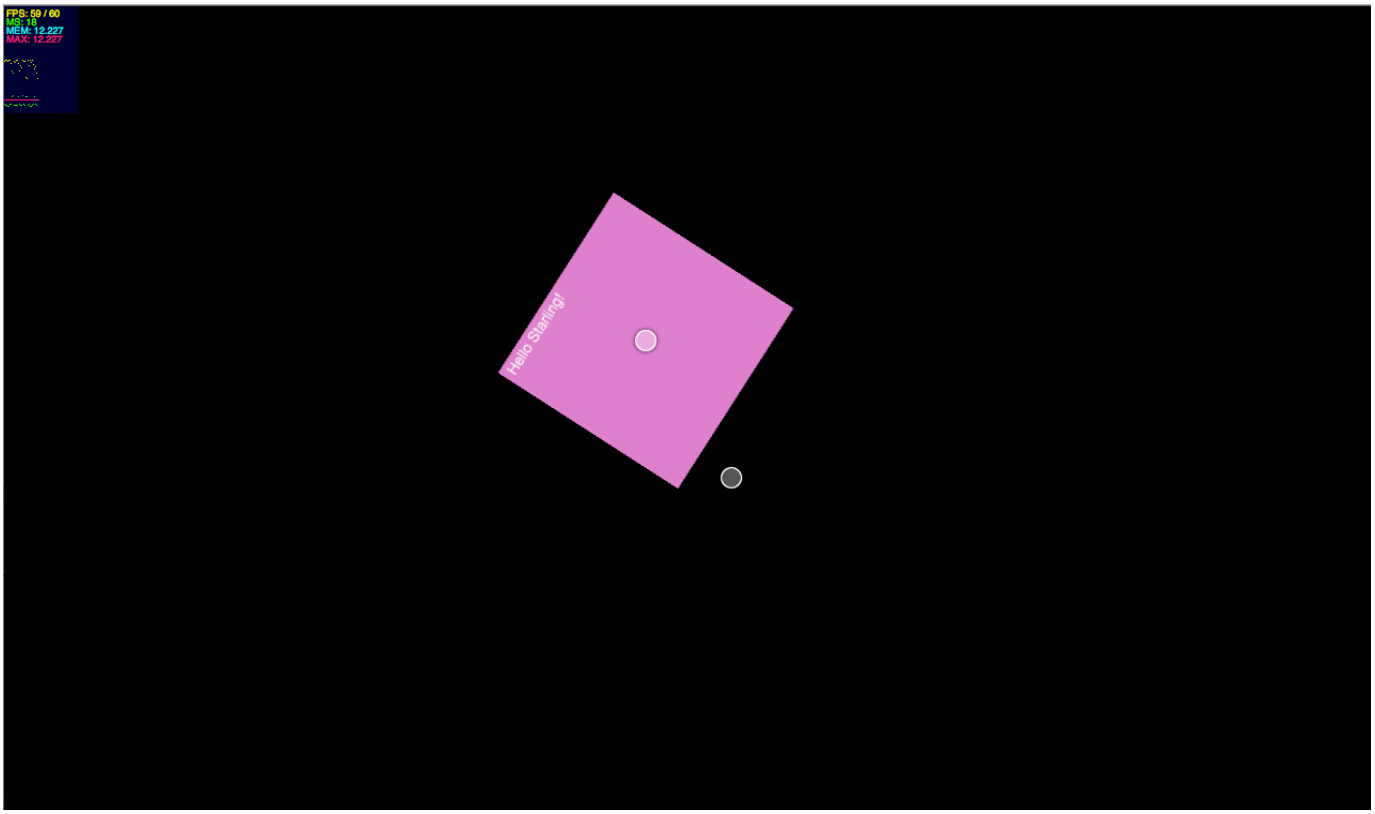

Figure 1.14

Multi-touch simulation.

In the code below, we are scaling the quad when using multiple touch points, like two fingers. We retrieve the touch points and calculate the distance between them:

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

Let's cover textures now!

Texture

A texture object must be created to feed an `Image` object. Think of it, as the relationship between `Bitmap` and `BitmapData` when using the native APIs. At anytime, when you need to create a `Texture` object, you need to use the `Texture` API, which contains the following APIs:

- **base** : The Stage3D texture object the texture is based on.
- **dispose** : Disposes the underlying texture data.
- **empty** : Returns a Texture object out of dimensions (width and height).
- **frame** : The texture frame (see class description).
- **fromBitmap** : Returns a Texture object out of a Bitmap object. This Bitmap object can be embedded or loaded dynamically.
- **fromBitmapData** : Returns a Texture object out of a BitmapData object.
- **fromAtfData** : Allows the use of a compressed texture using the ATF (Adobe Texture Format). Compressed textures allows you to save a lot of memory especially on constrained environments like mobile devices.
- **fromTexture** : Allows the use of a texture and returns a new texture.
- **height** : The height of the texture in pixels.
- **mipmapping** : Indicates if the texture contains mip maps.
- **premultipliedAlpha** : Indicates if the alpha values are premultiplied into the RGB values.
- **repeat** : Indicates if the texture should repeat like a wallpaper or stretch the outermost pixels.
- **width** : The width of the texture in pixels.

Different image formats can be used for your textures. The list below summarizes the various formats that can be used for your textures:

- **PNG** : As alpha channel is often required, PNG is one of the most common file format used for textures.
- **JPEG** : The classic JPEG format can also be used. Remember that on the GPU the image will be decompressed, so using JPEG will not limit the memory usage and you will not be able to use transparency in your textures.
- **JPEG**-XR : JPEG XR (abbr. for JPEG extended range[4]) is a still-image compression standard and file format for continuous tone photographic images, based on technology originally developed and patented by Microsoft under the name HD Photo (formerly Windows Media Photo). It supports both lossy and lossless compression, and is the preferred image format for Ecma-388 Open XML Paper Specification documents.
- **ATF** : Adobe Texture Format. This is the best file format for the best compression. ATF files are primarily a file container to store lossy texture data. It achieves its lossy compression through to the use of two common techniques: JPEG-XR1 compression and block based compression. JPEG-XR compression provide a competitive method to save storage space and network bandwidth. Block based compression provides a way to reduce texture memory usage on the client, at a fixed ratio of 1:8 compared to RGBA textures. ATF supports three types of block based compression: DXT12, ETC13 and PVRTC4.

Let's dig a little more on the concept of textures and discover an essential concept for images on the GPU, mipmapping. Mipmapping is an important and easy concept to understand. Scaled down versions of a texture are called mipmaps. When working with textures on the GPU, the latter needs to scale images sometimes depending on the size of the content. This can happen if your camera is moving towards the content, or when the content is moving towards the camera. Both scenarios will scale your content and as a result your textures.

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.

To ensure the best quality, the GPU requires all mipmap levels of the image (mipmaps). Which means all the versions of the image from its original size (which needs to be a power of two) to 1. Without Starling you would have to manually generate the miplevels by using a very simple trick involving `BitmapData.draw` and a scaled by 2 transformation matrix.

> 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.

Fortunately, as mentioned earlier, Starling does it for you automatically, and here is the code used by Starling, which generates the miplevels:

```
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();
 }
```

When using the ATF format (*Adobe Texture Format*) you actually do not need to worry about it, the ATF file format contains already the miplevels, which are not generated at runtime but ahead of time. This is valuable cause it saves the time you would spend generating those miplevels. In a scenario with a lot of textures this can be precious time saved and bring faster initialization of your content.

Notice that there is also a `frame` property on the `Texture` object, this allows us to define the position of the texture when assigned to an `Image` object. Let's say you would like to have some borders around your image, you would then use a smaller texture than the image and position the texture in the center of the image:

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

Talking about the `Image` object, let's have a closer look at it.

Image

In Starling, a `starling.display.Image` object is the equivalent of a native `flash.display.Bitmap` object:

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

To display an image, you then need to create an `Image` object and pass a `Texture` object to it:

```
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;
              }
         }
    }
}
```

Note that we used the static `fromBitmap` API on the `Texture` class to generate our `Texture` object.

If we test our code, this should gives us the following result:

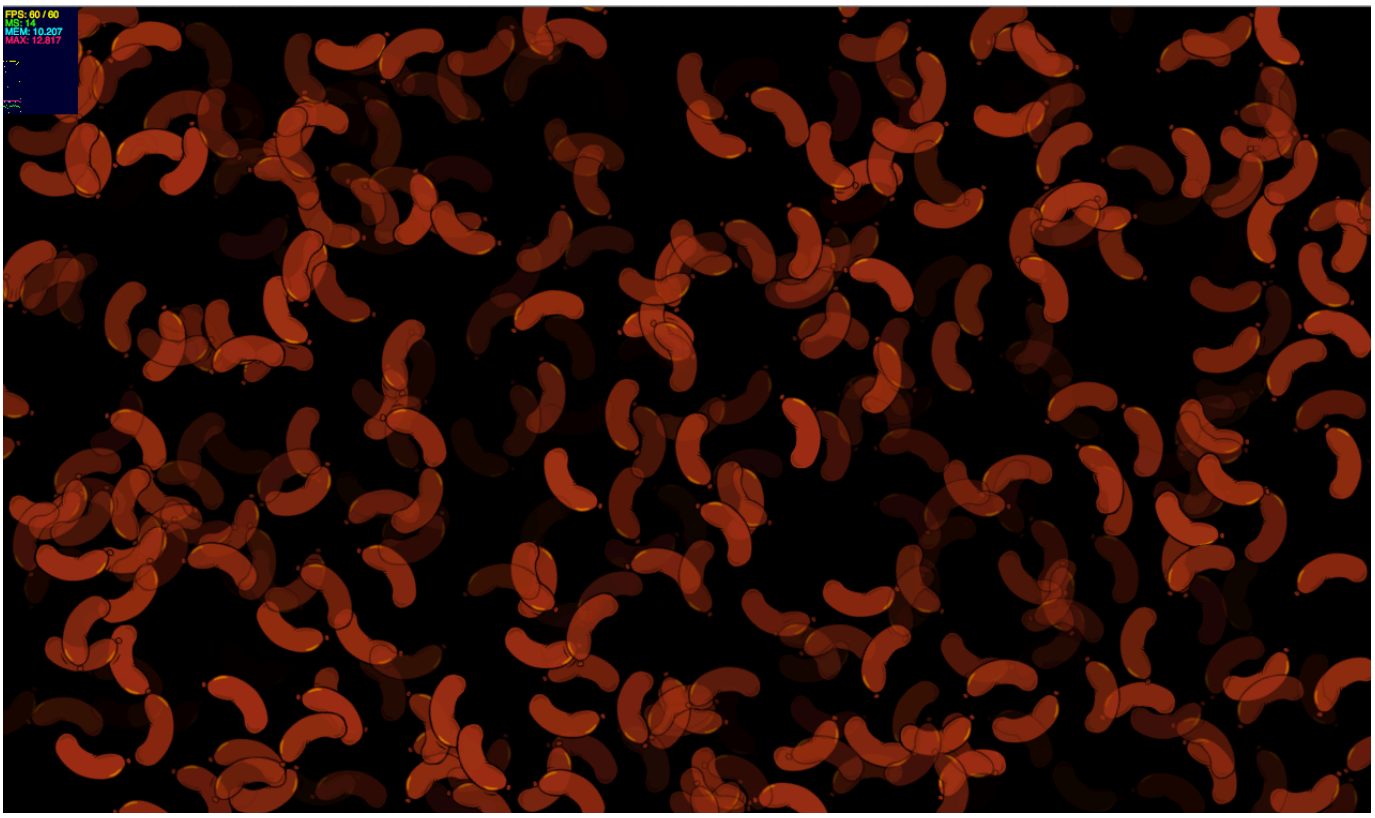

Figure 1.15

Our sausages positioned randomly.

The bitmap we use here is embedded but could be loaded dynamically. To achieve this, we would use a `Loader` object, to load our texture, then retrieve the loaded `Bitmap` and use the `fromBitmap` API to generate our Starling texture:

```
// 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 )
}
```

Another API is available to generate a `Texture` object out of a `BitmapData` object, but we will come back to this later.

Note that we are reusing here the same texture for all our sprites on screen. If you look at the isolated code below, for each iteration, we reuse the one and only texture that we initially created outside of the loop:

```
// 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);
```

A bad practice would be to recreate the texture for each iteration, like the following:

```
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()));
```

This would have a bad impact on the memory cause multiple copies of the same bitmap for the texture are allocated, but also a general impact on the performance of your content, cause multiple copies of the same image would be uploaded to the GPU, which would be inefficient. Last but not least, the performance of your loop would be affected, given that for each call to `fromBitmap`, you will also generate the mipmaps.

Let's make things move now, for this, lets create a `CustomImage` class:

```
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);
         }
    }
}
```

Then let's use this `CustomImage` class in our code:

```
package
{
    import flash.display.Bitmap;

    import starling.display.Sprite;
    import starling.events.Event;
    import starling.textures.Texture;
    import starling.utils.deg2rad;

    public class Game2 extends Sprite
    {
         private var sausagesVector:Vector.<CustomImage> = new Vector.<CustomImage>(NUM_SAUSAGES, true);

         private const NUM_SAUSAGES:uint = 400;

         [Embed(source = "../media/textures/sausage.png")]
         private static const Sausage:Class;

        public function Game2()
        {
              addEventListener(Event.ADDED_TO_STAGE, onAdded);
```

```
        }

         private function onAdded (e:Event):void
         {
              // create a Bitmap object out of the embedded image
              var sausageBitmap:Bitmap = new Sausage();

              // create a Texture object to feed the Image object
              var texture:Texture = Texture.fromBitmap(sausageBitmap, false);

              for (var i:int = 0; i < NUM_SAUSAGES; i++)
              {
                   // create a Image object with our one texture
                   var image:CustomImage = new CustomImage(texture);
                   // set a random alpha, position, rotation
                   image.alpha = Math.random();

                   // define a random destination
                   image.destX = Math.random()*stage.stageWidth;
                   image.destY = Math.random()*stage.stageWidth;

                   // define a random initial position
                   image.x = Math.random()*stage.stageWidth
                   image.y = Math.random()*stage.stageHeight
                   image.rotation = deg2rad(Math.random()*360);

                   // show it
                   addChild(image);

                   // store references for later
                   sausagesVector[i] = image;
              }

              // main loop
              stage.addEventListener(Event.ENTER_FRAME, onFrame);
         }

         private function onFrame (e:Event):void
         {
              var lng:uint = sausagesVector.length;

              for (var i:int = 0; i < lng; i++)
              {
                   // move the sausages around
                   var sausage:CustomImage = sausagesVector[i];
                   sausage.x -= ( sausage.x - sausage.destX ) * .1;
                   sausage.y -= ( sausage.y - sausage.destY ) * .1;

                   // when reached destination
                   if ( Math.abs ( sausage.x - sausage.destX ) < 1 &&  Math.abs ( sausage.y - sausage.destY )
)
                   {
                        sausage.destX = Math.random()*stage.stageWidth;
                        sausage.destY = Math.random()*stage.stageWidth;
                        sausage.rotation = deg2rad(Math.random()*360);
                   }
              }
         }
    }
}
```

As covered previously, Starling handles event propagation also, making our code cleaner to catch touch events from all those images moving around. For this, we will just listen to the `TouchEvent.TOUCH` on the stage:

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

If we test the target and `currentTarget` properties, we see that the object dispatching the event (`currentTarget`) is the `Stage` and the object initiating the event is a `CustomImage` instance:

```
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 );
         }
    }
}
```

By using this approach, we do not have to register a listener to each image. We just listen to the event on their container, here the stage and catch the event during the bubbling phase. If we use the `bubble` property from the `TouchEvent` object, we see that the event is bubbling:

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

OK, that's it for the event propagation mechanism. Let's have a look at the `Image` object and what capabilities it offers. As expected, the `Image` object exposes all the APIs herited from `DisplayObject`, plus one specific `smoothing` property to handle image smoothing. The following values are allowed, stored as constants in the `TextureSmoothing` class:

- **BILINEAR** : Applies a bilinear filter to the texture when scaled. (default)
- **NONE** : Applies a bilinear filter to the texture when scaled.
- **TRILINEAR** : Applies a trilinear filter to the texture when scaled.

Which works like this:

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

Below an image being scaled using a bilinear filter (`TextureSmoothing.BILINEAR`):

Figure 1.16
TextureSmoothing.BILINEAR.

Below an image being scaled using a trilinear filter (`TextureSmoothing.TRILINEAR`):

Figure 1.17
TextureSmoothing.TRILINEAR.

Below an image being scaled not using any filter (`TextureSmoothing.NONE`):

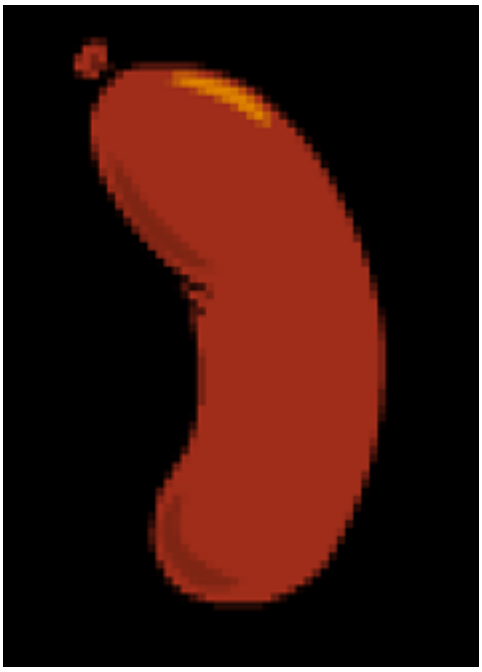

Figure 1.18
TextureSmoothing.NONE.

Note how cool this one looks when no filtering applied.

Keep in mind that the `Image` object exposes a `color` property allowing you to specify a color value. For each pixel, the final color will be the result of the multiplication of the color of the texture with the color you specified. This allows you to tint an image easily, and create variations of an image without having to use different textures.

The figure below illustrates the idea:

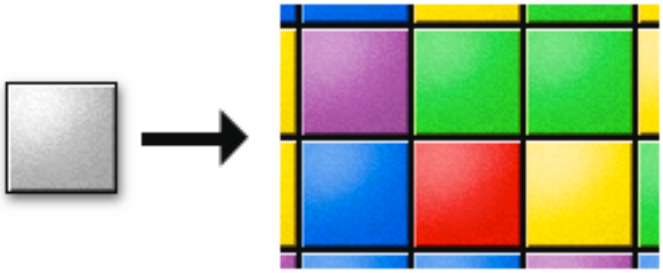

Figure 1.19
Texture tinted by the quad color (figure courtesy of sparrow-framework.org).

What if you wanted to use dynamic custom shapes with Starling? No problem, check the following section.

Collision detection

In most games, when not relying on a physics engine (like Box2D, that we will cover later), you may want to handle simple hit detection. When working with shapes like circles, a simple test checking if the distance between the two points is less than the sum of the radii will be enough. In some other scenarios, we can just do a simple test between the bounding boxes, but what about pixel perfect collision?

The figure below illustrates a typical scenario, where we need to detect the collision between two objects, containing transparent areas:

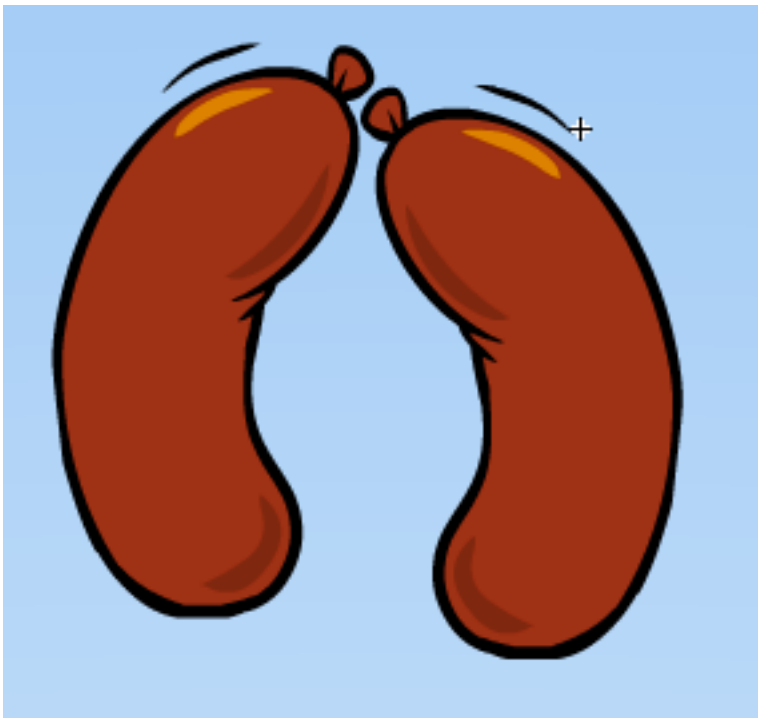

Figure 1.20
Pixel perfect collision.

In this scenario, we have curves, and we really want to detect the collision at the pixel level. Of course, to perform pixel-level hit detection; we will be using transparent images.

It could be actually pretty expensive to test pixel detection by executing ActionScript code. Fortunately, given that Starling relies on the native `BitmapData` API to create textures, we can use the native `hitTest` API from the `BitmapData` object.

The `hitTest` API is really simple to use but can be confusing at first sight, here is the signature of the API:

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

Note that the `secondObject` parameter can be a `Point`, a `Rectangle` or a `BitmapData` object, making the API pretty useful for many other scenarios. We will be using two `BitmapData` objects in our example, which is very likely to be a common scenario with Starling, because of the use of `BitmapData` for textures used by the `Image` object:

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

We first pass a point, being the location of the second object we want to test the collision against, an alpha threshold, allowing us to specify which pixels values are considered opaque (you might be using 255 or 0xFF) most of the time. Finally, we need to pass the second point, being the location of the current image, and then the second alpha threshold.

Because we might be calling this hit detection test one each frame, we want to minimize the garbage collector load. In our previous code, we create `Point` objects when performing the hit test. A better approach would be to update the x and y properties of the `Point` objects so that we do not have to allocate them on every frame:

```
private function onFrame(event:Event):void
{
    point1.x = sausageImage1.x;
    point1.y = sausageImage1.y;
    point2.x = sausageImage2.x;
    point2.y = sausageImage2.y;

    if ( sausageBitmapData1.hitTest(point2, 255, sausageBitmapData1, point1, 255))
    {
       trace("touched!");
    }
}
```

Let's have a look now, at how we can use the software native drawing API with Starling!

Drawing API

Starling does not expose a drawing API like you have today natively in the native `flash.display.Graphics` object. However, it is easy to replicate such a feature by using the drawing API then draw this inside a `BitmapData` object and use it as a texture.

Let's say you want to use a star shape and display it using Starling, you would write the following code:

```
// 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);
```

The idea is simple, you use the traditional native `Graphics` API to draw your lines, strokes, and fills, using the CPU, then you rasterize that on a bitmap and upload this as a texture.

The figure below illustrates an example, where circles are being used with Starling:

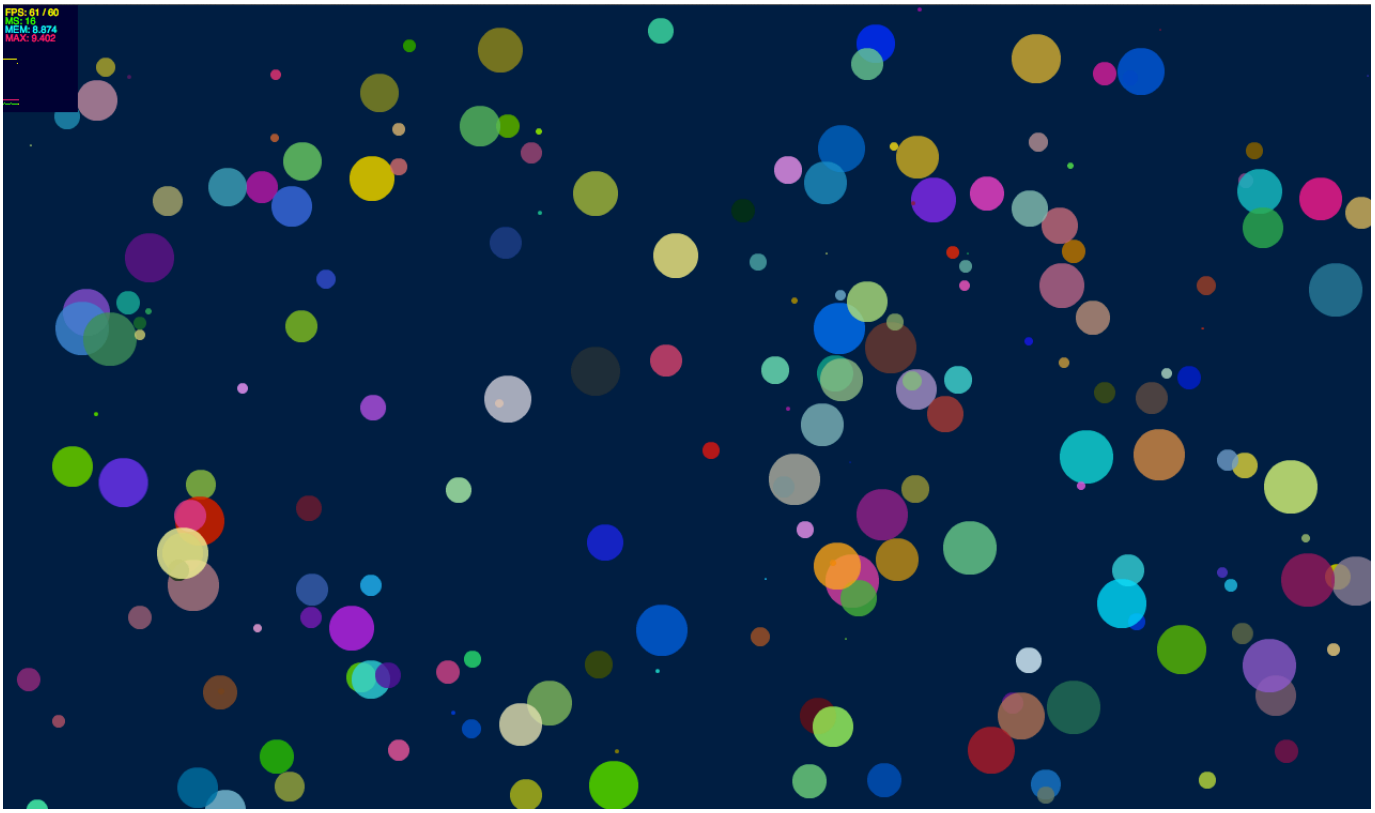

Figure 1.21

Custom dynamic shapes.

Did you know about flat sprites in Starling? So let's check the following section and see how we can improve performance by using this very powerful feature.

Flat Sprites

Starling contains a very powerful feature called flat sprites (compiled sprites in Sparrow) which allows you to bring great performance improvements.

To better understand how the display list works by default, let's have a look at the figure below which illustrates a typical display tree in an application with many different objects nested together:

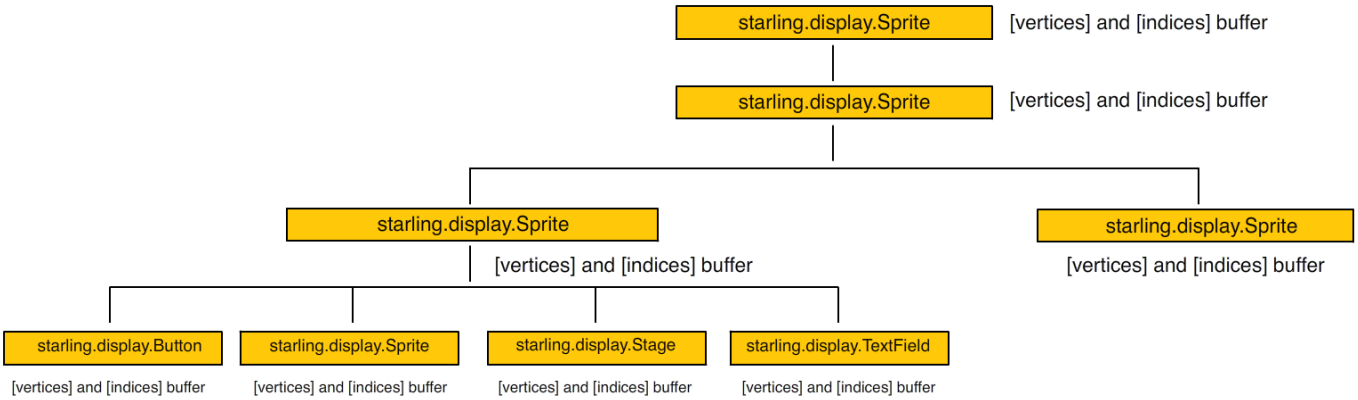

Figure 1.22

Children having their own vertex and index buffer.

As you can see on the figure above, Starling has to handle each child with its own vertex and index buffer and all the children behaviors independently, which can require a lot of computation and in some cases alter your performance.

What Starling can do, is gather all the children's geometry to a single big vertex buffer and draw the entire content (the container and its children) in one draw call, like a simple texture (if children share the same texture of course):

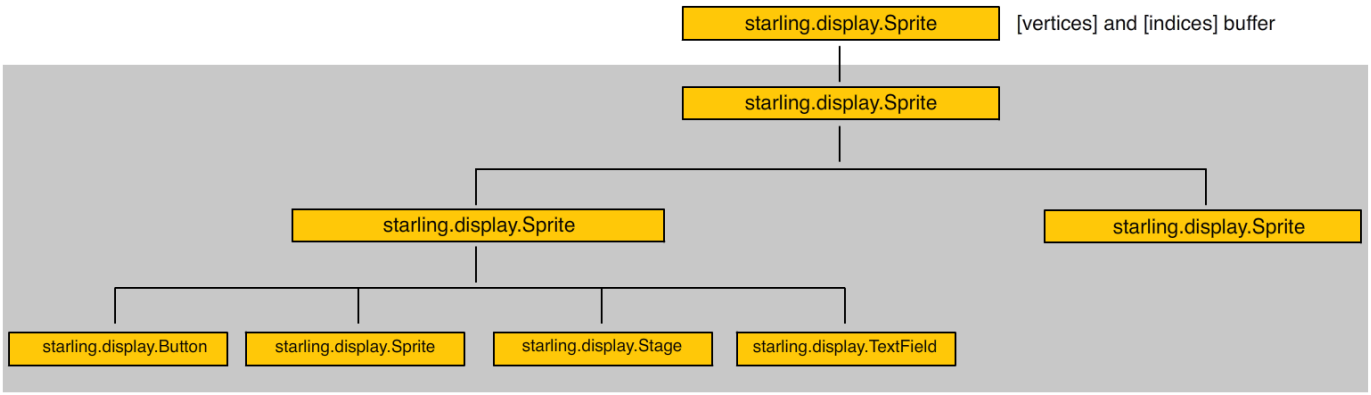

Figure 1.23

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

You can think about it as a similar approach as the `cacheAsBitmap` (bitmap caching) feature natively supported in the native dispay list. The exception (which is key here) is that the surface being drawn does not get renerated automatically when changing a child in the tree. You will have to call explicitely the flatten API to see the changes.

Here is a list of the APIs available:

- **flatten**: Call flatten when you want to render content as fast as possible. Once called, Starling gathers all the geometry required to draw the display tree and groups all the data in a single buffer and the content is drawn in one draw call, as if you were drawing a simple texture. Of course this power comes with a limitation, once called, all changes done to the children will not be reflected until you call flatten again to reflect the changes.
- **unflatten**: Disables the flatten behavior.

- **isFlatenned**: Indicates if the sprite is currently flattened or not.

Let's try this out, in the code below, we add multiple images inside a `Sprite` and rotate the container on each frame:

```
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;
        }

    }
}
```

In this test, the animation is perfectly smooth and runs at 60 frames per second, but we can optimize this, to make sure we reduce the number of draw calls to the minimum. So let's call the `flatten` API:

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

Now, all the children are drawn through a single draw call. You may not see the performance difference on desktop in terms of framerate, but this will make a difference in terms of CPU usage. By doing a few tests, you could see up to 10x CPU usage drop or even more.

> 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.

This feature will also help a lot on mobile. Of course, the other value of this feature is that it is dynamic, you can rearrange things after calling `unflatten` and then `flatten` again, yes, you can compile textures from dynamic sprites in a way. At anytime, a child can be modified; the changes will be reflected on screen, when calling explicitly `flatten` again.

Remember, that the flatten behavior works only for static content (Sprite), Starling does not currently offer a similar optimization for MovieClips. Such a feature could be added in a future release.

MovieClip

We just saw how to use the `Sprite` API in Starling, but what about animations like with `MovieClip`? Every Flash developer is familiar with the concept of movieclip, for the past few years, most AS3 developers have recreated the `MovieClip` API based on `BitmapData`.

The image below illustrates a sprite atlas showing each frame of our `MovieClip`:

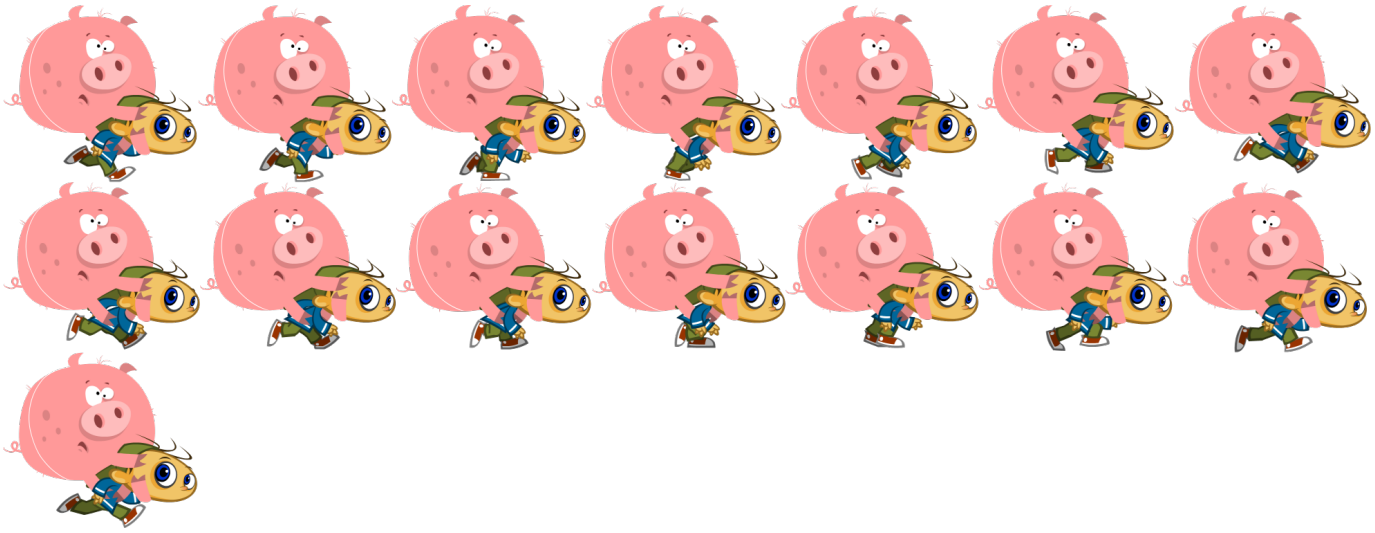

Figure 1.24

A sprite atlas for our running boy.

The idea is that each frame will be sampled by the GPU and displayed in our scene; by updating the texture on each frame we will reproduce the concept of movieclip. So how do we create this animation? Well, Flash Pro is your best friend for that, each frame of your animation is exported to a sequence of images. Then those images are loaded into a tool like TexturePacker (http://www.texturepacker.com), which exports them merged into a single texture used to feed our `MovieClip` object.

The image below illustrates the frames of a sprite atlas:

Figure 1.25

Frames in a sprite atlas.

If you are mipmapping your atlases, you need to make sure you that there is a 2pixels space between each frame so that when mipmapped, there is no bleeding. Meaning that when the GPU will sample a frame, it does not sample some pixels from other frames too.

Now, remember, there are some restrictions on the size of the textures, you need to remember that Stage3D (Molehill) has been designed with mobile in mind. As a result, Stage3D enforces OpenGL ES2 restrictions, like power of two textures. As a result, *TexturePacker* helps you to respect this and also integrates a nice feature called *AutoSize*, determining for you the best width and height for the texture, while respecting of course the maximum width and height. (Starling limits to 2048*2048):

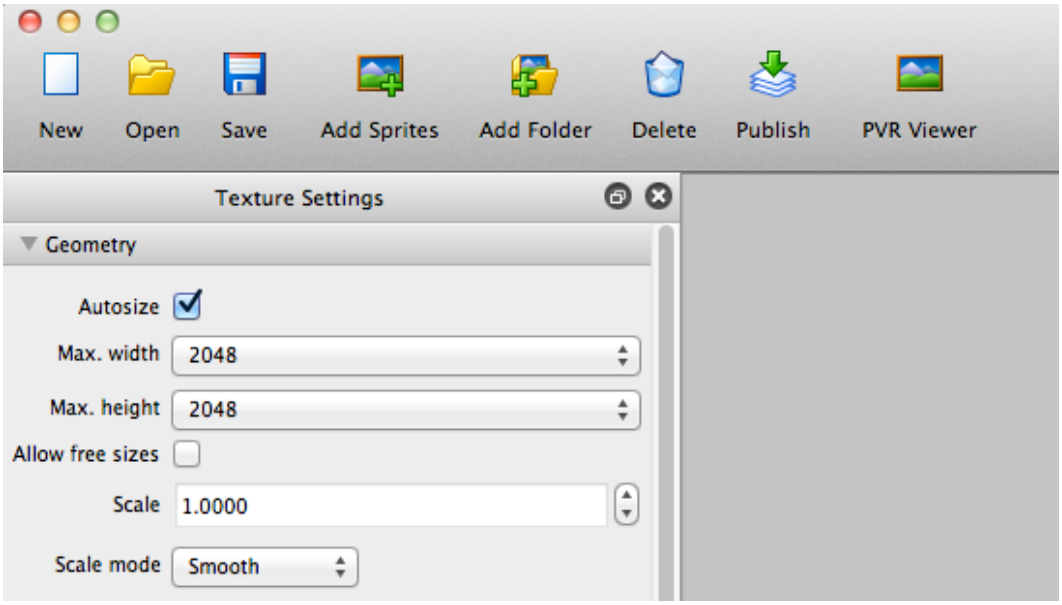

Figure 1.26

Autosize feature in TexturePacker.

As mentioned before, Starling will also automatically make sure you are using power of two sizes. If you do not, Starling will take care of it for you and automatically find the next power of two size available for your image and crop it.

To let Starling know at which position the frames are located, you need to provide an XML file to the `TextureAtlas` API, which will generate out of that our `Vector` of textures. Here again, TexturePacker generates this file for us automatically, when generating the spritesheet. You can choose which format to use: XML, JSON, etc.

Starling supports natively XML, here is below what this XML file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<TextureAtlas imagePath="running-sheet.png">
    <!-- Created with TexturePacker -->
    <!-- http://texturepacker.com -->
    <!-- $TexturePacker:SmartUpdate:2b3f5fa2588769393bcea9b632749826$ -->
    <SubTexture name="running0001" x="0" y="0" width="304" height="284"/>
    <SubTexture name="running0002" x="304" y="0" width="304" height="284"/>
    <SubTexture name="running0003" x="608" y="0" width="304" height="284"/>
    <SubTexture name="running0004" x="0" y="284" width="304" height="284"/>
    <SubTexture name="running0005" x="304" y="284" width="304" height="284"/>
    <SubTexture name="running0006" x="608" y="284" width="304" height="284"/>
    <SubTexture name="running0007" x="0" y="568" width="304" height="284"/>
    <SubTexture name="running0008" x="304" y="568" width="304" height="284"/>
    <SubTexture name="running0009" x="608" y="568" width="304" height="284"/>
    <SubTexture name="running0010" x="0" y="852" width="304" height="284"/>
    <SubTexture name="running0011" x="304" y="852" width="304" height="284"/>
```

```
    <SubTexture name="running0012" x="608" y="852" width="304" height="284"/>
    <SubTexture name="running0013" x="0" y="1136" width="304" height="284"/>
    <SubTexture name="running0014" x="304" y="1136" width="304" height="284"/>
    <SubTexture name="running0015" x="608" y="1136" width="304" height="284"/>
</TextureAtlas>
```

The beauty of this is that you get total control over the frames and therefore the framerate, allowing you to have multiple movieclips using independent framerate. Well, the good news is that Starling applies the same technique, but on the GPU, here is how the `MovieClip` constructor looks like:

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

In the code below, we create our texture containing our movieclip frames:

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

var bitmap:Bitmap = new SpriteSheet();

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

Then we retrieve our XML description, describing the position of each frame in the spritesheet:

```
[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);
```

Then we can retrieve the frames related to our running boy:

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

Note that we passed here the argument “running_” meaning that we only want the frames related to this from our spritesheet, we could have requested another sequence like “jump”, “fire” or any other, if they we defined.

Let’s have a look at our complete code:

```
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 );
         }
    }
}
```

When testing, we get the following result:

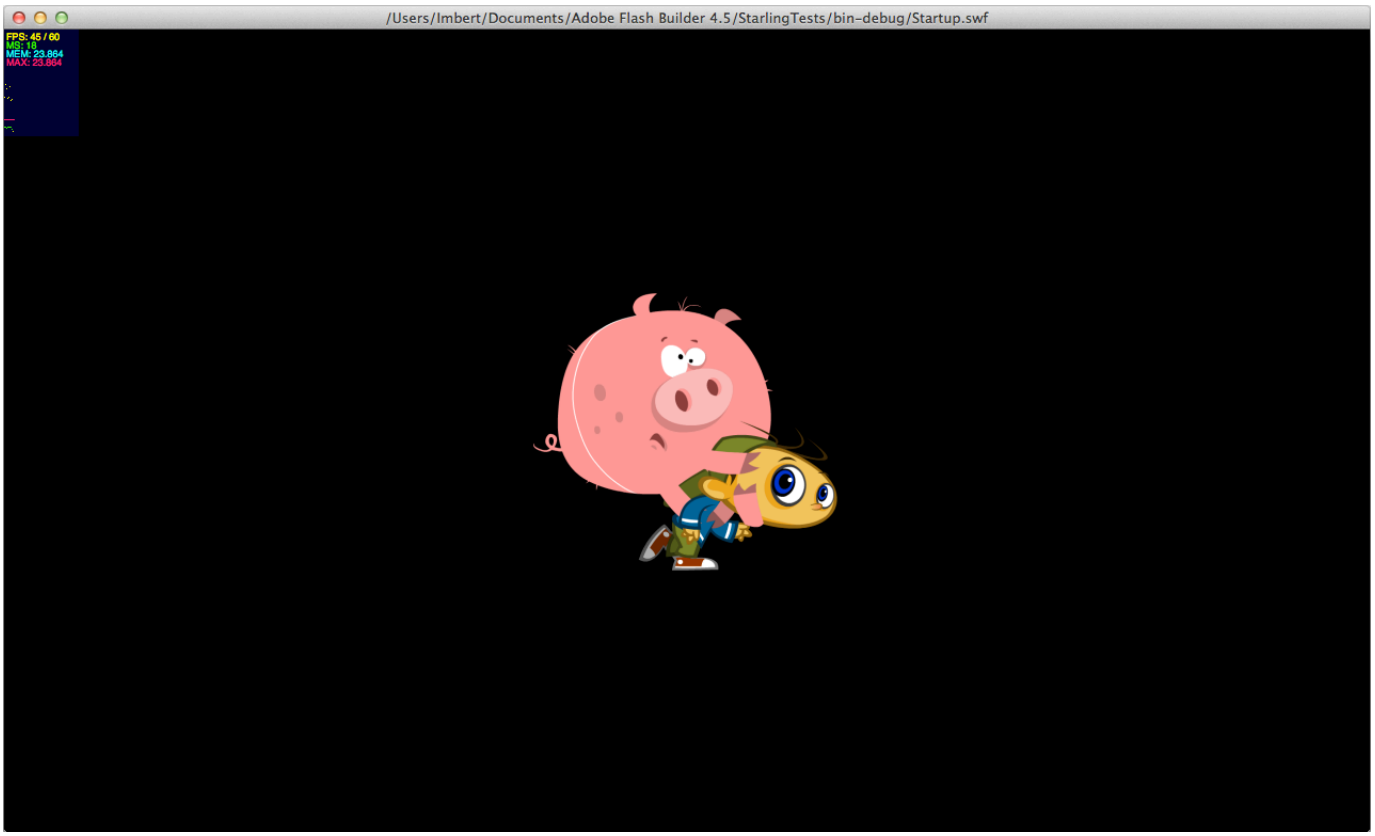

Figure 1.27

Our running boy rendered.

Later on we will cover a simple caching technique to reuse assets, so that you limit the number of objects to instanciate during your application's lifetime.

To flip our movieclip, as with the native Flash APIs, just use the `scaleX` property, and then add the width to reposition it at the exact same place:

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

mMovie.scaleX = -1;

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

Which gives us the following result:

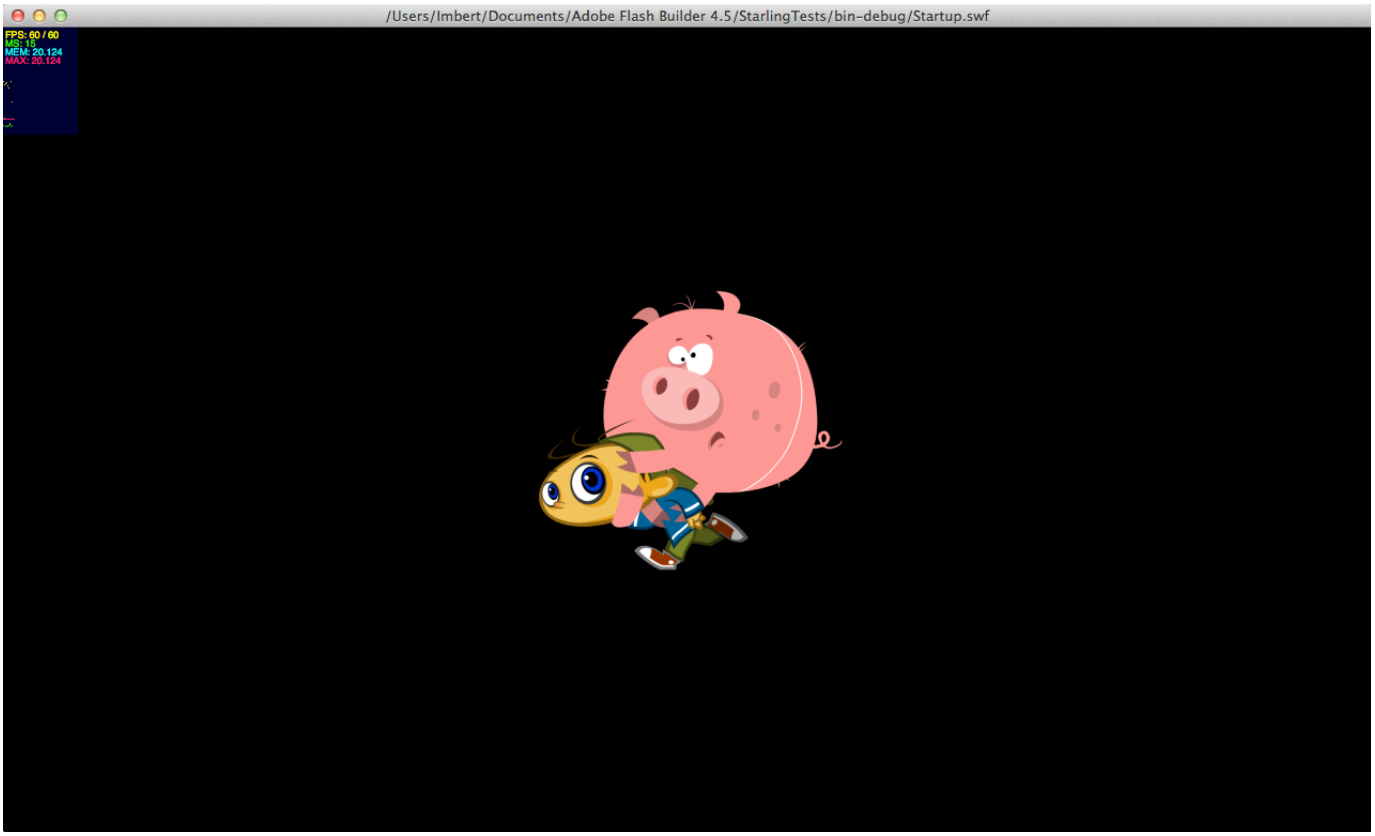

Figure 1.28

Our MovieClip flipped.

As stated earlier, it is very common and actually a good practice to put all of your assets on a single texture file. Why?

First, it is convenient, all the assets of your game are stored in one texture file. Using a single texture minimizes the number of upload you will ask to the GPU. Remember, uploading is costly on the GPU especially on mobile. So the less you upload, the better it is. Finally, switching from one texture to another is also costly, so the better it is if you have a single texture to sample your assets from rather than switching constantly, from one texture to another.

Texture Atlas

We just discovered the concept of sprite atlas, now, I would like to introduce you to the concept of texture atlases. In a texture atlas, all our assets are contained in one single texture.

In the following figure, we add another sequence of frames in the same bitmap:

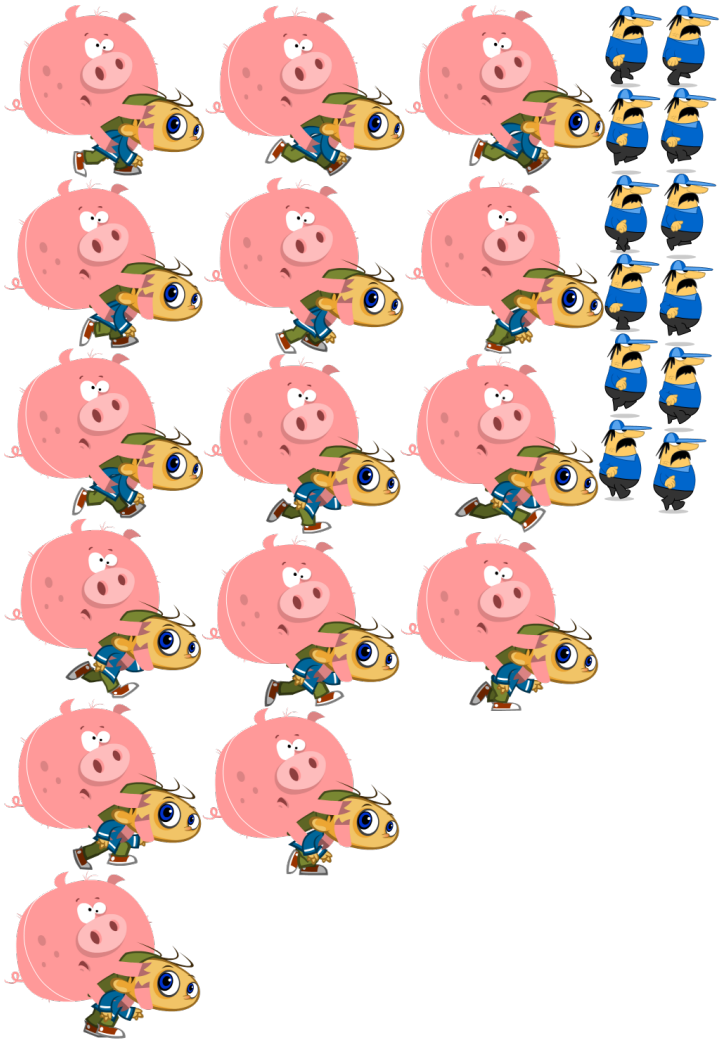

Figure 1.29

Texture Atlas containing all our assets.

Our XML descriptor file now contains also now the butcher textures:

```
<?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"/>
…
```

We now can reference those frames by using the `getTextures` API on the `TextureAtlas`:

```
// 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 );
```

We then end with our butcher next to our little boy:

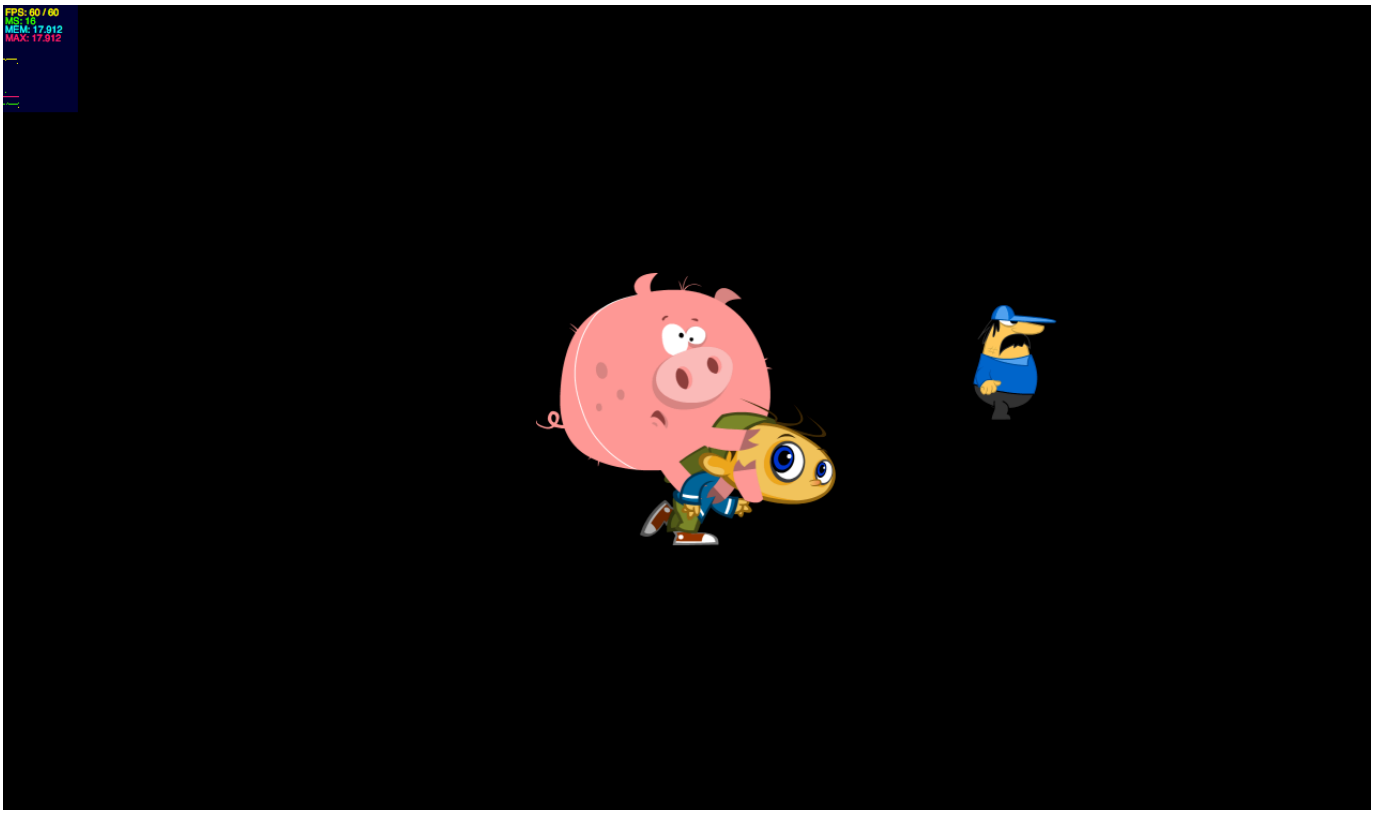

Figure 1.30

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

Note that at this state, we have our movieclips created and on the scene, but they do not play. To play them, we need to use a `Juggler` object.

A default juggler is available as a `juggler` property of the `Starling` object, handling your content. To animate the boy and the butcher, we would add the following lines:

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

Once added, our movieclips are now animated! Of course, anytime we want, we can pause or stop the playback:

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

Note that the difference between the two is subtle. The `pause` API will pause the playback leaving the current frame at its position, whereas the `stop` API, will reposition the playhead at the first frame.

Let's have a look at the entire set of APIs available on the `MovieClip` object, some of them may look familiar to you as a Flash developer, some others will be very useful like the ability to set a specific framerate, replace or add frames at runtime and many others:

- **currentFrame** : The current frame.
- **fps** : The default frames per second. Used when you add a frame without specifying a duration.
- **isPlaying** : Indicates if the movie is currently playing.
- **loop** : Indicates if the movie is looping.
- **numFrames** : The number of frames of the clip.
- **totalTime** : The accumulated duration of all frames.
- **addFrame** : Adds a frame with a specified duration.
- **addFrameAt** : Inserts a frame at the index specified.
- **addFrame** : Adds a frame with the default duration.
- **getFrameDuration** : Returns the duration (in seconds) of a frame at a certain index.
- **getFrameSound** : Returns the sound of a frame at a certain index.
- **getFrameTexture** : Returns the texture of a frame at a certain index.
- **pause** : Pause playback.
- **play** : Start playback. Make sure that the clip has been added to a Juggler too.
- **removeFrameAt** : Removes the frame at the index specified.
- **setFrameDuration** : Sets the duration of a certain frame in seconds.
- **setFrameSound** : Sets the sound that will be played back when a certain frame is active.
- **setFrameTexture** : Sets the texture of a certain frame.

We will not cover through examples all of them, but we can see some pretty useful ones, like for instance `addFrameAt` and `removeFrameAt`, allowing you at runtime to add or remove frames at runtime. Or even set a different duration for a specific frame, and of course helper APIs like `isPlaying` or `loop`.

In the code below, we want the frame 5, to have a 2 seconds duration:

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

We could also add dynamically a sound to a specific frame:

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

Thanks to those APIs, it is possible to entirely assemble movieclips at runtime, from dynamically loaded or embedded assets, which can be extremely powerful.

A common scenario requiring the use of such APIs like `addFrameAt`, `removeFrameAt` etc, is when you want to have multiple states of an animation grouped inside one `MovieClip`. Using the native `MovieClip` API, you would have MovieClips on each frames of the parent MovieClip and have the states played when switching to one frame from another. Starling MovieClips are not containers; as a result we will need to dynamically change the frames to play the state we want.

To finish, let's plug this little running boy with the keyboard like we would do in a game to control it:

```
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;
        }
    }
}
```

If you need to listen to the end of the animation, you can listen to the `Event.COMPLETE` event:

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

We have been using the `Juggler` object in this example, let's see how this object works internally and what you can do with it.

Juggler

The Juggler API allows you to animate any objects implementing the `IAnimatable` interface. `MovieClip` objects implement the latter, but you can also define you own type of animated object for Starling, all you need to do is implement the `IAnimatable` interface and override the `advanceTime` method. This is how the particles extension works, we will come back to this at the end of this tutorial.

Below we can see how animation is done on a `MovieClip`, the main logic is located here. On each frame, the texture is swapped. With the native APIs, a similar mechanism would be changing the `bitmapData` used by a `Bitmap` object on each frame:

```
// 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);
}
```

Here is a list of APIs available on the `Juggler` API:

- **add** : Adds an object to the juggler.
- **advanceTime** : API intended to be called if needed to manually handle the Juggler main loop.
- **delayCall** : Delays the execution of a certain method. Returns a proxy object on which to call the method instead. Execution will be delayed until time has passed.
- **elapsedTime** : The total life time of the juggler.

- **isComplete** : The status of the Juggler.
- **purge** : Removes all objects at once.
- **remove** : Removes an object from the juggler.
- **removeTweens** : Removes all objects of type Tween that have a certain target.

Another interesting feature of the juggler is the ability to delay calls. In the following code, we use the juggler to delay the moment when the child is going to be removed from the parent:

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

In some scenarios, you may require to create another `Juggler` to handle animation while the main content of your game for instance is paused. To animate elements like menus overlaying your paused game, you may want to use a Juggler, all you need to do is create an instance and call its `advanceTime` API.

For this, you would need to architect your game by using a `Juggler` for each main blocks used for your game. When the user will press pause, you do not want to pause the entire content, calling the `stop` API from the Starling object would actually cause this, by stopping all draw calls and all frame events:

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

Instead, each main block of your game (menus, background, playfield) will be handled by separated jugglers. When the game needs to be paused, only specific jugglers will be paused, allowing you to control which parts of the game needs to be paused or resumed.

In the code below, we define a custom class `BattleScene` containing our battle scene:

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

From the outside, we would call the `advanceTime` API by using the `EnterFrameEvent.EVENT` event and passing the time passing:

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

When the game is paused, we stop calling the `advanceTime` API, hence stopping the battle scene content. Of course, we would need our menus to be overlayed and animated, to handle this, we would just add:

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

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

Just switching the `paused` Boolean would do the magic.

Let's have a look now at another important part of Starling for interactions, the `Button` API.

Button

Starling natively supports the concepts of buttons. Here is the signature of the `Button` constructor:

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

By default, the `Button` class creates an internal `TextField` to support labels and the text is centered inside the button. In the code below we create a simple button out of an embedded bitmap that we use as a skin:

```
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);
        }

    }
}
```

Note that we use here the `fromBitmap` API, to create the texture that we need to feed the `Button` object for its skin:

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

Let's create a simple menu out of some data contained in a `Vector`, a simple loop and we are done:

```
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);
            }
        }
    }
```

By testing the code above, we get the following result:

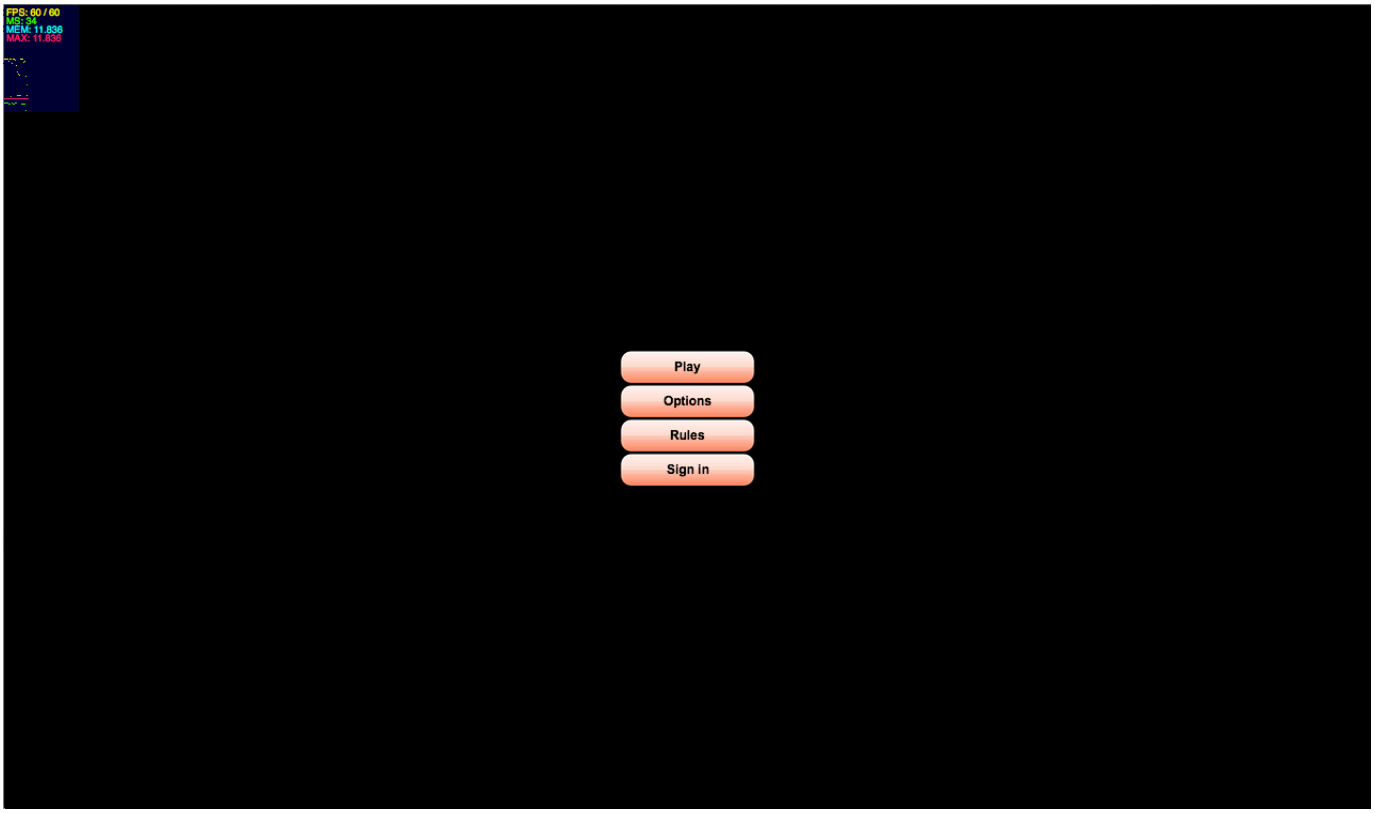

Figure 1.31

A simple menu, made out of buttons.

But wait, we did not use here a sprite sheet with all the button skins. We just embedded one skin and we had to upload it to the GPU through the `fromBitmap` API of the `Texture` object. That is fine if you intend to use a single skin for all your buttons. A best practice would be to define all our skins into a single texture atlas, just like what we did for the two movieclips previously (boy and butcher).

Now let's have a look at the list of properties available on the `Button` class:

- **alphaWhenDisabled** : The alpha value used when button is disabled.
- **downState** : The texture used when the button is in its downstate (clicked).
- **enabled** : Determines if the button can be triggered.
- **fontBold** : Determines if the label's font is using bold styling or not.
- **fontColor** : The color of the font.
- **fontName** : The font used for the button's label. Can be a system's font or a registered bitmap font.
- **fontSize** : Size of the font used for the button 's label.
- **scaleWhenDown** : The scale factor when the button is touched. When a down state texure is used, the button will not scale.
- **text** : The text to use for the button's label.
- **textBounds** : The position of the button's label.
- **upState** : The texture used when the button is not being touched.

In contrary of the native Flash APIs, the `Button` object is a subclass of `DisplayObjectContainer`, meaning that you are not restricted to the state properties to skin your button. You can decorate it the way you want, just like any other container.

Note that the `Button` object also dispatches a specific `Event.TRIGGERED` event to handle click state:

```
// listen to the Event.TRIGGERED event
myButton.addEventListener(Event.TRIGGERED, onTriggered);

private function onTriggered(e:Event):void
{
    trace ("I got clicked!");
}
```

The `Event.TRIGGERED` event does bubble; if you want to leverage event propagation you can rely on it and catch the event at the container's level:

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

Let's add a background to our interface, like a simple scrolled texture:

```
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;
        }
    }
}
```

We now have our background scrolling, with our menu on top, the figure below illustrates the result:

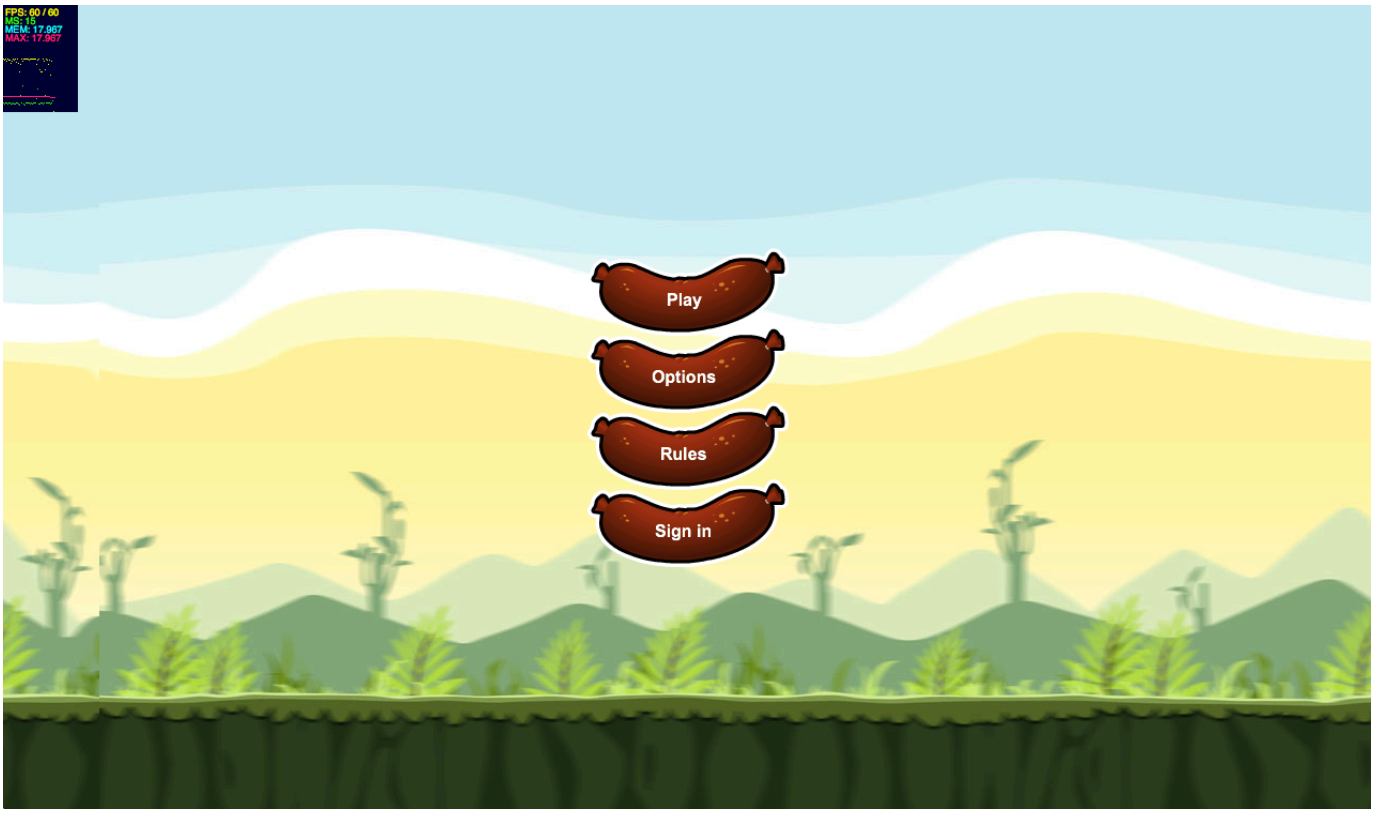

Figure 1.32

Our menu and a scrolled background.

Note that we applied a little motion blur to the image in Photoshop, to emphasize the feeling of motion when the image is being scrolled.

TextField

We used the `starling.text.TextField` API briefly earlier when playing with quads. Let's spend some little time on how text works with Starling. You may wonder how a GPU renders a font, but there is a trick here. Behind the scenes Starling creates a native `TextField` object on the CPU, and uses it as a offscreen buffer to render the font. Once rasterized, the texture is uploaded to the GPU and you have your text on screen.

> Note that the TextField API does not create a native flash.text.TextField instance for each starling.text.TextField you use. One instance is cached and reused to render all the text.

In the following code, we create a `TextField` object and display some text using the *Verdana* system font:

```
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);
            }
        }
}
```

The following figure illustrates the result:

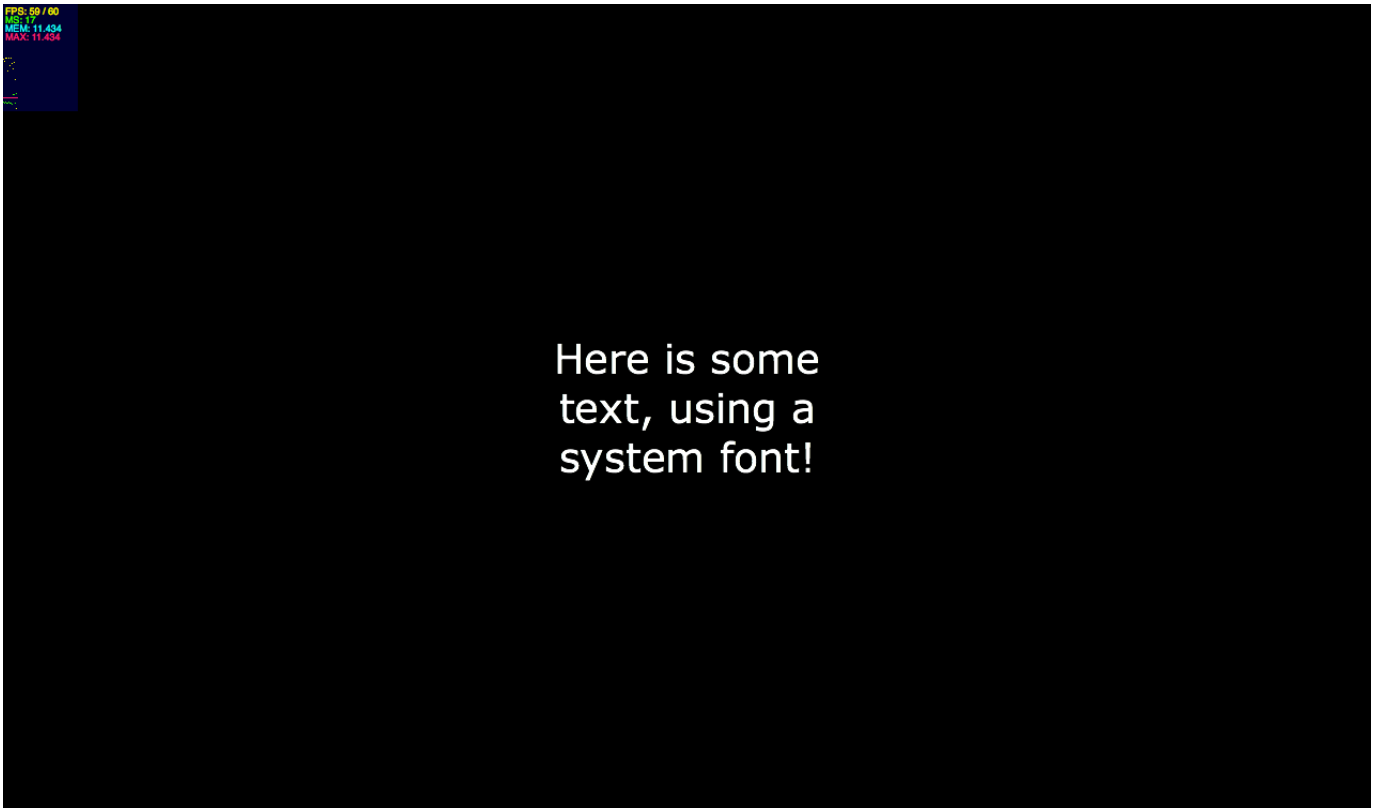

Figure 1.33

Some simple text.

Again, remember that what you see here on screen is not a real text field, but a snapshot of the text field that got drawn to a bitmap texture and uploaded to the GPU.

Let's have a closer look at the set of properties exposed by the `TextField` API:

- **alpha** : Alpha of the text.
- **autoScale** : Scales the text automatically to fit the dimensions of the text block.
- **bold** : Determines if the label's font is using bold styling or not.
- **border** : Allows displaying a border around the edges of the text field. Useful for visual debugging.
- **bounds** : The bounds of the actual characters inside the text field.
- **color** : The color of the text.
- **fontName** : The name of the font.
- **fontSize** : The size of the font.
- **hAlign** : The horizontal alignment of the text.
- **italic** : Defines if the text is italic or not.
- **kerning** : Allows using kerning information with a bitmap font (where available). Default is YES.
- **text** : The displayed text.
- **textBounds** : The bounds of the actual characters inside the text field.
- **underline** : Defines if the text is underlined or not.
- **vAlign** : The vertical alignment of the text.

One of the coolest features of `TextField` is the `autoScale` property that we will cover very soon. But first, let's play with a few of those properties, in the following code, we add a border around the text and make it bold and then change its color:

```
// 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;
```

The figure below illustrates the result:

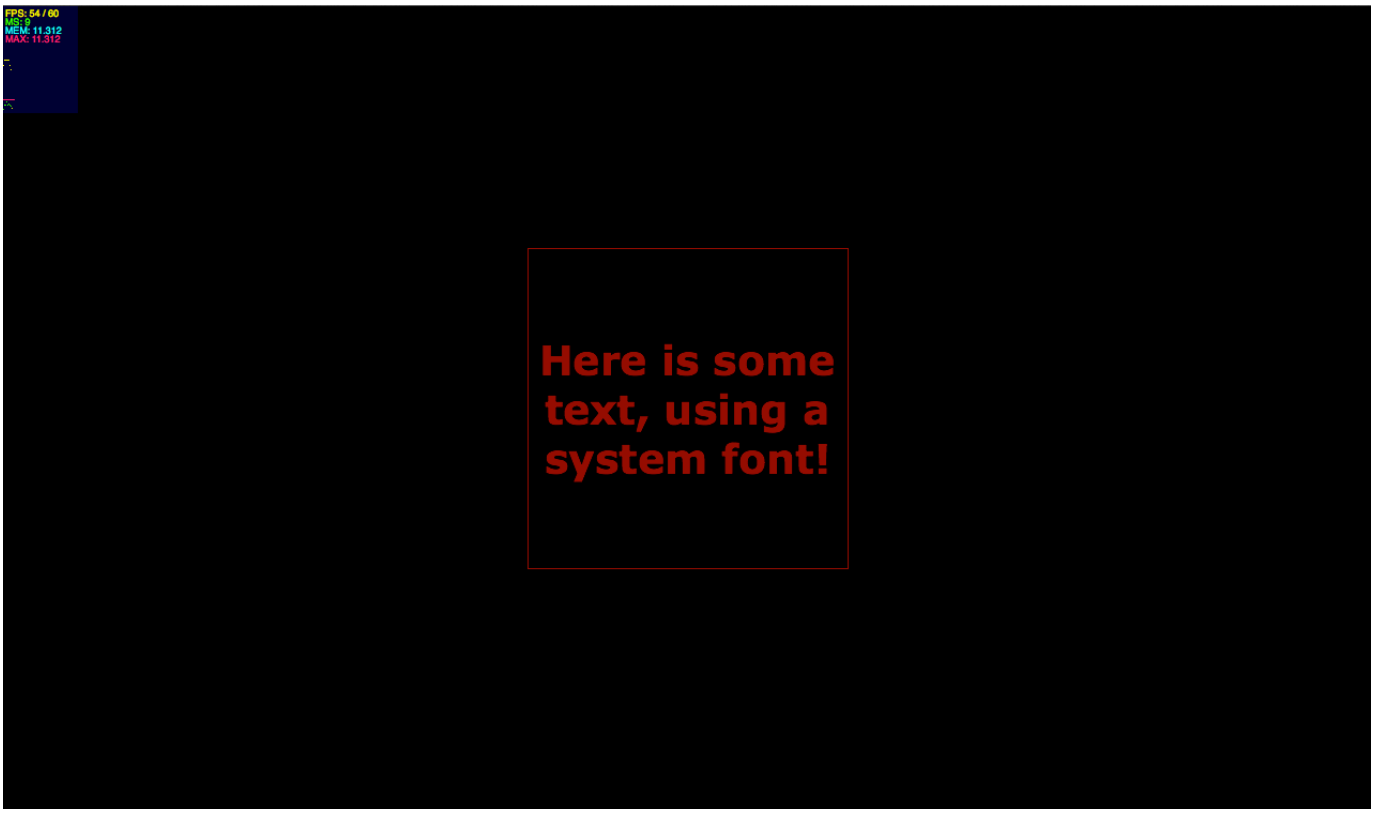

Figure 1.34

Some simple colored text.

Note that the `TextField` object does not handle HTML text natively. However, Starling may introduce such a feature in a future version. Do not hesitate to raise your voice on the Starling forums, if you would like to see such a feature added. Remember Starling is open source, you can also implement such a feature by yourself and provide it as an extension to the community.

We just used in the previous example a very common system font, which are pretty useful but you will need to use embedded fonts in most projects today. In the context of games for instance, you want to be able to deliver a strongly branded experience, so you will need to use embedded fonts.

Embedded fonts

Starling does not expose an `embedFonts` property just like the native `TextField` API does. But don't worry, it is actually very easy to use embedded fonts with Starling. First, as expected, you need to do is embed or load at runtime your font, and then just use it as the font name passed to the `TextField` constructor.

In the code below, we embed a font, instanciate it, and use it by passing the `fontName` property to the `TextField` constructor:

```
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);
        }
    }
}
```

Automatically, the `TextField` object finds the embedded font by its name and uses it. The figure below shows our text using the embedded *Abduction* font:

Figure 1.35

Some simple text using an embedded font.

Pretty simple right? In your content, you may want to offer a way to input text, to enter let's say, a user name, an email and other informations. As you can imagine, text editing, it is a tricky thing to do on the GPU. Most platforms, even on mobile use the same trick as the one we will discuss. The idea is simple, for this scenario, we will just use the classic native display list we have always used.

As the display list sits above the Stage3D scene, we will be able to place our input text field above the GPU content. To do this, we will rely on a very convenient feature from Starling, native overlay.

Earlier we saw how Starling is able to display an error message when the wmode value used is incorrect. Starling relies internally on the native overlay feature. As you know, Starling works on top of Stage3D, the native overlay feature allows you to get access to the display list from the Starling object and add objects to the native display list that you've always used in Flash and overlay native elements like video, or text input on top of the Stage3D content.

When Starling detects that the wrong wmode value us being used, the internal `showFatalError` function is called to display the warning message on top of Stage3D. Obviously, in this scenario, as there is no GPU surface to render into, Starling relies here on the display list on top of Stage3D:

```
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);
}
```

In the code below, we create an input text field and add it on top of our Starling content, on the display list:

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

This can be extremely useful when text input is required, to enter various informations when starting a game for instance. Note that you can also access the native stage (`flash.display.Stage`) at anytime by using the `nativeStage` property on the `Starling` object:

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

Let's have a look at bitmap fonts now, to get the best performance out of fonts with Starling!

Bitmap fonts

Because of the texture creation done behind the scene, for the best performance and lowest cost in terms of resources and GC load, you can use the `TextField` API with bitmap fonts. The idea is the same, all the glyphs from a font are exported to a spritesheet, this texture is then sampled to render each glyph required.

Here is below, the GlyphDesigner tool (from 71 squared - commercial) preparing glyphs spritesheet for the *Britannic Bold* font:

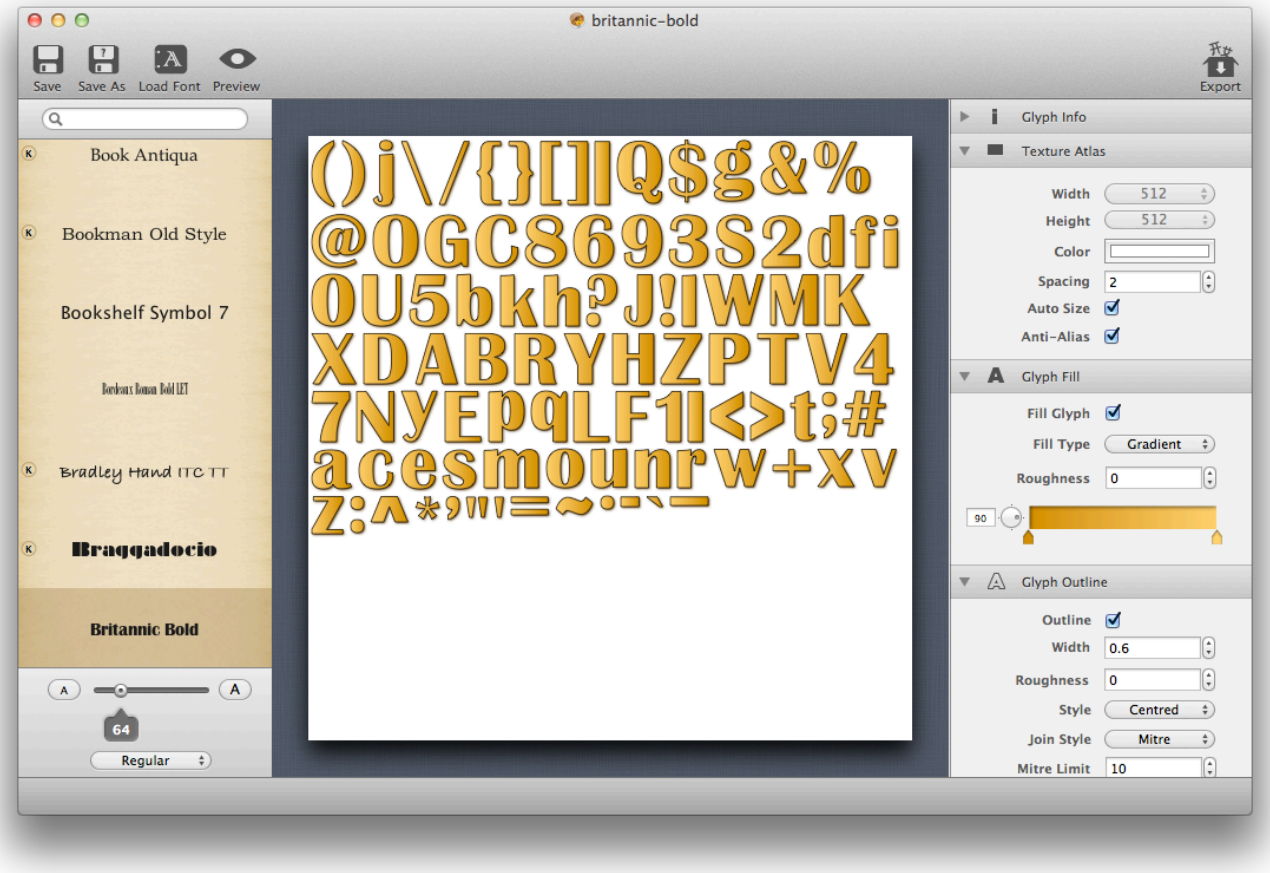

Figure 1.36
GlyphDesigner on MacOS.

On Windows a similar tool called *Bitmap Font Generator* (from Angel Code - free) can be used:

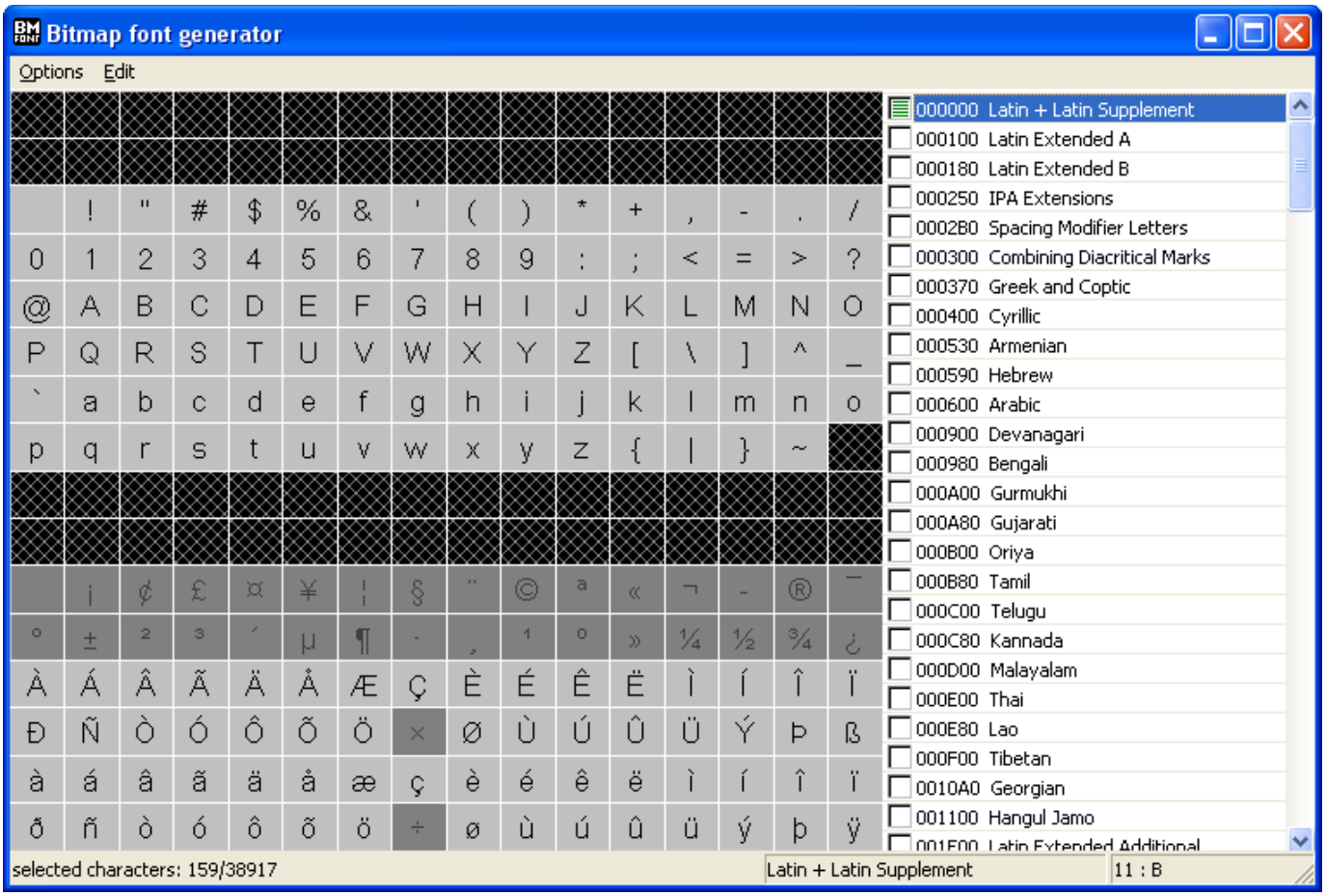

Figure 1.37

Bitmap Font Generator on Windows.

Yes, trust me, you want to use the MacOS tool ☺

Of course, nothing prevents you from generating this glyphs spritesheet at runtime, using a `TextField` containing each glyph, laying out the glypes as on the spritesheet and just use the `BitmapData` API and draw it. Depending on your platform (desktop or mobile) your choices may vary. Doing it at runtime would be valuable for size reasons, whereas embedding would give you faster startup time for your content. You will decide!

Once exported, the texture will be saved as an image and the description of the image, giving details about the position of each glyph on the image as a .fnt file (XML or text), which looks like the following:

```
<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>
```

Once our bitmap texture and .fnt description file exported, we can embed them as usual:

```
[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;
```

To use those, we will be using the following static `TextField` APIs:

- **registerBitmapFont**: Registers a bitmap font.
- **unregisterBitmapFont**: Unregisters a bitmap font.

We will then pass our font texture and its file decription to the `BitmapFont` object and register it through the `registerBitmapFont` API:

```
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);
        }
    }
}
```

Once registered, we can pass the font name when creating the `TextField` object.

The figure below illustrates the result:

Figure 1.38

Some text rendered through a bitmap font.

Now, let's change the string to make it little longer, and see what happens:

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

As expected, the bounding box of our text field is too small to display our entire string, as a result we end up with our text being cut and not properly displayed:

Figure 1.39

Some large bitmp text being cut off.

Fortunately, Starling exposes an `autoScale` property on `TextField`:

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

This property is extremely useful in games when you need to localize your strings and you need to make sure that text fits into a specific box. Most of the time, for design reasons, you want the text to be scaled a little bit so that your layout stays exactly the same and aligned whatever the length of the string.

Below is a figure showing the `autoScale` property enabled, with our string slightly changed:

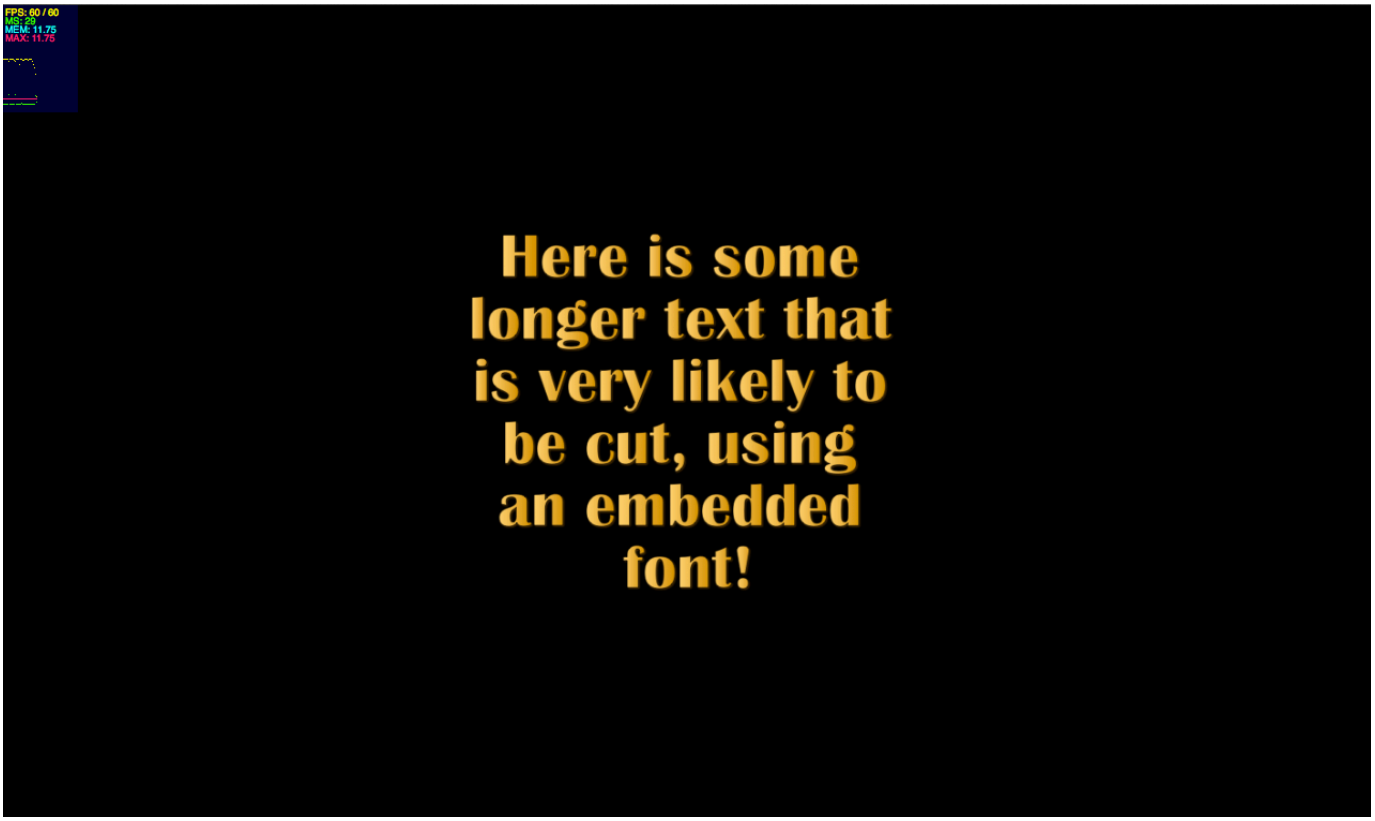

Figure 1.40

Large text being scaled down to fit the bounding box.

We can enhance our previous example with bitmap fonts for our menu and scrolled background:

```
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;
        }
    }
}
```

The figure below illustrates the result:

Figure 1.41
Custom font used in our menu.

Let's discover now another neat Starling feature, called rendered textures.

RenderTexture

The `starling.textures.RenderTexture` API, allows developers to create non-destructive drawing. As a Flash developer, think about the `BitmapData` API. This feature can be really useful when creating applications like drawing tools, where you need to draw continuously inside a texture and preserve previous drawings.

In the code below, we replicate the `BitmapData.draw` feature, on the GPU through Starling:

```
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);
```

```
                }
            }
        }
    }
```

When holding the mouse down, and moving the mouse around, we get the following result:

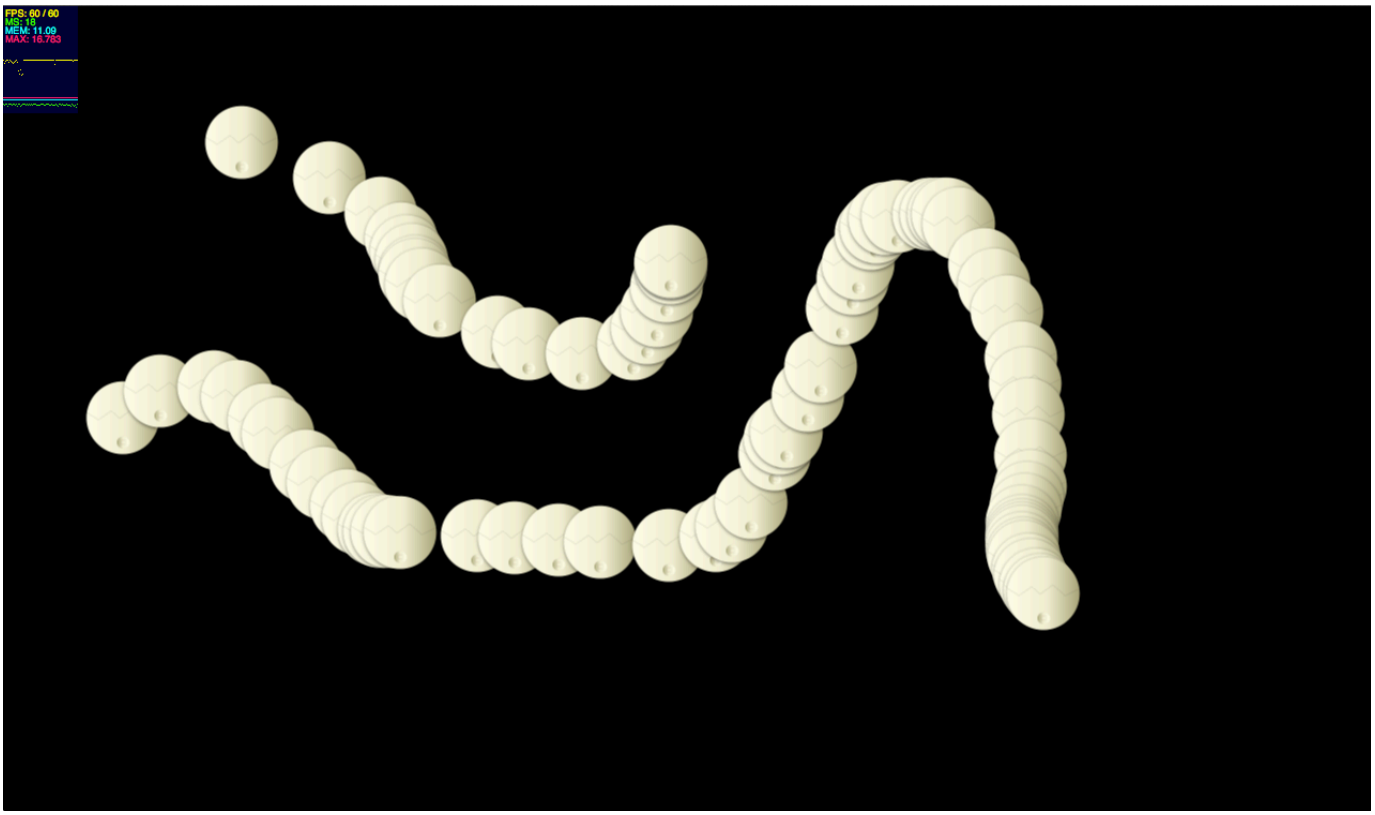

Figure 1.42

Non destructive drawing.

We are going to talk now about a very well known topic to Flash developers, tweening!

Tweens

Starling also supports tweening and comes with its own tweening engine, supporting most of the most common equations, illustrated below:

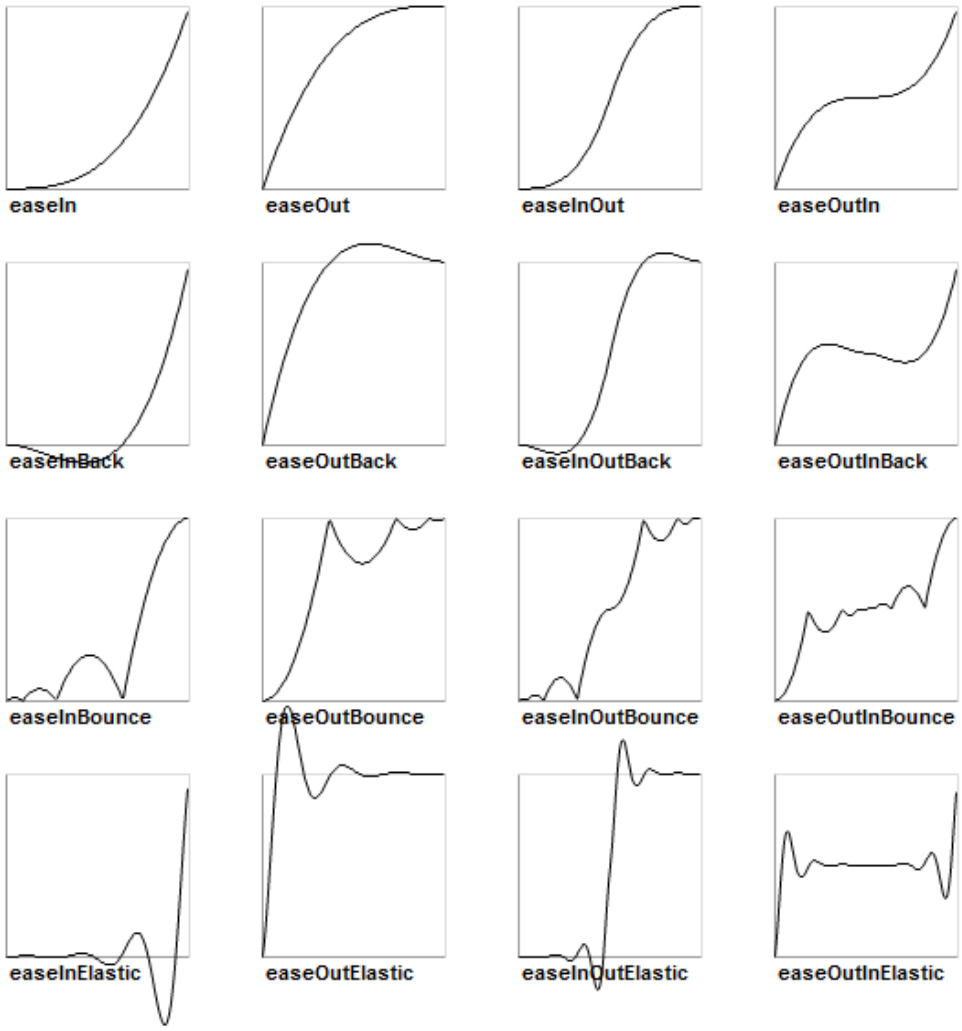

Figure 1.43

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

In the code below, we apply a tween to the `x` and `y` properties of our text field with a bounce effect:

```
// 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);
```

Here is the list of APIs available on a `Tween` object:

- **animate** : Animates the property of an object to a target value. You can call this method multiple times on one tween.

- **complete** : End the tween when called.
- **currentTime** : The current timing for the tween.
- **delay** : The delay before the tween is started.
- **fadeTo** : Animates the alpha property of an object to a target value. You can call this method multiple times on one tween.
- **isComplete** : Informs if the tweening is complete or not.
- **moveTo** : Animates the x and y properties of an object simultaneously.
- **onComplete** : Reference to the callback called when tween is complete.
- **onCompleteArgs** : Parameters to be passed to the the callback when tween is complete.
- **onStart** : Reference to the callback called when tween starts.
- **onStartArgs** : Parameters to be passed to the the callback when tween starts.
- **onUpdate** : Reference to the callback called when tween is in progress.
- **onUpdateArgs** : Parameters to be passed to the the callback when tweenis in progress.
- **roundToInt** :
- **scaleTo** : Animates the scaleX and scaleY properties of an object simultaneously.
- **target** : The target object that is animated.
- **totalTime** : The total time the tween will take (in seconds).
- **transition** : The transition method used for the animation.

As an example, in the code below, we listen to the end of the tween, then we dispose the object being animated by passing the dispose argument through the `onCompleteArgs` API:

```
// 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];
```

In the code below we listen to the progress of the tween by using three callbacks assigned to the `onStart`, `onUpdate` and `onComplete` events:

```
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")
        }
    }
}
```

Let's see now how we can handle our assets in a more organized way. So far, we have been using simple embed tags in our code to embed what is needed. In a large project, you want to be able to have all your assets in a central location. In the following section, we will see a few simple techniques to group your assets and reuse them.

Asset management

So far we have been using assets in a very simple way. In order to optimize the way you work with assets, it is highly recommend using an `Assets` object working as a central location to get resources from. This `Assets` object would be responsible for pooling the assets used and make sure they are reused instead of being thrown away and instanciated again which could pressure the GC.

In the code below, we degine a `getTexture` API on the `Assets` object, allowing us to retrieve an embedded texture

```
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.");
}
```

Note here, the use of a simple `Dictionary`, to store the asset, so that the next time we need, we just grab it from the pool, instead of recretating it.

As usual, the textures are embedded through the `Embed` tag:

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

Note that Starling does not force you to use embedded textures; you can also load a texture dynamically by using a `Loader` object for instance:

```
// 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);
}
```

A dynamic texture can also be generated out of a `BitmapData` object, so far we have been using pre-athored bitmaps with Starling, but what if we wanted to create a texture dynamically and use it as a texture?

For this, we would be using the static `fromBitmapData` API on the `Texture` class:

```
// 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);
```

Your Starling applications are likely to run on mobile devices, desktop browsers and as a result, on a large set of different screen sizes. Starling allows you to easily handle screen resizes, let's see how this works in the following section.

Handling screen resizes

As a Flash developer, you generally rely on a very simple event to handle screen resizes, such event is called `Event.RESIZE` and allows you to be notified everytime the page dimensions are being changed. By using the `stage.stageWidth` and `stage.stageHeight` properties you can then layout your content appropriately, no matter what screen size you are targeting.

To handle this, the `starling.display.Stage` used by Starling also dispatches a similar event, called `ResizeEvent.RESIZE`, allowing you to handle dynamic resize elegantly.

In the code below, we update our first example with our quad centered in the scene:

```
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;
        }
    }
}
```

Everytime our SWF is resized, the `ResizeEvent.RESIZE` is dispatched; note that the new dimensions are sent through the `ResizeEvent` object, we manually write them to the `stageWidth` and `stageHeight` properties. We then reposition our content, any other content in our application based on the stage dimensions would then correctly layout.

Plugging Starling with Box2D

The fantastic thing about Starling is its display list API, which makes it very easy to plug with existing frameworks. For instance, let's say we would like to use the Box2D framework for adding some physics to our game?

You can learn more about Box2D here and download the library that we will use in the following example: http://box2dflash.sourceforge.net/

In the code below, we make boxes fall on the ground with some simple gravity. Of course, everything is rendered through the GPU:

```
package
{
    import Box2D.Collision.Shapes.b2CircleShape;
    import Box2D.Collision.Shapes.b2PolygonShape;
    import Box2D.Common.Math.b2Vec2;
    import Box2D.Dynamics.b2Body;
    import Box2D.Dynamics.b2BodyDef;
```

```
import Box2D.Dynamics.b2FixtureDef;
import Box2D.Dynamics.b2World;

import starling.display.DisplayObject;
import starling.display.Quad;
import starling.display.Sprite;
import starling.events.Event;

public class PhysicsTest extends Sprite
{
     private var mMainMenu:Sprite;
     private var bodyDef:b2BodyDef;
     private var inc:int;

     public var m_world:b2World;
     public var m_velocityIterations:int = 10;
     public var m_positionIterations:int = 10;
     public var m_timeStep:Number = 1.0/30.0;

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

     private function onAdded (e:Event):void
     {
          // Define the gravity vector
          var gravity:b2Vec2 = new b2Vec2(0.0, 10.0);

          // Allow bodies to sleep
          var doSleep:Boolean = true;

          // Construct a world object
          m_world = new b2World( gravity, doSleep);

          // Vars used to create bodies
          var body:b2Body;
          var boxShape:b2PolygonShape;
          var circleShape:b2CircleShape;

          // Add ground body
          bodyDef = new b2BodyDef();
          //bodyDef.position.Set(15, 19);
          bodyDef.position.Set(10, 28);
          //bodyDef.angle = 0.1;
          boxShape = new b2PolygonShape();
          boxShape.SetAsBox(30, 3);
          var fixtureDef:b2FixtureDef = new b2FixtureDef();
          fixtureDef.shape = boxShape;
          fixtureDef.friction = 0.3;
          // static bodies require zero density
          fixtureDef.density = 0;

          // Add sprite to body userData
          var box:Quad = new Quad(2000, 200, 0xCCCCCC);
          box.pivotX = box.width / 2.0;
          box.pivotY = box.height / 2.0;

          bodyDef.userData = box;
          bodyDef.userData.width = 34 * 2 * 30;
          bodyDef.userData.height = 30 * 2 * 3;
          addChild(bodyDef.userData);
```

```
            body = m_world.CreateBody(bodyDef);
            body.CreateFixture(fixtureDef);

            var quad:Quad;

            // Add some objects
            for (var i:int = 1; i < 100; i++)
            {
                bodyDef = new b2BodyDef();
                bodyDef.type = b2Body.b2_dynamicBody;
                bodyDef.position.x = Math.random() * 15 + 5;
                bodyDef.position.y = Math.random() * 10;
                var rX:Number = Math.random() + 0.5;
                var rY:Number = Math.random() + 0.5;

                // Box
                boxShape = new b2PolygonShape();
                boxShape.SetAsBox(rX, rY);
                fixtureDef.shape = boxShape;
                fixtureDef.density = 1.0;
                fixtureDef.friction = 0.5;
                fixtureDef.restitution = 0.2;

                // create the quads
                quad = new Quad(100, 100, Math.random()*0xFFFFFF);
                quad.pivotX = quad.width / 2.0;
                quad.pivotY = quad.height / 2.0;

                // this is the key line, we pass as a userData the starling.display.Quad
                bodyDef.userData = quad;
                bodyDef.userData.width = rX * 2 * 30;
                bodyDef.userData.height = rY * 2 * 30;
                body = m_world.CreateBody(bodyDef);
                body.CreateFixture(fixtureDef);

                // show each quad (acting as a skin of each body)
                addChild(bodyDef.userData);
            }

            // on each frame
            addEventListener(Event.ENTER_FRAME, Update);
        }

        public function Update(e:Event):void
        {
            // we make the world run
            m_world.Step(m_timeStep, m_velocityIterations, m_positionIterations);
            m_world.ClearForces() ;

            // Go through body list and update sprite positions/rotations
            for (var bb:b2Body = m_world.GetBodyList(); bb; bb = bb.GetNext()){

                // key line here, we test if we find any starling.display.DisplayObject objects and apply t
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);
            }
        }
    }
```

When testing the code above, we get the following result:

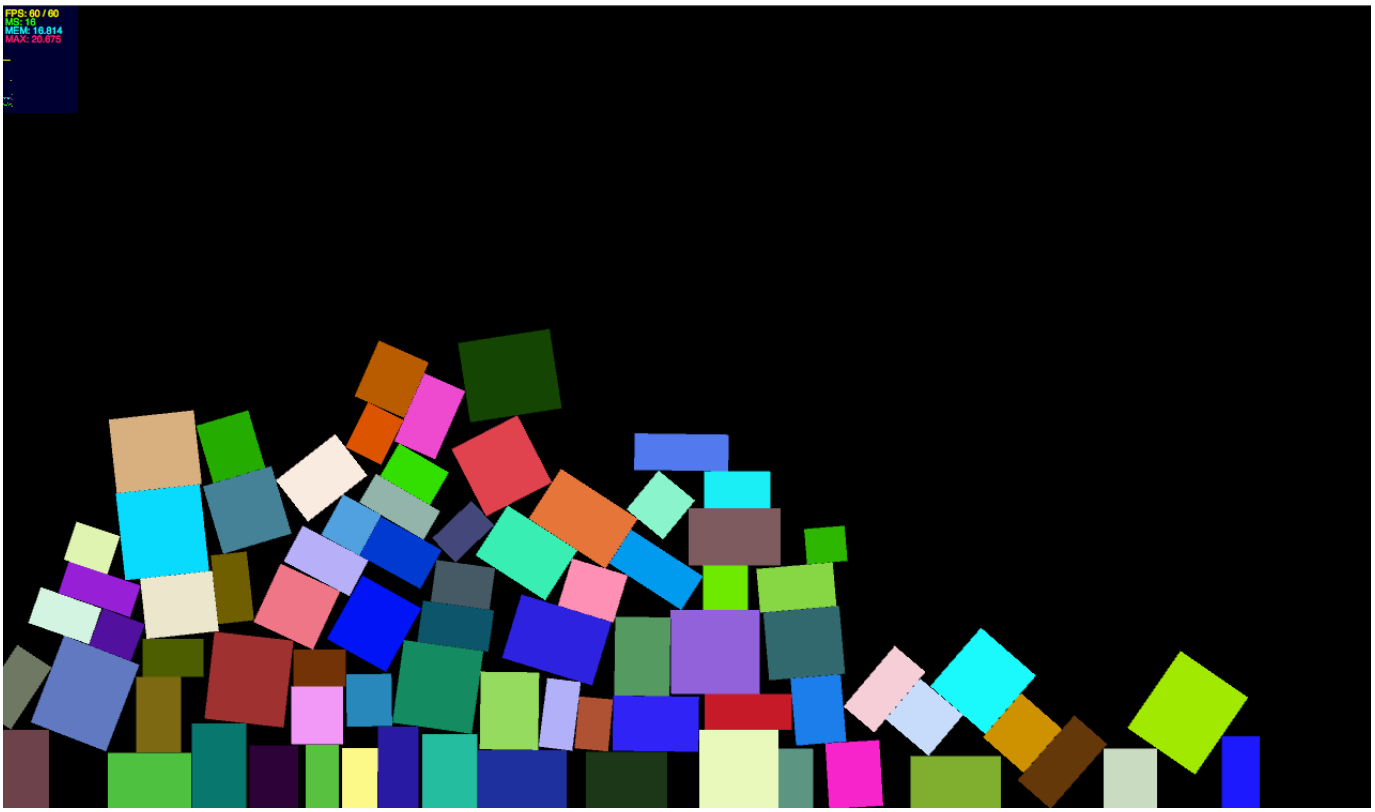

Figure 1.44
Box2D used with Starling.

And here is the same content running on a next version of AIR for mobile with Stage 3D support:

Figure 1.45

The same applicaton running on a tablet at 60fps.

Of course, we can easily replace those quads with textures, we would end up very quickly with something way richer graphically. Up to you now to build amazing 2D GPU content!

Profiling Starling

When it comes to profiling performance. The first and absolute must have in all content, is the frames per second benchmark. Generally, developers will keep this debug information visible at all times during debugging to monitor performance hickups, pain points and areas of improvements in content.

We have been using since the beginning of this tutorial the classic `Stats` class from mr.doob, available here: https://github.com/mrdoob/Hi-ReS-Stats

We could keep using the Stats class as we did so far. But of course there are always exceptions, especially on mobile. In a future version of AIR for mobile, Stage 3D will be available. When running on mobile devices, on platforms like Android, using the display list at the same time as the `Stage3D` surface, you risk a big performance hit. To give you an idea, on some GPU's, using the display list, will reduce your framerate by a factor of 2, so using a compiled framerate of 60 will end up resulting in 30fps for the display list and 30fps for the Stage3D content. As a result, even for single debugging purposes, you need to make sure that everything goes to Stage3D.

We just covered previously the concept of bitmap fonts, so we could totally develop a little FPS class displaying the framerate, which would be running on Stage3D?

The figure below illustrates the glyphs spritesheet we are exporting, note that we select only the glyphs useful for a FPS counter, to make the spritesheet as lightweight as possible:

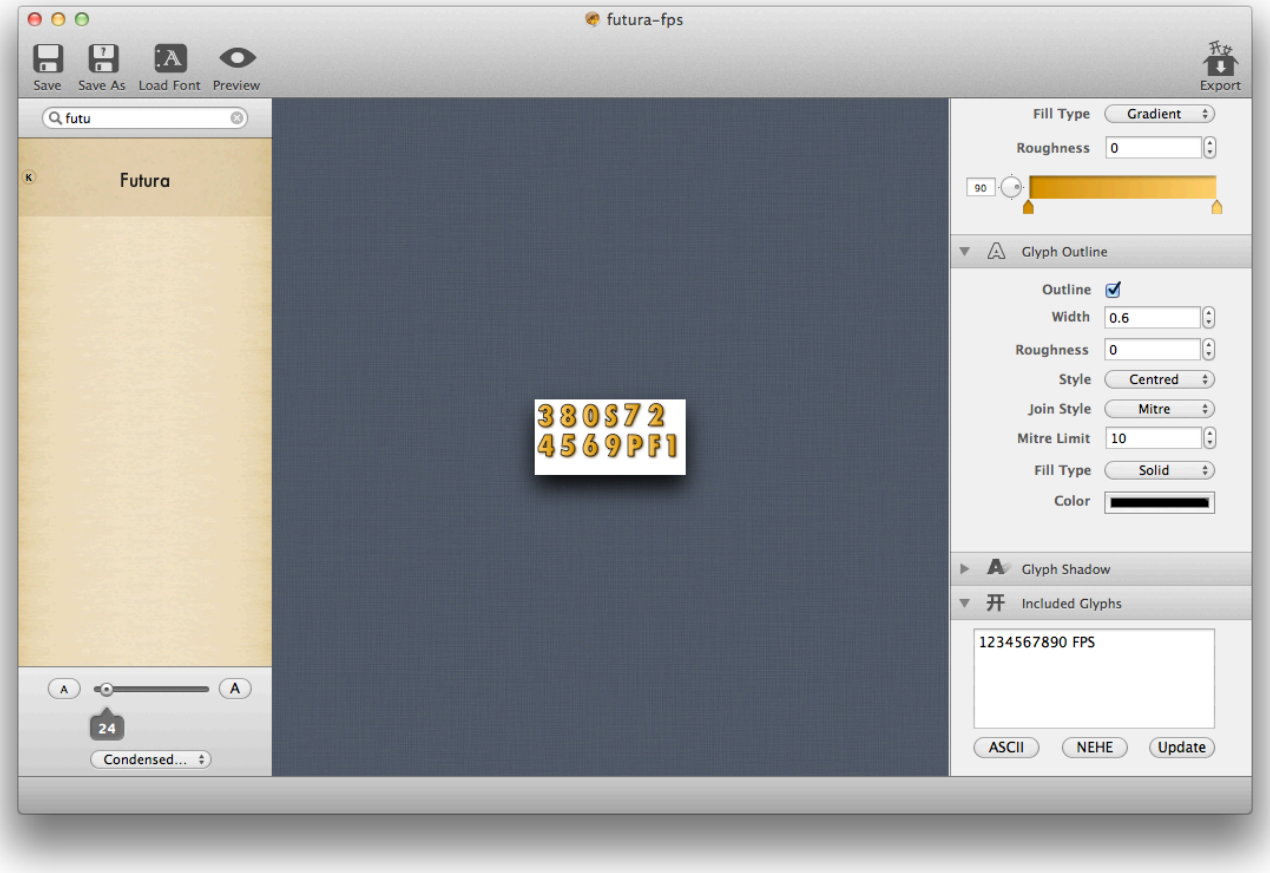

Figure 1.46

The text texture for our FPS counter.

Once the spritesheet and glyphs description files are exported, we use a `TextField` object with a `BitmapFont` to display the 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;
```

```
            }
        }
    }
```

To use it, very easy, add the following line anywhere inside your Starling world:

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

The figure below illustrates our FPS counter at work:

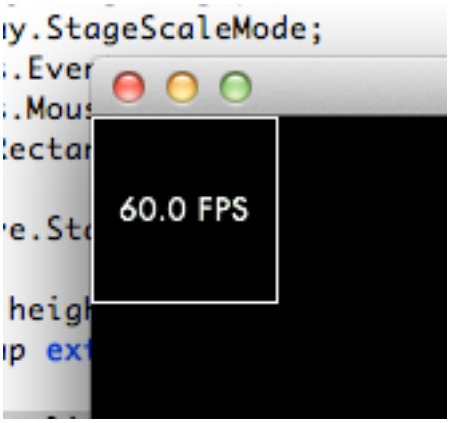

Figure 1.47

Our FPS counter rendered through the GPU.

We can now update our physics demo and integrate our FPS counter in it:

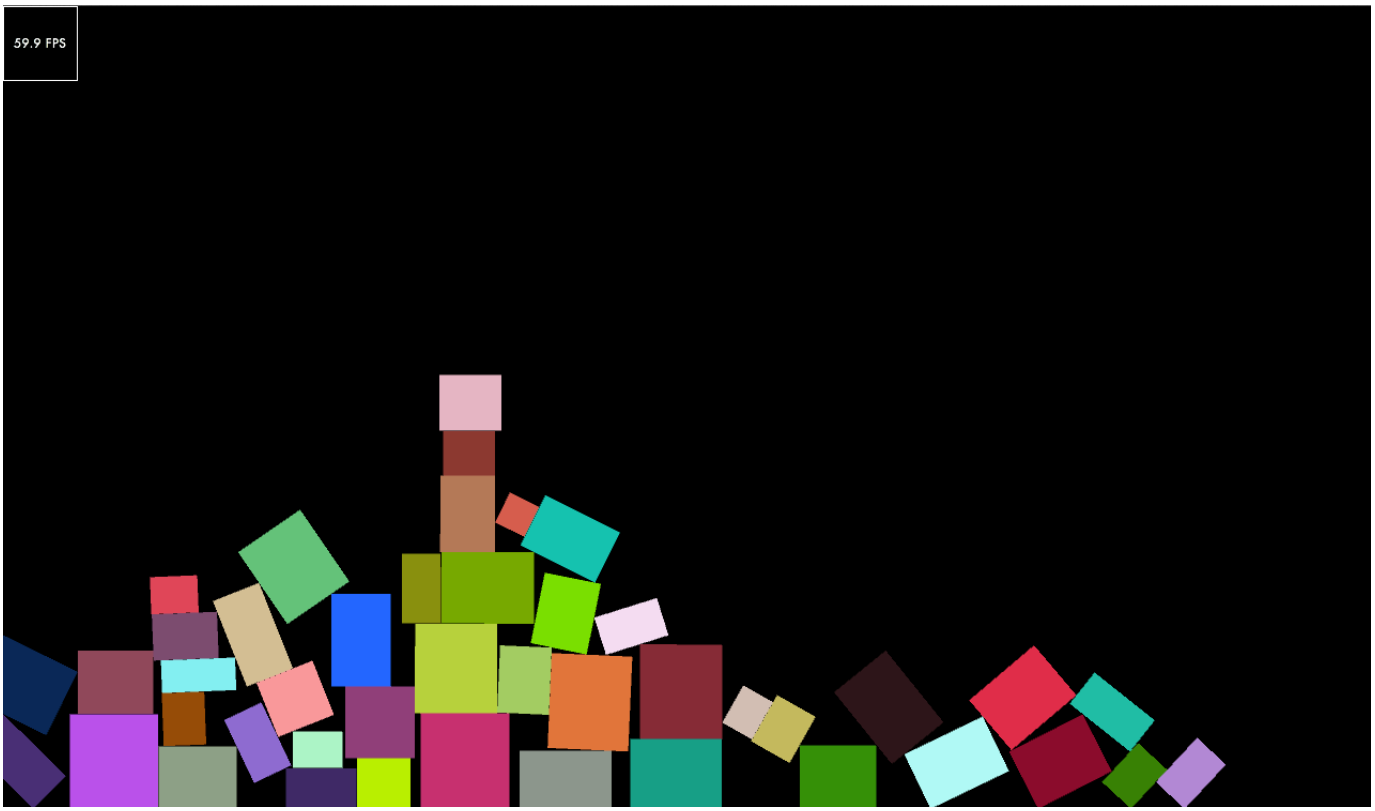

Figure 1.48

Our FPS counter integrated in our application.

We now have a way to profile the performance of our content also on mobile. As we mentioned previously, this solution allows us to completely get rid of the display list, not only for menus, UI animations, but also for displaying simple debugging informations like the framerate.

Note that the famous Stats class by mr.doob was recently ported to Starling, you can find it here amongst other contributions:

http://forum.starling-framework.org/topic/list-of-user-contributions#post-181

Particles

When it comes to beautiful effects, particles are definitely my favorites. And believe it or not, particles look amazing but are not really complex. Technically, particles are literally textured quads moved around using a specific blending mode producing beautiful combinations.

To design our particles for Starling, we will be using a very convenient tool named ParticleDesigner, developed by the same company (71 squared) behind GlyphDesigner, that we used for our bitmap fonts. Here is below a snapshot of ParticleDesigner, notice on the right, the emulation mode, where you can preview the particles you are designing:

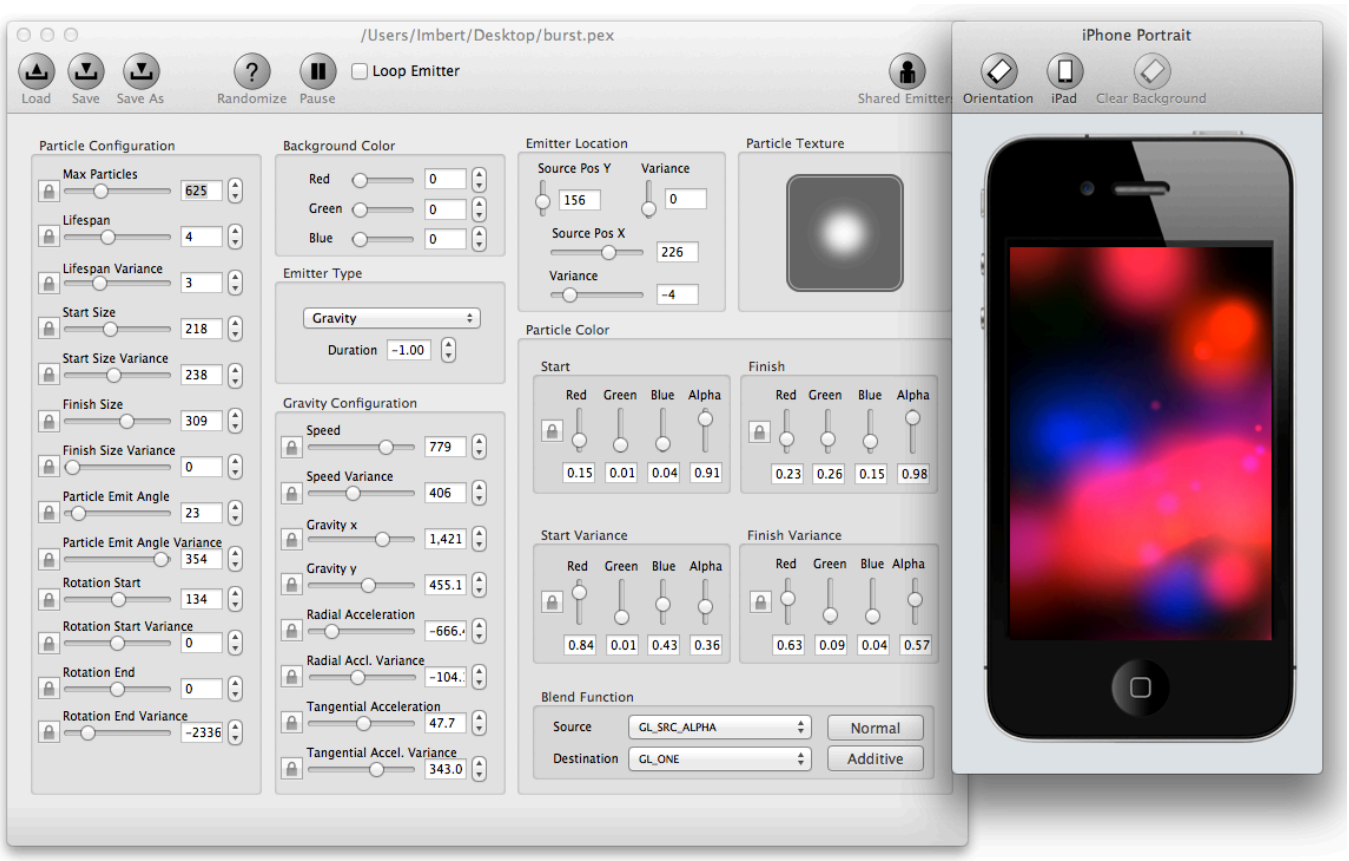

Figure 1.49

Particle Designer on MacOS.

The main window is just a bunch of parameters to modify. You can easily spend hours playing with all those options to create the particles you want. There is actually a randomize button to automatically generate parameters and as a result produce different particles effects.

The figure below illustrates, the save-as dialog box of Particle Designer, exporting the ParticleEmitter file (.pex) and the texture to render our particles. Those two files will be the files we will be using to feed our `ParticleDesignerPS` object.

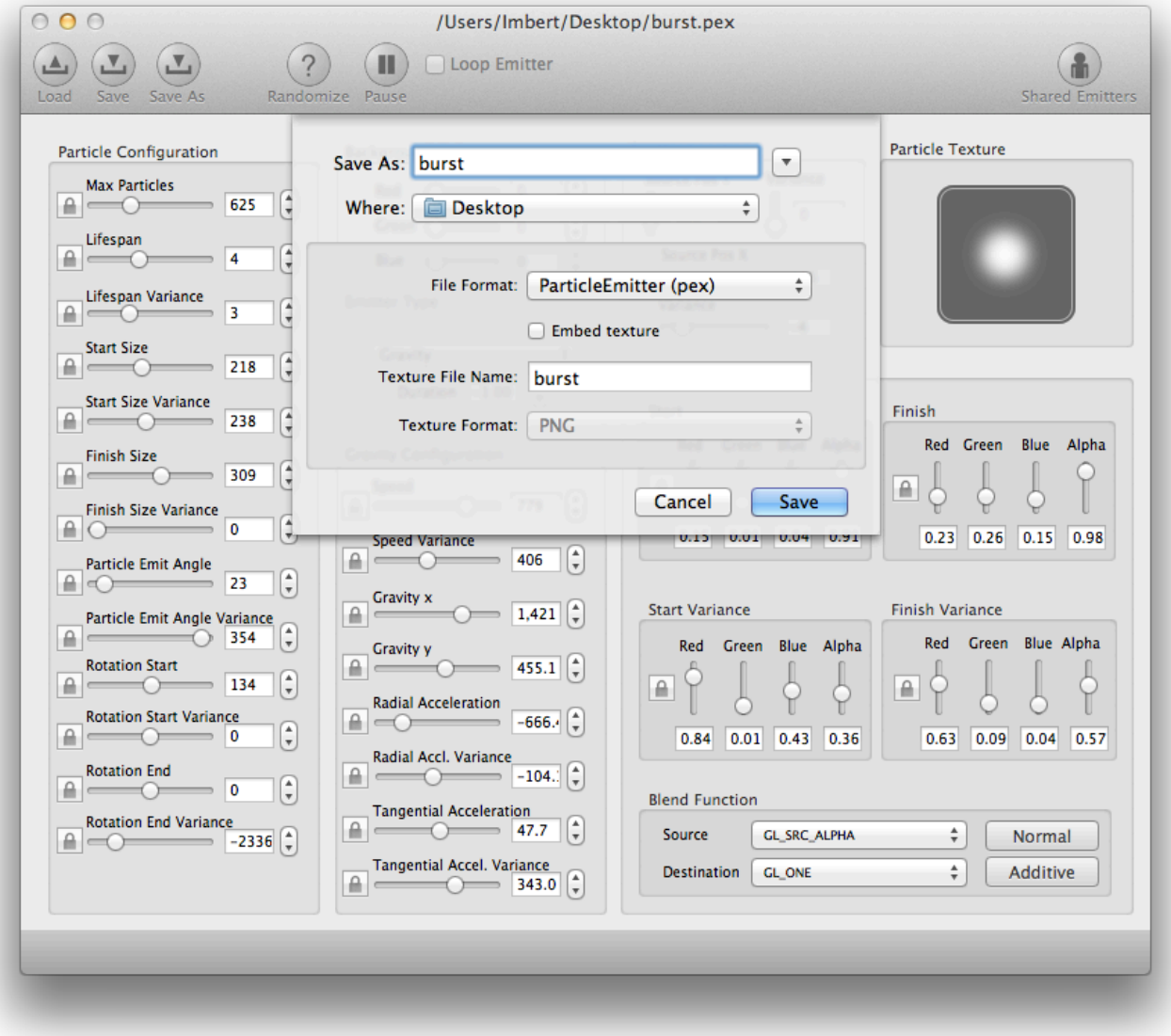

Figure 1.50
Particle Designer on MacOS.

Here is below, an example of particles created from ParticleDesigner and running through Starling:

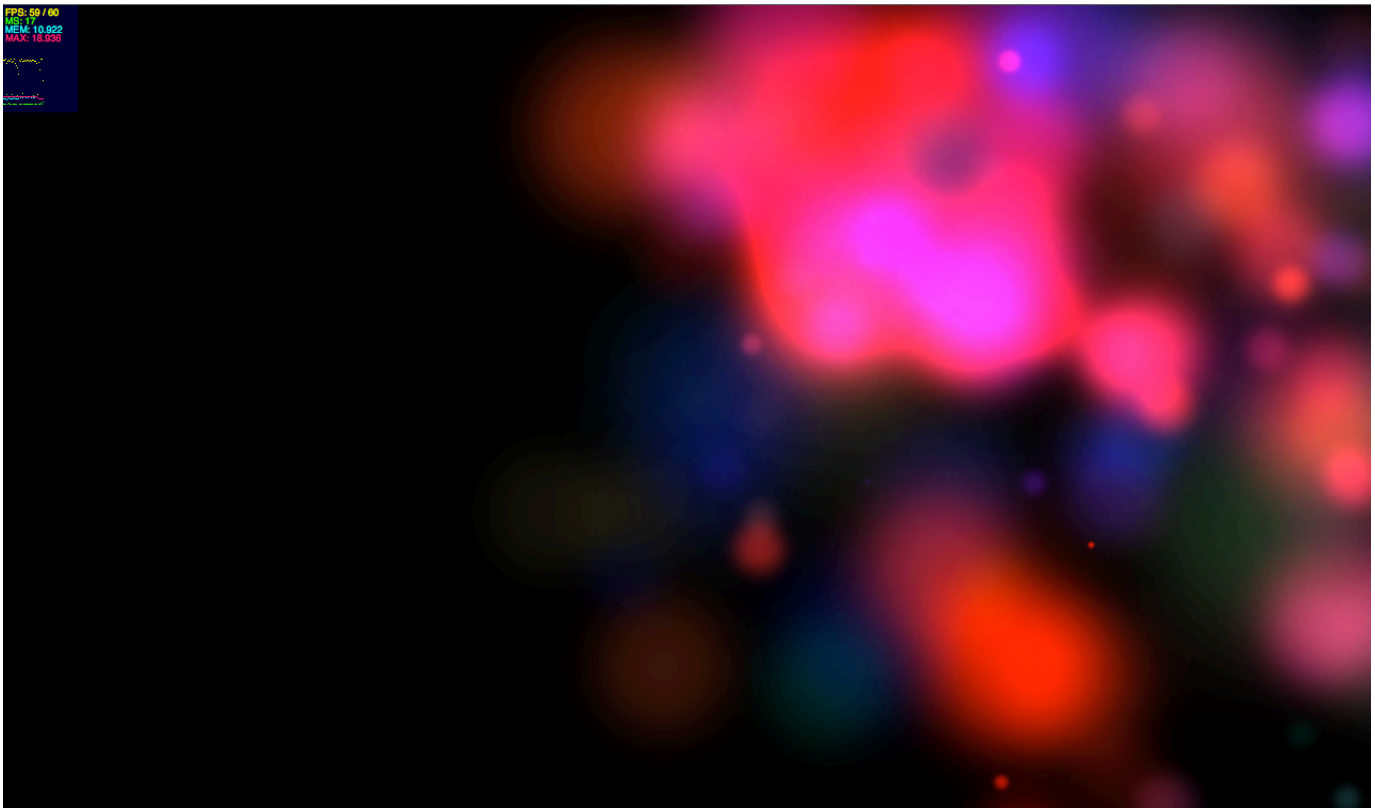

Figure 1.51

Custom particles running through Starling.

Beautiful no? If you look at it, Starling does not come out of the box with the particles feature. That's right, the particles engine is actually an extension of Starling and can be download at the following address: https://github.com/PrimaryFeather/Starling-Extension-Particle-System

Here is below another beautiful example of particles:

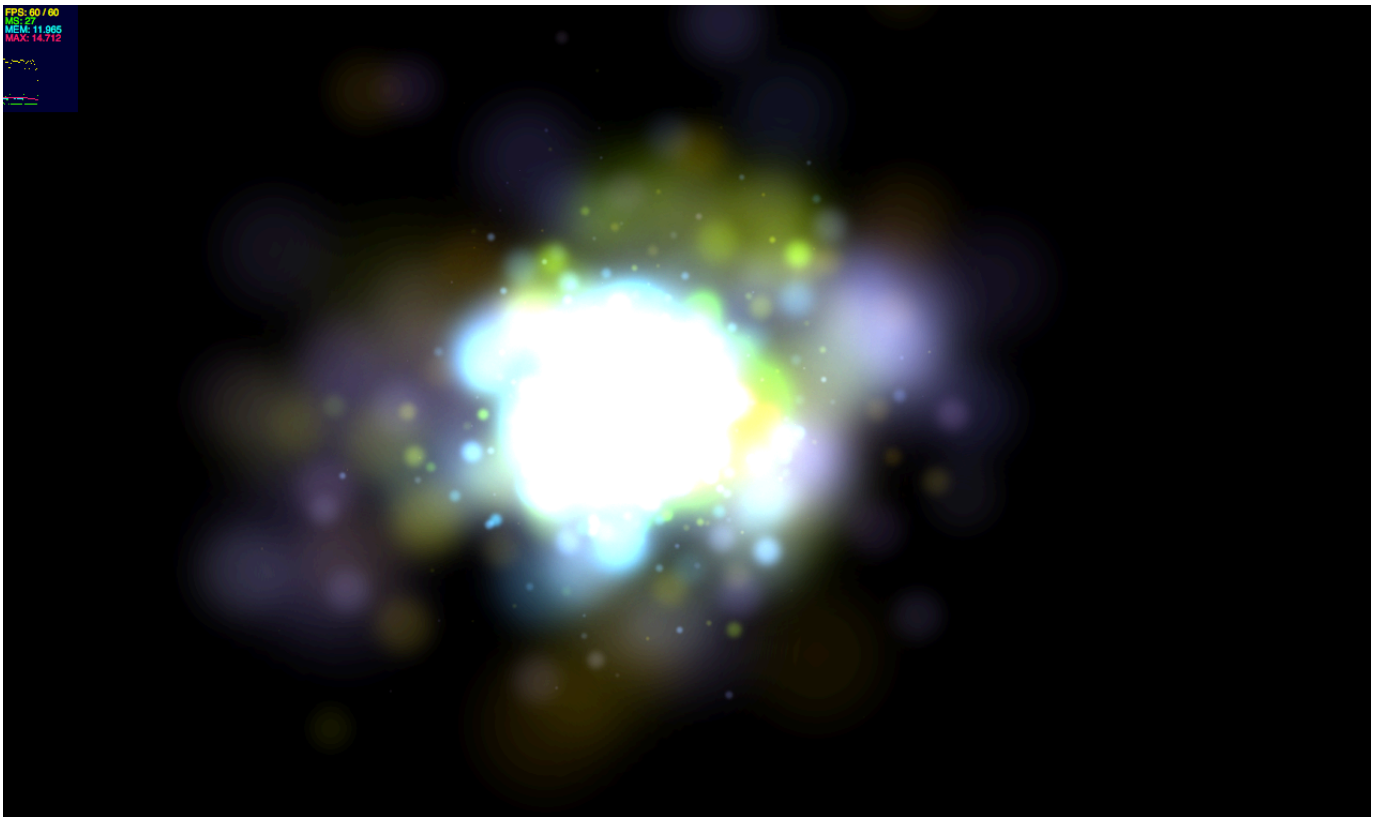

Figure 1.52

Custom particles running through Starling.

Keep in mind that the particles have to be considered as any other animated object in Starling, as a result, you need to add your particle object to a `Juggler` see them animated:

```
// 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);
```

Feel free to position the particles the way you want, the ParticleDesignerPS object is actually a `DisplayObject`, so you can use all of the expected features. Of course, as always, when you are done, you need to remove the particles from the juggler, and dispose the particles by calling the `dispose` API on the `ParticleDesignerPS` object.

Lee Brimelow (leebrimelow.com) created this example, where particles are used to simulate fire for a space ship, the figure below illustrates the idea:

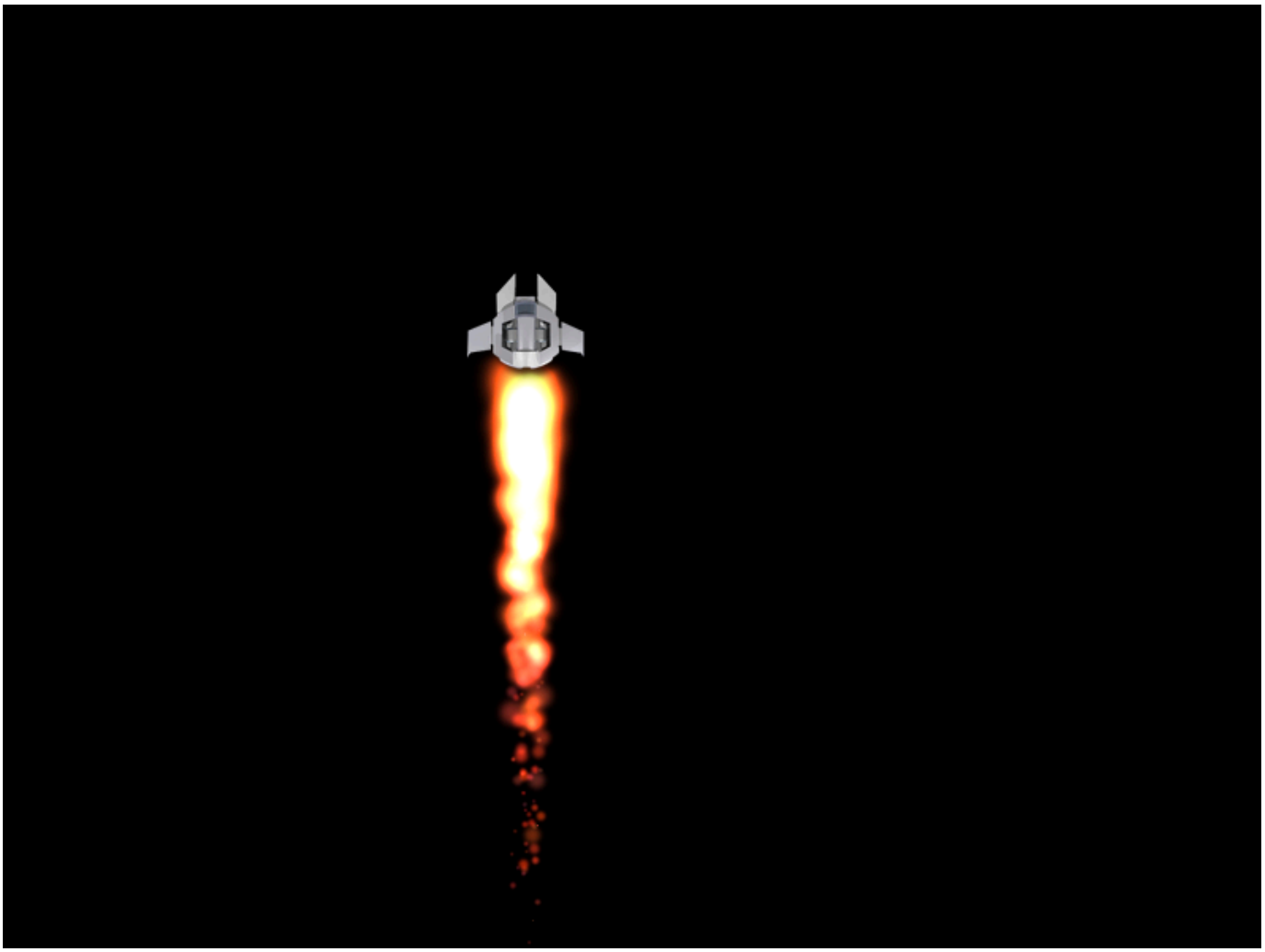

Figure 1.53

Particle effect integrated to our space ship.

We can now add some little particles on the rockets that our space ship fires:

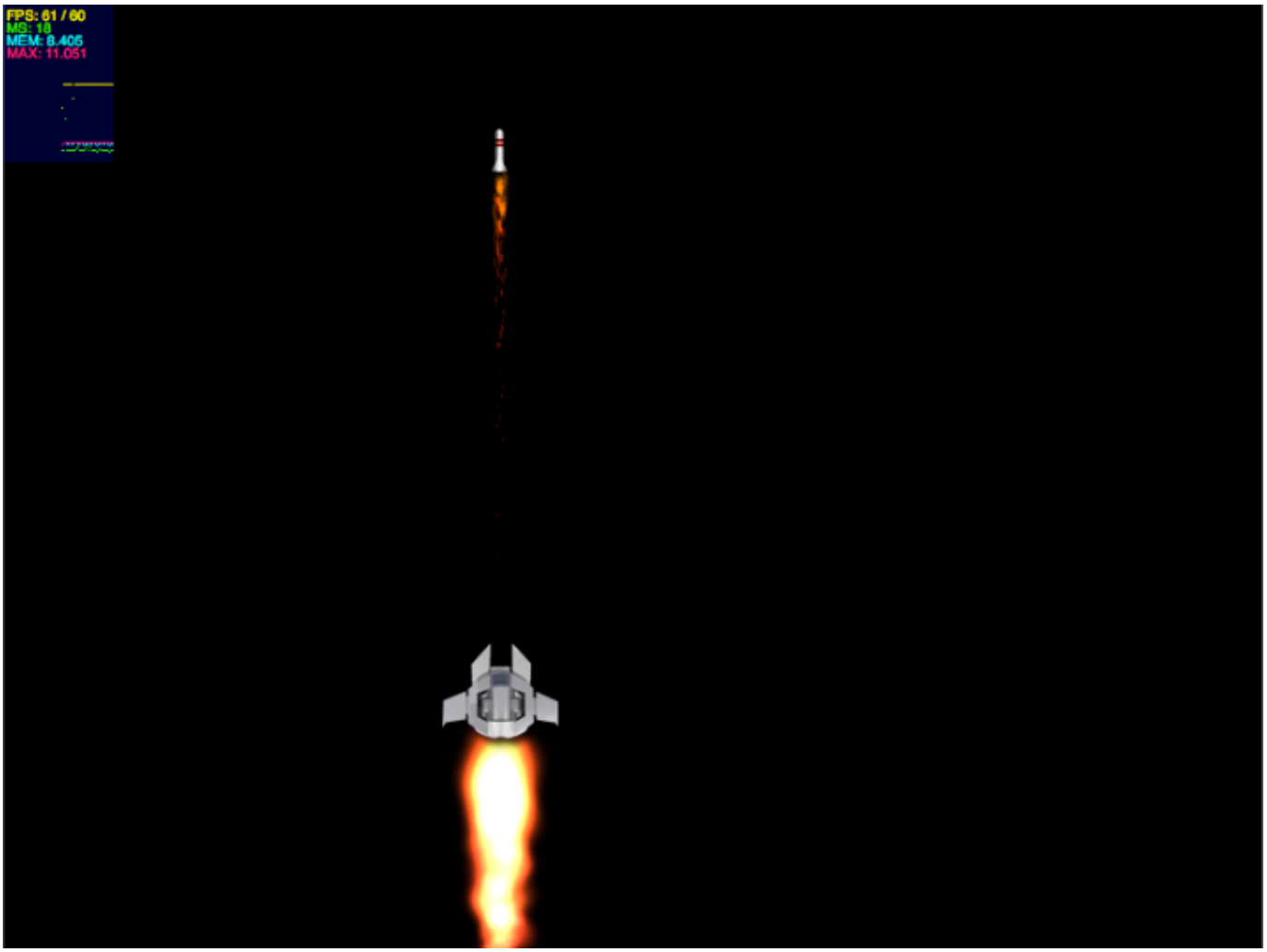

Figure 1.54

Rockets with fire.

Once our rockets are moving out of the stage, we need to remove them from the scene and from the juggler:

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

We actually missed here a very important detail, we forgot to dispose the particles, not disposing them would not clean them from the GPU memory, our code above should be changed:

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

You may also use the `removeChild` API which will allow you to remove the particle system and dispose it:

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

There is also another potential scenario that you could encounter. In our previous example, the particles are attached to a rocket and are moving out of the screen, it is then easy to check for the particles current position and dispose them when out of the screen bounds.

Another scenario would be, the particles are exploding at random locationms but do not move out of the screen, they would vanish inside the bounds of the game screen. In such a scenario, testing the location of the particles does not help, you need to check for the completion of the particles animation and dispose them when the animation is complete.

Fortunately, the `ParticleDesignerPS` class exposes a `isComplete` property, informing you about the completion state. We can then iterate over a simple `Vector` of particles inside our game loop, and keep checking the `isComplete` property to dispose particles:

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

Anytime we need to show particles, we would add a reference to the `particlesVector`, our main game loop would handle automatically the disposing of the particles when they are done animating.

We are now done with our introduction to Starling, I hope you enjoyed it, now it is up to you to create amazing content on top of Starling!

Credits

I would like to thank Chris Georgenes (mudbubble.com) for the gorgeous assets used in this book.

I would like to thank Daniel Sperl (author of Starling), Starling is a beautiful framework, working with you on this initiative has been extremely exciting.