# Table of Contents

1. Getting Started ................................................................. 2
   1.1. Introduction ............................................................. 2
   1.2. What is Adobe AIR? .................................................. 2
   1.3. What is Starling? ...................................................... 4
   1.4. Choosing an IDE ...................................................... 6
   1.5. Resources .............................................................. 10
   1.6. Hello World ........................................................... 13
   1.7. Summary ............................................................... 16

2. Basic Concepts ............................................................... 17
   2.1. Configuring Starling .................................................. 17
   2.2. Display Programming ............................................... 24
   2.3. Textures & Images ................................................... 32
   2.4. Dynamic Text ........................................................ 39
   2.5. Event Handling ....................................................... 43
   2.6. Animations ............................................................ 54
   2.7. Asset Management ................................................... 65
   2.8. Fragment Filters ..................................................... 70
   2.9. Meshes ................................................................ 74
   2.10. Masks .................................................................. 81
   2.11. Sprite3D ................................................................. 83
   2.12. Utilities ................................................................. 86
   2.13. Summary ............................................................... 89

3. Advanced Topics ............................................................... 90
   3.1. ATF Textures ............................................................. 90
   3.2. Context Loss ............................................................. 95
   3.3. Memory Management ............................................... 99
   3.4. Performance Optimization ........................................ 108
   3.5. Custom Filters ........................................................ 117
   3.6. Custom Styles ........................................................ 126
   3.7. Distance Field Rendering .......................................... 136
   3.8. Summary ............................................................... 142

4. Mobile Development ......................................................... 143
   4.1. Multi-Resolution Development ................................... 143
   4.2. Device Rotation ....................................................... 162
   4.3. Summary ............................................................... 163

5. Final Words ................................................................. 164
   5.1. Achievement Unlocked! ............................................ 164
   5.2. Getting Help .......................................................... 164
The goal of this manual is to provide an extensive introduction to the Starling Framework. Starling is a cross-platform engine for ActionScript 3 that can be used for all kinds of applications, with a special focus on 2D games.

In this guide, you will learn:

- The technologies Starling was built upon and the principles it follows.
- How to pick an IDE and set it up for your first project.
- The basic concepts like the display list, events and the animation system.
- Advanced techniques, like how to tap into the potential of fragment and vertex programs.
- How to get the best performance out of the framework.
- What’s required to get your game to run on mobile phones and tablets.

Keep on reading to dive right into it!

The Starling Handbook

Are you looking for an extended version of this manual? That’s the Starling Handbook. In addition to all you find here, it adds a plethora of additional information:

- Learn how to create a complete game from start to finish.
- Get inspired by cookbook-like recipes for all tastes.
- Learn first hand about Feathers and Starling Builder.

Find out all about it here: The Starling Handbook
Chapter 1. Getting Started

This chapter will provide you with an overview of Starling and the technologies it builds upon. We will first take a brief look at the history of *Flash* and *AIR* and how Starling came into being. Then, we will evaluate the tools and resources that will help you when working with Starling. At the end of the chapter, you will have your first "Hello World" example up and running.

1.1. Introduction

First of all: welcome to the Starling community! Since its first release in 2011, developers from all over the world have created amazing games and apps with the help of the little red bird. It's great you want to join us!

You have come to the right place: this manual will teach all you need to know. We'll cover everything from 'A' like *Asset Management* to 'Z' like, ehm, *Zen-like coding*.

I will start from scratch, beginning with a small game that will give you a feeling for the framework. Then we'll look at all the concepts in detail, including the display list architecture and Starling's animation system. Later chapters contain information for advanced Starling users, like how to write custom rendering code. Before you know it, you'll be a master of Starling!

However, there is one small precondition: you should have a basic understanding of *ActionScript 3* (AS3). Fear not: if you have used any other object oriented language, you will get the hang of it quickly. There are numerous books and tutorials available that will get you started.

If you ask me for one book in particular, I can recommend "Essential ActionScript 3" by Colin Moock. When I started to work with AS3, it taught me all the important nuts and bolts, and I'm still taking it off the bookshelf from time to time (especially with AS3's weird E4X syntax for XML processing).

AS3 vs. JS

Did you know that *ActionScript 3* was actually designed to become a successor to *JavaScript*? As you might know, *JavaScript* is an implementation of the *ECMAScript* language specification; and *ActionScript 3* is based on *ECMAScript 4*. However, due to mostly political reasons, *ECMAScript 4* was abandoned and never made it into browsers; today, *ActionScript 3* remains the only implementation of the standard. Ironically, though, modern versions of *JavaScript* look more and more like *ActionScript 3*.

1.2. What is Adobe AIR?

Just a few years ago, Adobe's *Flash Player* was omnipresent when browsing the web. At that time, it was basically the only choice when you wanted to create interactive or animated content for the web. Browsers had very limited capabilities when it came to video, sound, and animation; and the few features they had were plagued by browser incompatibilities. In short, it was a mess.
That’s why *Adobe Flash* was so popular. It allowed designers and developers to create multimedia content in an intuitive authoring program (now called *Adobe Animate*), and ensured that it would look the same across all platforms. With *ActionScript 3*, it also featured a language that was both easy to learn and very powerful.

*Flash* was originally created by *Macromedia*, which was acquired by Adobe in 2005.

Building on the popularity of its platform, Adobe realized that there was a demand to use the same technology for standalone applications that run outside the browser. That’s what the *Adobe AIR* runtime is for. Applications built with the *AIR SDK* can be deployed as standalone applications that run on desktop (*Windows*, *macOS*) or mobile (*Android*, *iOS*). Its standard library is a superset of the one from Flash, so you can do anything in *AIR* that you can do in *Flash*; but it also provides a huge number of additional APIs for things like file system access or window management.

Of course, when you want to create a desktop application, you also need a way to create a graphical user interface, right? Since standard Flash was not very well suited for this task, this was moved into another SDK: *Flex* (now Apache Flex). Flex also introduced an XML-based markup language (called *MXML*) to define user interface layouts.

For Starling, you don’t need *Flex*, just the *AIR SDK*.

### 1.2.1. The current state of Flash and AIR

At the time of its introduction, *AIR* was part of a trend summarized under the term "Rich Internet Applications" (*RIA*) — a buzzword that was all the rage in the late 2000’s. There was a fierce competition between Adobe’s *AIR* and Microsoft’s *Silverlight* (as well as Sun’s *JavaFX*). Fast forward to the present day, though, and it’s apparent that the tides have turned. The winner is clearly *HTML5/JavaScript*, which is now the most popular technology stack when building applications with web technologies.

I won’t sugarcoat it: the *Flash Plugin* is long past its peak — especially since Adobe’s announcement that they will stop supporting it after 2020. All popular browsers will probably remove the plugin altogether at the same time.

Make no mistake, though: that’s the *Flash plugin* we’re talking about here, not *AIR*. *AIR* is still seeing solid commitment from Adobe’s side, with regular releases and major feature updates throughout each year. Granted, it may no longer be the “cool kid” in town, but it’s still an extremely attractive platform to build software with:

- Compared to the fragmented world of *HTML5*, where trending libraries change faster than Lady Gaga’s costumes, it is very mature and easy to use.

- It comes with an extensive standard library that provides all the tools you need for day-to-day development.

- Especially combined with *Starling* and *Feathers*, it provides one of the smoothest paths for true cross-platform development (targeting all major desktop and mobile platforms with a single code-base).
Talking about *Starling* … how does it fit into this picture?

When it comes to software development, don’t fall into the trap of blindly following the masses. For every problem, there are multiple solutions; some of them better suited than others. Pick the tool you are most comfortable with; a tool that gets out of your way and lets you focus on the software you want to create.

### 1.3. What is Starling?

The *Starling Framework* allows you to create hardware accelerated apps in *ActionScript 3*. The main target is the creation of 2D games, but Starling may be used for any graphical application. Thanks to *Adobe AIR*, Starling-based applications can be deployed to all major mobile and desktop platforms.

![Figure 1. This red little fella represents the logo of the Starling Framework.](image)

While Starling mimics the classic display list architecture of *Adobe AIR/Flash*, it provides a much better performance: all objects are rendered directly by the GPU (using the *Stage3D* API). The complete architecture was designed for working well with the GPU; common game development tasks were built right into its core. Starling hides *Stage3D* internals from developers, but makes it easy to access them for those who need full performance and flexibility.

#### 1.3.1. Why another display API?

As outlined above, Starling’s API is very similar to the native Flash API, namely: the `flash.display` package. So you might ask: why go to all that effort to recreate Flash inside … err, Flash?

The reason is that the original `flash.display` API, with its flexible display list, vector capabilities, text rendering, filters, and whatnot, was designed in an era of desktop computers. Those computers had powerful CPUs, but (by modern standards) primitive, fixed-logic graphics hardware. Today’s mobile hardware, on the other hand, has an almost reversed setup: a weak (i.e. battery-conserving) CPU with a very advanced graphics chip.

The problem: it’s extremely difficult (if not impossible) to change an API that was designed for pure CPU rendering to suddenly use the GPU efficiently. Thus, such attempts failed spectacularly, and the Flash plug-in is nowadays completely gone from browsers on phones and tablets.

To its credit, Adobe was very aware of this issue. That’s why, back in 2011, they introduced a low level graphics API called *Stage3D*. That API is decidedly low-level; it’s basically a wrapper of
OpenGL and DirectX, allowing developers to access the raw power of the GPU.

The problem: such a low-level API didn’t help users of the classic display list much, at least not right away. The Stage3D API is as low-level as it gets, so it’s nothing a typical developer can (or should!) directly work with when creating an app or game. Clearly, Adobe needed a more high-level API, built on top of Stage3D, but as easy to use as flash.display.

Well ... this is where Starling enters the stage (pun intended)! It was designed from ground up for Stage3D, while mimicking the classic Flash API as much as possible. This makes it possible to fully leverage today’s powerful graphics hardware, while using concepts countless developers are already familiar with.

Adobe could have made such an API themselves, of course. However, monolithic APIs built by big companies have the tendency to become big and inflexible. A small open source project, on the other hand, powered by a community of equal-minded developers, can act much more swiftly. That’s the insight that led Thibault Imbert, product manager of the Flash and AIR platforms in 2011, to initiate the Starling project.

To this day, Starling is funded and supported by Adobe.

There is an interesting blog article in which one of Adobe’s engineers describes the reasons for supporting Starling instead of providing an accelerated version of the classic display list. It’s still available via archive.org.

1.3.2. Starling’s philosophy

One of the core aims of Starling was to make it as lightweight and easy to use as possible. In my opinion, an open source library should not only be easy to use—it should also encourage diving into the code. I want developers to be able to understand what’s going on behind the scenes; only then will they be able to extend and modify it until it perfectly fits their needs.

That’s why Starling’s source is well documented and surprisingly concise. With a size of just about 15k lines of code, it’s probably smaller than most games that are written with it!

I really want to emphasize that: if you’re one day stuck or confused why your code isn’t working as expected, don’t hesitate to step into the source code of Starling. Oftentimes, you’ll quickly see what’s going wrong, and you’ll get a much better understanding of its internals.

Another important goal of Starling is, of course, its close affinity to the display list architecture. That’s not only because I really like the whole idea behind the display list, but also to make it easy for developers to transition to Starling.

Nevertheless, I was never trying to create a perfect duplicate. Targeting the GPU requires specific concepts, and those should shine through! Concepts like Textures and Meshes aim to blend in seamlessly with the original API, just as if it had always been designed for the GPU.
1.4. Choosing an IDE

As you have just read, Starling apps and games are built using the Adobe AIR SDK. Technically, you could just use a text editor and the command line to compile and deploy your code, but that's not recommended. Instead, you'll definitely want to use an integrated development environment (IDE). That will make debugging, refactoring and deployment much easier. Thankfully, there are several to choose from. Let's look at all the candidates!

The prices specified below were effective in April 2017. Please check the product sites for the very latest prices and plans.

1.4.1. Adobe Flash Builder

*Flash Builder* is the official IDE from Adobe; you might also know it under its former name *Flex Builder*. You can either purchase it as a standalone version (in a standard and premium edition) or get it as part of a *Creative Cloud* subscription.

Built upon *Eclipse*, it is a very powerful piece of software, supporting all the features you'd expect, like mobile debugging and refactoring. The premium edition even includes a very useful performance profiler.

Personally, I used *Flash Builder* for a very long time, and the Starling download even comes with suitable project files. However, there is one caveat: *Flash Builder* has apparently been abandoned by Adobe. The last update (version 4.7) was released in late 2012, and it wasn't particularly stable. There are no indications that this situation will change anytime soon.

Thus, I can only recommend it if you are a *Creative Cloud* user anyway (because then, you'll get it for free), or if you've got an old license lying around somewhere. Don't get me wrong: it has a great set of features, and you will get stuff done with it. But you will have to live with occasional crashes, and updating the AIR SDK is a chore.

*Platforms*

  - Windows, macOS.

*Price*

1.4.2. IntelliJ IDEA

The next candidate might be called "the IDE to rule them all", because IDEA supports a plethora of languages and platforms. AIR support is handled via the plug-in "Flash/Flex Support".

I've been using IDEA for quite a while, and I really like it (especially for its powerful refactoring features). Feature-wise, it feels just like it was built for AS3; all the important parts are in place.

Different to Flash Builder, the IDE receives regular updates. Unfortunately, that's not the case for the Flash plug-in in particular, though. There are some (minor) deficits that have been waiting for a fix for quite a while.

That's all just nitpicking, though. It's an excellent IDE, and it's my recommendation if you're on macOS. The only caveat might be that JetBrains recently switched to a subscription based pricing model, which might not be attractive for everybody.

There's also a free community edition of IDEA, but unfortunately it doesn't include the Flash/Flex Support plug-in.

Platforms

Windows, macOS.

Price

For individuals: USD 149 (first year), USD 119 (second year), USD 89 (third year onwards). Business licenses are more expensive (USD 499/399/299 per year).

The subscription model contains a so-called "perpetual fallback license", which means that after 12 months, you'll be able to keep a version of IDEA even if you cancel the subscription. Personally, I think this mitigates the downsides of the subscription model.
1.4.3. FlashDevelop

As much as I love working on macOS, here’s an area where I occasionally envy Windows users: they have access to an excellent free (open source) IDE for Starling development: FlashDevelop. It has been around since 2005 and is still seeing updates on a regular basis. If you’re into Haxe, it has you covered, as well.

Since I’m primarily using macOS, I don’t have very much first-hand experience with FlashDevelop; but from countless posts in the Starling forum, I’ve heard only good about it. Some people are even using it on the Mac via a virtual machine (like Parallels).

**Platforms**

Windows only.

**Price**

Free and open source.
1.4.4. PowerFlasher FDT

Just like Flash Builder, FDT is built on the Eclipse platform. Thus, it’s a great choice when moving away from Flash Builder, since it all looks and feels very similar. You can even import all your Flash Builder projects.

FDT does improve on Flash Builder in several areas; for example, you can easily change the type of an existing project from Flash to AIR — something that is impossible in Flash Builder. It also supports several additional languages, like HTML5/JavaScript, Haxe and PHP.

All in all, it’s a very solid IDE. If you like Eclipse, you can’t go wrong with FDT!

There is even a free edition available, which is a great way to get started. Contrary to what the product page suggests, you can also use it for mobile AIR development.

Platforms

Windows, macOS.

Price

Between USD 25 and USD 55 per month (depending on contract length). Students and teachers may apply for special terms.

1.4.5. Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor that’s available for all major desktop operating systems. It does not provide any ActionScript support out of the box, but there is an extension in the works that might make it a viable (and free!) alternative to the other IDEs soon.

The NextGen ActionScript extension is being developed by Josh Tynjala, who has also brought you the fantastic Feathers user interface components. It turns Visual Studio Code into a fully-fledged ActionScript development environment, including code intelligence features like auto-completion, automatic class imports, etc. At the time of this writing, Josh just added support for debugging SWF files in AIR or Flash Player — the deal-breaker for any serious development environment.
Being brand-new, it's still a little rough around the edges, but (knowing Josh) I have no doubt that this extension is going to become rock-solid in the future. Definitely keep an eye out for this one!

**Platforms**
- Windows, macOS.

**Price**
- Free and open source.

---

### 1.4.6. Adobe Animate

If you’re a designer or a developer who has been using Flash for a very long time, you might wonder where *Adobe Flash Professional* is coming up in this list. Well, here it is! If you're not recognizing it, that's because Adobe recently renamed it to *Adobe Animate*. That actually makes a lot of sense, since the new name reflects a major change in its focus. *Animate* is now a general-purpose animation tool, supporting output not only to Flash, but also *HTML5*, *WebGL*, and *SVG* formats.

While you can use *Animate* for Starling, I wouldn’t recommend it. It’s a fantastic tool for designers, but it simply wasn’t built for writing code. You’ll be much better off using it just for the graphics, and writing the code in one of the other mentioned IDEs.

**Platforms**
- Windows, macOS.

**Price**

---

### 1.5. Resources

We almost have all the building blocks in place now. Let’s see how to set everything up so that we can finally get started with our first project.
1.5.1. AIR SDK

If you have picked and installed your IDE, the next step is to download the latest version of the AIR SDK. Adobe releases a new stable version once every three months, so be sure to keep up to date! Each new release typically contains several important bug-fixes, which is especially important to maintain compatibility with the latest mobile operating systems. You will also constantly see the team experimenting with new features, and I’m trying hard to keep up with that pace in Starling.

The latest release can always be found here: Download Adobe AIR SDK

Starling 2 requires at least AIR 19.

1.5.2. Flash Player Projector

If your project also aims to run in the Flash Player, I recommend you get the standalone version of it, called Projector (available as Debug and Release versions). The advantage of the projector is a much easier debugging experience. Yes, you could also debug via the browser (after installing the debug-version of the plug-in)—but personally, I find that extremely cumbersome. The projector starts up much faster, and you don’t need to configure any HTML files to get it going.

This page contains a list of all Flash Player versions suitable for developers. Look for the "projector content debugger": Adobe Flash Player Debug Downloads

The debugger is significantly slower than the standard version. Keep that in mind when you are working on performance optimization.

Your IDE might need to know how to find the correct player. For example, in IDEA, this setting is part of the debug configurations screen; other IDEs might simply use the system default. In any case, it’s important to always use a player version that’s equal to or higher than the AIR SDK version you compiled the SWF with.

1.5.3. Starling

Now all that’s left to download is Starling itself. You can choose between two ways of doing that.

a. Download the latest release as a zip-file from the Gamua website.

b. Clone Starling’s Git repository.

The advantage of the former is that it comes with a precompiled SWC file (found in the folder starling/bin) that is easily linked to your project. However, you’ll always only get the stable releases this way, which means you’re missing out on the latest hot updates and fixes! For this reason, I’d rather advocate using Git.

Let’s say you report a bug and it is fixed a few days later (yes, that happens!): with the standard download, you’ll have to wait for the fix until I’m creating a new stable release, which could be quite a while. When you’re using the Git repository, you’ll have the fix right away.
Going into depths about *Git* would exceed the scope of this manual, but you'll find lots of great tutorials about it on the web. Once installed, *Git* can *clone* the complete Starling repository to your disk with the following command:

```
git clone https://github.com/Gamua/Starling-Framework.git
```

This will copy Starling to the folder *Starling-Framework*. Look for the actual source code inside the sub-folder *starling/src*. All of the mentioned IDEs support adding this folder as a *source path* to your project, which will make Starling part of your project. That's not any more complicated than linking to the SWC file; and as a neat side effect, you will even be able to step into Starling's source on debugging.

But what's best about this approach is how easy it is to update Starling to the latest version. Simply navigate into the repository's directory and *pull*:

```
cd Starling-Framework
git pull
```

That's much simpler than opening up the browser and manually downloading a new version, isn't it?

Some additional information for advanced *Git*-users:

- All day-to-day development in Starling happens on the *master* branch.
- Stable releases are *tagged* (like v2.0, v2.0.1, v2.1).
- Each tag is marked as a *Release* on *GitHub*, at which time I'll also attach a precompiled SWC file.

### 1.5.4. Getting Help

The best of us get stuck sometimes. You might hit a road block because of a bug in Starling, or maybe because of a problem that seems impossible to solve. Either way, the Starling community won't leave you alone in your misery! Here are some resources that you can go to when you need help.

**Starling Forum**

That's the main hub of the Starling community. With dozens of new posts each day, it's very likely that your problem has already been asked before, so be sure to make use of the *search* feature. If that doesn't help, feel free to register an account and ask away. One of the most friendly and patient communities you'll find on the web!

- [http://forum.starling-framework.org](http://forum.starling-framework.org)

**Starling Manual**

The online manual you are reading right now. I try my best to keep it updated for each new release.

Starling Wiki
The wiki contains links and articles about different Starling related topics, and most importantly: a list of Starling extensions. We will discuss some of those later.
→ http://wiki.starling-framework.org

API Reference
Don’t forget to consult the Starling API Reference for detailed information about all classes and methods.
→ http://doc.starling-framework.org

Gamua Blog
Keep up to date with the latest news about Starling via the Gamua blog. I must admit I’m a little lazy when it comes to writing blog posts, but there will always be at least one for each Starling release.
→ http://gamua.com/blog

Twitter
I’m using several social networks, but the best way to reach me is via @Gamua. Follow this account to get updates about new developments or links to other Starling-powered games.
→ https://twitter.com/Gamua

1.6. Hello World
Phew, that was quite a lot of background information. It’s time we finally get our hands dirty! And what better way to do that than a classical “Hello World” program. This manual wouldn’t be complete without one, right?

1.6.1. Checklist
Here’s a quick summary of the preparations you should already have made:

• Chosen and downloaded an IDE.
• Downloaded the latest version of the AIR SDK.
• Downloaded the latest version of the Flash Player Projector.
• Downloaded the latest version of Starling.
• Configured your IDE to use the correct SDK and player.

Configuring the IDE and setting up a project is done slightly different in each IDE. To help you with that, I created a specific how-to for each IDE in the Starling Wiki. Please follow the appropriate tutorial before you continue.

Admittedly, all of those set-up procedures are a pain. But bear with me: you only need to do this very rarely.
1.6.2. Startup Code

Create a new project or module in your IDE; I recommend you start with a Flash Player project (target platform: Web) with the name "Hello World". As part of the initialization process, your IDE will also setup a minimal startup class for you. Let's open it up and modify it as shown below. (Typically, that class is named like your project, so exchange the class name below with the correct one.)

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

    [SWF(width="400", height="300", frameRate="60", backgroundColor="#808080")]
    public class HelloWorld extends Sprite {
        private var _starling:Starling;

        public function HelloWorld()
        {
            _starling = new Starling(Game, stage);
            _starling.start();
        }
    }
}
```

This code creates a Starling instance and starts it right away. Note that we pass a reference to the “Game” class into the Starling constructor. Starling will instantiate that class once it is ready. (It’s done that way so you don’t have to take care about doing stuff in the right order.)

That class first needs to be written, of course. Add a new class called Game to your project and add the following code:
package
{
    import starling.display.Quad;
    import starling.display.Sprite;
    import starling.utils.Color;

    public class Game extends Sprite
    {
        public function Game()
        {
            var quad:Quad = new Quad(200, 200, Color.RED);
            quad.x = 100;
            quad.y = 50;
            addChild(quad);
        }
    }
}

The class just displays a simple red quad to see if we've set up everything correctly.

Note that the Game class extends starling.display.Sprite, not flash.display.Sprite! This is crucial, because we're in the Starling world now. It's completely separate from the flash.display package.

1.6.3. First Launch

Now start up the project. For some of you, the output might be a little anticlimactic, because you are seeing an error message like this:

Context3D not available! Possible reasons: wrong wmode or missing device support.

In that case, it was probably the browser that opened up instead of the standalone Flash Player. Check the run/debug configuration and make sure the Flash Player Projector (debug version) is used, not the browser. That should fix the problem.

Fixing the browser error

One day, though, you'll want to embed your SWF file into an HTML page. In that case, you can fix the error by changing the wmode Flash parameter to direct in the HTML file that's embedding the SWF file. Typically, this means you have to make the following change:

// find the following line ...  
var params = {};  

// ... and add that one directly below:  
params.wmode = "direct";
Fixing the AIR error

You will also see this error if you created an AIR application instead of an SWF file. In that case, you will need to edit the AIR application descriptor, which is probably called HelloWorld-app.xml or similar. Find the renderMode XML node (which might be commented out) and change its value to direct.

Find this:
<!-- <renderMode></renderMode> -->

Replace with this:
<renderMode>direct</renderMode>

What we've been doing here is allowing the runtime to access the GPU. Without those changes, Stage3D is simply not accessible.

1.6.4. Fixed Launch

Congratulations! You have successfully compiled and run your first Starling based project.

Figure 2. Fantastic: a red Starling in a red box.

Seriously: the most daunting part now lies behind you. Finally, we are ready to dig into a real project!

1.7. Summary

You should now have a basic understanding of Starling and the tools and resources you will work with. You know how to acquire and install the Adobe AIR SDK, and you picked the IDE that fits your needs best. The Starling-powered "Hello World" application proved that everything is set up correctly.

It's time to dive into the actual Starling API!
Chapter 2. Basic Concepts

Starling might be a compact framework, but it still boasts a significant number of packages and classes. It is built around several basic concepts that are designed to complement and extend each other. Together, they provide you with a set of tools that empower you to create any application you can imagine.

**Display Programming**

Every object that is rendered on the screen is a *display object*, organized in the *display list*.

**Textures & Images**

To bring pixels, forms and colors to the screen, you will learn to use the *Texture* and *Image* classes.

**Dynamic Text**

Rendering of dynamic text is a basic task in almost every application.

**Event Handling**

Communication is key! Your display objects need to talk to each other, and they can do that via Starling's powerful event system.

**Animation**

Bring some motion into the picture! There are different strategies to animate your display objects.

**Asset Management**

Learn how to load and organize all kinds of assets, like textures and sounds.

**Special Effects**

Effects and filters that will make your graphics stand out.

**Utilities**

A number of helpers to make your life easier.

We've got a lot of ground to cover, so, in the words of Super Mario: "Let's-a go!"

### 2.1. Configuring Starling

The first step of every Starling-powered application: creating an instance of the *Starling* class (package *starling.core*). Here is the complete declaration of Starling's constructor:
```java
public function Starling(
    rootClass:Class,
    stage:Stage,
    viewPort:Rectangle = null,
    stage3D:Stage3D = null,
    renderMode:String = auto,
    profile:Object = auto);
```

**rootClass**

The class that is instantiated as soon as Stage3D has finished initializing. Must be a subclass of `starling.display.DisplayObject`.

**stage**

The traditional Flash stage that will host Starling’s content. That’s how the Starling and Flash display list are connected to each other.

**viewPort**

The area within the Flash stage that Starling will render into. Since this is often the full stage size, you can omit this parameter (i.e. pass `null`) to use the full area.

**stage3D**

The Stage3D instance used for rendering. Each Flash stage may contain several Stage3D instances and you can pick any one of those. However, the default parameter (`null`) will suffice most of the time: this will make Starling use the first available Stage3D object.

**renderMode**

The whole idea behind Stage3D is to provide hardware-accelerated rendering. However, there is also a software fallback mode; it may be forced by passing `Context3DRenderMode.SOFTWARE`. The default (`auto`) is recommended, though — it means that software rendering is used only when there’s no alternative.

**profile**

Stage3D provides a set of capabilities that are grouped into different profiles (defined as constants within `Context3DProfile`). The better the hardware the application is running on, the better the available profile. The default (`auto`) simply picks the best available profile.

Most of these arguments have useful default values, so you will probably not need to specify all of them. The code below shows the most straightforward way to start Starling. We are looking at the `Main` class of a Flash Player or AIR project.
package {
    import flash.display.Sprite;
    import starling.core.Starling;

    [SWF(width="640", height="480",
         backgroundColor="#808080",
         frameRate="60")]
    public class Main extends Sprite {
        private var _starling:Starling;

        public function Main() {
            _starling = new Starling(Game, stage);
            _starling.start();
        }
    }
}

Note that the class extends flash.display.Sprite, not the Starling variant. That's simply a necessity of all Main classes in AS3. However, as soon as Starling has finished starting up, the logic is moved over to the Game class, which builds our link to the starling.display world.

Configuring the Frame Rate

Some settings are configured right in the "SWF" MetaData tag in front of the class. The frame rate is one of them. Starling itself does not have an equivalent setting: it always simply uses the frameRate from the native stage. To change it at runtime, access the nativeStage property:

Starling.current.nativeStage.frameRate = 60;

Starling's setup process is asynchronous. That means that you won't yet be able to access the Game instance at the end of the Main method. However, you can listen to the ROOT_CREATED event to get notified when the class has been instantiated.
public function Main()
{
    _starling = new Starling(Game, stage);
    _starling.addEventListener(Event.ROOT_CREATED, onRootCreated);
    _starling.start();
}

private function onRootCreated(event:Event, root:Game):void
{
    root.start(); // 'start' needs to be defined in the 'Game' class
}

2.1.1. The ViewPort

Stage3D provides Starling with a rectangular area to draw into. That area can be anywhere within the native stage, which means: anywhere within the area of the Flash Player or the application window (in case of an AIR project).

In Starling, that area is called the viewPort. Most of the time, you will want to use all of the available area, but sometimes it makes sense to limit rendering to a certain region.

Think of a game designed in an aspect ratio of 4:3, running on a 16:9 screen. By centering the 4:3 viewPort on the screen, you will end up with a "letterboxed" game, with black bars at the left and right.

You can't talk about the viewPort without looking at Starling's stage, as well. By default, the stage will be exactly the same size as the viewPort. That makes a lot of sense, of course: a device with a display size of 1024 × 768 pixels should have an equally sized stage.

You can customize the stage size, though. That's possible via the stage.stageWidth and stage.stageHeight properties:

```javascript
stage.stageWidth = 1024;
stage.stageHeight = 768;
```

But wait, what does that even mean? Is the size of the drawing area now defined by the viewPort or the stage size?

Don't worry, that area is still only set up by the viewPort, as described above. Modifying the stageWidth and stageHeight doesn't change the size of the drawing area at all; the stage is always stretched across the complete viewPort. What you are changing, though, is the size of the stage's coordinate system.

That means that with a stage width of 1024, an object with an x-coordinate of 1000 will be close to the right edge of the stage; no matter if the viewPort is 512, 1024, or 2048 pixels wide.

That becomes especially useful when developing for HiDPI screens. For example, Apple's iPad exists in a normal and a "retina" version, the latter doubling the number of pixel rows and columns
(yielding four times as many pixels). On such a screen, the interface elements should not become smaller; instead, they should be rendered more crisply.

By differentiating between the viewPort and the stage size, this is easily reproduced in Starling. On both device types, the stage size will be 1024×768; the viewPort, on the other hand, will reflect the size of the screen in pixels. The advantage: you can use the same coordinates for your display objects, regardless of the device on which the application is running.

**Points vs. Pixels**

If you think this through, you’ll see that on such a retina device, an object with an x-coordinate of 1 will actually be two pixels away from the origin. In other words, the unit of measurement has changed. We are no longer talking about pixels, but points! On a low-resolution screen, one point equals one pixel; on a HiDPI screen, it’s two pixels (or more, depending on the device).

To find out the actual width (in pixels) of a point, you can simply divide viewPort.width by stage.stageWidth. Or you use Starling’s contentScaleFactor property, which does just that.

```javascript
starling.viewPort.width = 2048;
starling.stage.stageWidth = 1024;
trace(starling.contentScaleFactor); // -> 2.0
```

I will show you how to make full use of this concept in the Mobile Development chapter.

### 2.1.2. Context3D Profiles

The platforms Starling is running on feature a wide variety of graphics processors. Of course, those GPUs have different capabilities. The question is: how to differentiate between those capabilities at runtime?

That’s what Context3D profiles (also called render profiles) are for.

**What is a Context3D?**

When using Stage3D, you are interacting with a rendering pipeline that features a number of properties and settings. The context is the object that encapsulates that pipeline. Creating a texture, uploading shaders, rendering triangles — that’s all done through the context.

Actually, Starling makes every effort to hide any profile limitations from you. To ensure the widest possible reach, it was designed to work even with the lowest available profile. At the same time, when running in a higher profile, it will automatically make best use of it.

Nevertheless, it might prove useful to know about their basic features. Here’s an overview of each profile, starting with the lowest.

**BASELINE_CONSTRAINED**

If a device supports Stage3D at all, it must support this profile. It comes with several mean
limitations, e.g. it only supports textures with side-lengths that are powers of two, and the length of shaders is very limited. That profile is mainly found on old desktop computers.

**BASELINE**

The minimum profile to be found on mobile devices. Starling runs well with this profile; the removal of the power-of-two limitation allows for more efficient memory usage, and the length of shader programs is easily sufficient for its needs.

**BASELINE_EXTENDED**

Raises the maximum texture size from 2048x2048 to 4096x4096 pixels, which is crucial for high-resolution devices.

**STANDARD_CONSTRAINED, STANDARD, STANDARD_EXTENDED**

Starling currently doesn’t need any of the features coming with these profiles. They provide additional shader commands and other low-level enhancements.

My recommendation: simply let Starling pick the best available profile (auto) and let it deal with the implications.

---

**Maximum Texture Size**

There's only one thing you need to take care of yourself: making sure that your textures are not too big. The maximum texture size is accessible via the property `Texture.maxSize`, but only after Starling has finished initializing.

---

**2.1.3. Native Overlay**

The main idea behind Starling is to speed up rendering with its Stage3D driven API. However, there’s no denying it: the classic display list has many features that Starling simply can’t offer. Thus, it makes sense to provide an easy way to mix-and-match features of Starling and classic Flash.

The `nativeOverlay` property is the easiest way to do so. That’s a conventional `flash.display.Sprite` that lies directly on top of Starling, taking `viewPort` and `contentScaleFactor` into account. If you need to use conventional Flash objects, add them to this overlay.

Beware, though, that conventional Flash content on top of Stage3D can lead to performance penalties on some (mobile) platforms. For that reason, always remove all objects from the overlay when you don’t need them any longer.

---

Before you ask: no, you can’t add any conventional display objects below Starling display objects. The Stage3D surface is always at the bottom; there’s no way around that.

---

**2.1.4. Skipping Unchanged Frames**

It happens surprisingly often in an application or game that a scene stays completely static for several frames. The application might be presenting a static screen or wait for user input, for example. So why redraw the stage at all in those situations?

That’s exactly the point of the `skipUnchangedFrames` property. If enabled, static scenes are recognized...
as such and the back buffer is simply left as it is. On a mobile device, the impact of this feature can't
be overestimated. There's simply no better way to enhance battery life!

I'm already hearing your objection: if this feature is so useful, why isn't it activated by default?
There must be a catch, right?

Indeed, there is: it doesn't work well with Render- and VideoTextures. Changes in those textures
simply won't show up. It's easy to work around that, though: either disable skipUnchangedFrames
temporarily while using them, or call stage.setRequiresRedraw() whenever their content changes.

Now that you know about this feature, make it a habit to always activate it! In the meantime, I hope
that I can solve the mentioned problems in a future Starling version.

On mobile platforms, there's another limitation you should be aware of: as soon
as there's any content on the native (Flash) stage (e.g. via Starling's
nativeOverlay), Starling can't skip any frames. That's the consequence of a
Stage3D limitation.

2.1.5. The Statistics Display

When developing an application, you want as much information as possible about what's going on.
That way, you will be able to spot problems early and maybe avoid running into a dead end later.
The statistics display helps with that.

```javascript
_starling.showStats = true;
```

![Figure 3. The statistics display (by default at the top left).](image)

What's the meaning of those values?

- The framerate should be rather self-explanatory: the number of frames Starling managed to
  render during the previous second.
- Standard memory is, in a nutshell, what your AS3 objects fill up. Whether it's a String, a Sprite, a
  Bitmap, or a Function: all objects require some memory. The value is given in megabytes.
- GPU memory is separate from that. Textures are stored in graphics memory, as are vertex
  buffers and shader programs. Most of the time, textures will overshadow everything else.
- The number of draw calls indicates how many individual “draw“-commands are sent to the GPU
  in each frame. Typically, a scene renders faster when there are fewer draw calls. We will look in
detail at this value when we talk about Performance Optimization.

You might notice that the background color of the statistics display alternates between black and
dark green. That's a subtle clue that's referring to the skipUnchangedFrames property: whenever the
You can customize the location of the statistics display on the screen via the method `showStatsAt`.

**2.2. Display Programming**

With all the setup procedures out of the way, we can start to actually put some content onto the screen!

In every application you create, one of your main tasks will be to split it up into a number of logical units. More often than not, those units will have a visual representation. In other words: each unit will be a **display object**.

**2.2.1. Display Objects**

All elements that appear on screen are types of display objects. The `starling.display` package includes the abstract `DisplayObject` class; it provides the basis for a number of different types of display objects, such as images, movie clips, and text fields, to name just a few.

The `DisplayObject` class provides the methods and properties that all display objects share. For example, the following properties are used to configure an object's location on the screen:

- **x, y**: the position in the current coordinate system.
- **width, height**: the size of the object (in points).
- **scaleX, scaleY**: another way to look at the object size; `1.0` means unscaled, `2.0` doubles the size, etc.
- **rotation**: the object's rotation around its origin (in radians).
- **skewX, skewY**: horizontal and vertical skew (in radians).

Other properties modify the way the pixels appear on the screen:

- **blendMode**: determines how the object's pixels are blended with those underneath.
- **filter**: special GPU programs (shaders) that modify the look of the object. Filters can e.g. blur the object or add a drop shadow.
- **mask**: masks cut away all parts that are outside a certain area.
- **alpha**: the opacity of the object, from `0` (invisible) to `1` (fully opaque).
- **visible**: if false, the object is hidden completely.

Those are the basics that every display object must support. Let's look at the class hierarchy around this area of Starling's API:
You'll notice that the diagram is split up in two main sub-branches. On the one side, there are a couple of classes that extend Mesh: Quad, Image, and MovieClip.

Meshes are a fundamental part of Starling's rendering architecture. Everything that is drawn to the screen is a mesh, actually! Stage3D cannot draw anything but triangles, and a mesh is nothing else than a list of points that spawn up triangles.

On the other side, you'll find a couple of classes that extend DisplayObjectContainer. As its name suggests, this class acts as a container for other display objects. It makes it possible to organize display objects into a logical system — the display list.

2.2.2. The Display List

The hierarchy of all display objects that will be rendered is called the display list. The Stage makes up the root of the display list. Think of it as the literal "stage": your users (the audience) will only see objects (actors) that have entered the stage. When you start Starling, the stage will be created automatically for you. Everything that's connected to the stage (directly or indirectly) will be rendered.

When I say "connected to", I mean that there needs to be a parent-child relationship. To make an object appear on the screen, you make it the child of the stage, or any other DisplayObjectContainer that's connected to the stage.
Figure 4. Display objects are organized in the display list.

The first (and, typically: only) child of the stage is the application root: that’s the class you pass to the Starling constructor. Just like the stage, it’s probably going to be a DisplayObjectContainer. That’s where you take over!

You will create containers, which in turn will contain other containers, and meshes (e.g. images). In the display list, those meshes make up the leaves: they cannot have any child objects.

Since all of this sounds very abstract, let’s look at a concrete example: a speech bubble. To create a speech bubble, you will need an image (for the bubble), and some text (for its contents).

Those two objects should act as one: when moved, both image and text should follow along. The same applies for changes in size, scaling, rotation, etc. That can be achieved by grouping those objects inside a very lightweight DisplayObjectContainer: the Sprite.

**DisplayObjectContainer vs. Sprite**

DisplayObjectContainer and Sprite can be used almost synonymously. The only difference between those two classes is that one (DisplayObjectContainer) is abstract, while the other (Sprite) is not. Thus, you can use a Sprite to group objects together without the need of a subclass. The other advantage of Sprite: it’s just much faster to type. Typically, that’s the main reason why I’m preferring it. Like most programmers, I’m a lazy person!

So, to group text and image together, you create a sprite and add text and image as children:

```javascript
var sprite:Sprite = new Sprite(); ①
var image:Image = new Image(texture);
var textField:TextField = new TextField(200, 50, "¡Ay, caramba!");
sprite.addChild(image); ②
sprite.addChild(textField); ③
```

① Create a sprite.
② Add an *Image* to the sprite.

③ Add a *TextField* to the sprite.

The order in which you add the children is important— they are placed like layers on top of each other. Here, *textField* will appear in front of *image*.

![Sprite with Image and TextField](image)

*Figure 5. A speech bubble, made up by an image and a TextField.*

Now that those objects are grouped together, you can work with the sprite just as if it was just one object.

```javascript
var numChildren:int = sprite.numChildren; ①
var totalWidth:Number = sprite.width; ②
sprite.x += 50; ③
sprite.rotation = deg2rad(90); ④
```

① Query the number of children. Here, the result will be 2.

② width and height take into account the sizes and positions of the children.

③ Move everything 50 points to the right.

④ Rotate the group by 90 degrees (Starling always uses radians).

In fact, *DisplayObjectContainer* defines many methods that help you manipulate its children:

```javascript
function addChild(child:DisplayObject):void;
function addChildAt(child:DisplayObject, index:int):void;
function contains(child:DisplayObject):Boolean;
function getChildAt(index:int):DisplayObject;
function getChildIndex(child:DisplayObject):int;
function removeChild(child:DisplayObject, dispose:Boolean=false):void;
function removeChildAt(index:int, dispose:Boolean=false):void;
function swapChildren(child1:DisplayObject, child2:DisplayObject):void;
function swapChildrenAt(index1:int, index2:int):void;
```

### 2.2.3. Coordinate Systems

Every display object has its own coordinate system. The x and y properties, for example, are not given in screen coordinates: they are always depending on the current coordinate system. That coordinate system, in turn, is depending on your position within the display list hierarchy.

To visualize this, imagine pinning sheets of paper onto a pinboard. Each sheet represents a coordinate system with a horizontal x-axis and a vertical y-axis. The position you stick the pin through is the root of the coordinate system.
Figure 6. Coordinate systems act like the sheets on a pinboard.

Now, when you rotate the sheet of paper, everything that is drawn onto it (e.g. image and text) will rotate with it — as do the x- and y-axes. However, the root of the coordinate system (the pin) stays where it is.

The position of the pin therefore represents the point the x- and y-coordinates of the sheet are pointing at, relative to the parent coordinate system (= the pin-board).

Keep the analogy with the pin-board in mind when you create your display hierarchy. This is a very important concept you need to understand when working with Starling.

### 2.2.4. Custom Display Objects

I mentioned this already: when you create an application, you split it up into logical parts. A simple game of chess might contain the board, the pieces, a pause button and a message box. All those elements will be displayed on the screen — thus, each will be represented by a class derived from `DisplayObject`.

Take a simple message box as an example.

Figure 7. A game's message box.

That's actually quite similar to the speech bubble we just created; in addition to the background image and text, it also contains two buttons.

This time, instead of just grouping the object together in a sprite, we want to encapsulate it into a convenient class that hides any implementation details.

To achieve this, we create a new class that inherits from `DisplayObjectContainer`. In its constructor, we create everything that makes up the message box:
public class MessageBox extends DisplayObjectContainer
{
    [Embed(source = "background.png")]
    private static const BackgroundBmp:Class;

    [Embed(source = "button.png")]
    private static const ButtonBmp:Class;

    private var _background:Image;
    private var _textField:TextField;
    private var _yesButton:Button;
    private var _noButton:Button;

    public function MessageBox(text:String)
    {
        var bgTexture:Texture = Texture.fromEmbeddedAsset(BackgroundBmp);
        var buttonTexture:Texture = Texture.fromEmbeddedAsset(ButtonBmp);

        _background = new Image(bgTexture);
        _textField = new TextField(200, 100, text);
        _noButton = new Button(buttonTexture, "no");
        _yesButton = new Button(buttonTexture, "yes");

        _noButton.x = 20;
        _noButton.y = 100;
        _yesButton.x = 120;
        _yesButton.y = 100;

        addChild(_background);
        addChild(_textField);
        addChild(_noButton);
        addChild(_yesButton);
    }
}

Now you have a simple class that contains a background image, two buttons and some text. To use it, just create an instance of MessageBox and add it to the display tree:

```javascript
var msgBox:MessageBox = new MessageBox("Really exit?");
addChild(msgBox);
```

You can add additional methods to the class (like fadeIn and fadeOut), and code that is triggered when the user clicks one of those buttons. This is done using Starling’s event mechanism, which is shown in a later chapter.

2.2.5. Disposing Display Objects

When you don’t want an object to be displayed any longer, you simply remove it from its parent,
e.g. by calling `removeFromParent()`. The object will still be around, of course, and you can add it to another display object, if you want. Oftentimes, however, the object has outlived its usefulness. In that case, it’s a good practice to call its `dispose` method.

```javascript
msgBox.removeFromParent();
msgBox.dispose();
```

When you dispose display objects, they will free up all the resources that they have allocated. That’s important, because many `Stage3D` related data is not reachable by the garbage collector. When you don’t dispose that data, it will stay in memory, which means that the app will sooner or later run out of resources and crash.

⚠️ When you dispose a container, all of its children will be disposed, too.

To make things easier, `removeFromParent()` optionally accepts a Boolean parameter to dispose the `DisplayObject` that is being removed. That way, the code from above can be simplified to this single line:

```javascript
msgBox.removeFromParent(true);
```

### 2.2.6. Pivot Points

Pivot Points are a feature you won’t find in the traditional display list. In Starling, display objects contain two additional properties: `pivotX` and `pivotY`. The pivot point of an object (also known as `origin`, `root` or `anchor`) defines the root of its coordinate system.

Per default, the pivot point is at `(0, 0)`; in an image, that is the top left position. Most of the time, this is just fine. Sometimes, however, you want to have it at a different position — e.g. when you want to rotate an image around its center.

Without a pivot point, you’d have to wrap the object inside a container sprite in order to do that:

```javascript
var image:Image = new Image(texture);
var sprite:Sprite = new Sprite(); ①
image.x = -image.width / 2.0;
image.y = -image.height / 2.0;
sprite.addChild(image); ②
sprite.rotation = deg2rad(45); ③
```

① Create a sprite.
② Add an image so that its center is exactly on top of the sprite’s origin.
③ Rotating the sprite will rotate the image around its center.

Most long-time Flash developers will know this trick; it was needed quite regularly. One might
argue, however, that it’s a lot of code for such a simple thing. With the pivot point, the code is reduced to the following:

```javascript
var image: Image = new Image(texture);
image.pivotX = image.width / 2.0; ①
image.pivotY = image.height / 2.0; ②
image.rotation = deg2rad(45); ③
```

① Move pivotX to the horizontal center of the image.
② Move pivotY to the vertical center of the image.
③ Rotate around the center.

No more container sprite is needed! To stick with the analogy used in previous chapters: the pivot point defines the position where you stab the pin through the object when you attach it to its parent. The code above moves the pivot point to the center of the object.

![Figure 8. Note how moving the pivot point changes how the object rotates.](image)

Now that you have learned how to control the pivot point coordinates individually, let’s take a look at the method `alignPivot()`. It allows us to move the pivot point to the center of the object with just one line of code:

```javascript
var image: Image = new Image(texture);
image.alignPivot();
image.rotation = deg2rad(45);
```

Handy huh?

Furthermore, if we want the pivot point somewhere else (say, at the bottom right), we can optionally pass alignment arguments to the method:

```javascript
var image: Image = new Image(texture);
image.alignPivot(Align.RIGHT, Align.BOTTOM);
image.rotation = deg2rad(45);
```

That code rotates the object around the bottom right corner of the image.
Gotchas

Be careful: the pivot point is always given in the *local* coordinate system of the object. That’s unlike the *width* and *height* properties, which are actually relative to the *parent* coordinate system. That leads to surprising results when the object is e.g. scaled or rotated.

For example, think of an image that’s 100 pixels wide and scaled to 200% (*image.scaleX = 2.0*). That image will now return a *width* of 200 pixels (twice its original width). However, to center the pivot point horizontally, you’d still set *pivotX* to 50, not 100! In the *local* coordinate system, the image is still 100 pixels wide — it just appears wider in the *parent* coordinate system.

It might be easier to understand when you look back at the code from the beginning of this section, where we centered the image within a parent sprite. What would happen if you changed the *scale* of the sprite? Would this mean that you have to update the position of the image to keep it centered? Of course not. The scale does not affect what’s happening *inside* the sprite, just how it looks from the *outside*. And it’s just the same with the pivot point property.

If you still get a headache picturing that (as it happens to me, actually), just remember to set the pivot point *before* changing the scale or rotation of the object. That will avoid any problems.

2.3. Textures & Images

We came across the *Image* and *Texture* classes several times already; they are some of the most useful classes in Starling. But how are they used, and what’s the difference between the two?

2.3.1. Textures

A texture is just the data that describes an image — like the file that is saved on your digital camera. You can’t show anybody that file alone: it’s all zeros and ones, after all. You need an image viewer to look at it, or send it to a printer.

Textures are stored directly in GPU memory, which means that they can be accessed very efficiently during rendering. They are typically either created from an embedded class or loaded from a file. You can pick between one of the following file formats:

*PNG*

The most versatile of them all. Its lossless compression works especially well for images with large areas of solid color. Recommended as default texture format.

*JPG*

Produces smaller files than PNG for photographic (and photo-like) images, thanks to it lossy encoding method. However, the lack of an alpha channel limits its usefulness severely. Recommended only for photos and big background images.

*ATF*

A format created especially for Stage3D. ATF textures require little texture memory and load very fast; however, their lossy compression is not perfectly suited for all kinds of images. We will
look at ATF textures in more detail in a later chapter (see ATF Textures).

The `starling.textures_Texture` class contains a number of factory methods used to instantiate textures. Here are few of them (for clarity, I omitted the arguments).

```java
public class Texture
{
    static function fromColor():Texture;
    static function fromBitmap():Texture;
    static function fromBitmapData():Texture;
    static function fromEmbeddedAsset():Texture;
    static function fromCamera():Texture;
    static function fromNetStream():Texture;
    static function fromTexture():Texture;
}
```

Probably the most common task is to create a texture from a bitmap. That couldn’t be easier:

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

It’s also very common to create a texture from an embedded bitmap. That can be done in just the same way:

```javascript
[Embed(source="mushroom.png")]
public static const Mushroom:Class;

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

1. Embed the bitmap.
2. Instantiate the bitmap.
3. Create a texture from the bitmap.

However, there is a shortcut that simplifies this further:

```javascript
[Embed(source="mushroom.png")]
public static const Mushroom:Class;

var texture:Texture = Texture.fromEmbeddedAsset(Mushroom);
```

1. Embed the bitmap.
2. Create a texture right from the class storing the embedded asset.
Another feature of the `Texture` class is hidden in the inconspicuous `fromTexture` method. It allows you to set up a texture that points to an area within another texture.

What makes this so useful is the fact that no pixels are copied in this process. Instead, the created `SubTexture` stores just a reference to its parent texture. That’s extremely efficient!

```actionscript
var texture:Texture = getTexture();
var subTexture:Texture = Texture.fromTexture(
    texture, new Rectangle(10, 10, 41, 47));
```

Shortly, you will get to know the `TextureAtlas` class; it’s basically built exactly around this feature.

### 2.3.2. Images

We’ve got a couple of textures now, but we still don’t know how to display it on the screen. The easiest way to do that is by using the `Image` class or one of its cousins.

Let’s zoom in on that part of the family tree.

- A `Mesh` is a flat collection of triangles (remember, the GPU can only draw triangles).
- A `Quad` is a collection of at least two triangles spawning up a rectangle.
- An `Image` is just a quad with a convenient constructor and a few additional methods.
- A `MovieClip` is an image that switches textures over time.

While all of these classes are equipped to handle textures, you will probably work with the `Image`
class most often. That’s simply because rectangular textures are the most common, and the `Image`
class is the most convenient way to work with them.

To demonstrate, let me show you how to display a texture with a `Quad` vs. an `Image`.

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

var quad:Quad = new Quad(texture.width, texture.height); ①
quad.texture = texture;
addChild(quad);

var image:Image = new Image(texture); ②
addChild(image);
```

① Create a quad with the appropriate size and assign the texture, or:

② Create an image with its standard constructor.

Personally, I’d always pick the approach that saves me more keystrokes. What’s happening behind
the scenes is exactly the same in both cases, though.

![Image + Quad = Image](image.png)

*Figure 9. A texture is mapped onto a quad.*

### 2.3.3. One Texture, multiple Images

It’s important to note that a texture can be mapped to any number of images (meshes). In fact,
that’s exactly what you should do: load a texture only once and then reuse it across the lifetime of
your application.

```javascript
// do NOT do this!!
var image1:Image = new Image(Texture.fromEmbeddedAsset(Mushroom));
var image2:Image = new Image(Texture.fromEmbeddedAsset(Mushroom));
var image3:Image = new Image(Texture.fromEmbeddedAsset(Mushroom));

// instead, create the texture once and keep a reference:
var texture:Texture = Texture.fromEmbeddedAsset(Mushroom);
var image1:Image = new Image(texture);
var image2:Image = new Image(texture);
var image3:Image = new Image(texture);
```

Almost all your memory usage will come from textures; you will quickly run out of RAM if you
waste texture memory.
2.3.4. Texture Atlases

In all the previous samples, we loaded each texture separately. However, real applications should actually not do that. Here's why.

• For efficient GPU rendering, Starling batches the rendered *Meshes* together. Batch processing is disrupted, however, whenever the texture changes.

• In some situations, *Stage3D* requires textures to have a width and height that are powers of two. Starling hides this limitation from you, but you will nevertheless use more memory if you do not follow that rule.

By using a texture atlas, you avoid both the texture switches and the power-of-two limitation. All textures are within one big "super-texture", and Starling takes care that the correct part of this texture is displayed.

![Figure 10. A texture atlas.](image)

The trick is to have *Stage3D* use this big texture instead of the small ones, and to map only a part of it to each quad that is rendered. This will lead to a very efficient memory usage, wasting as little space as possible. (Some other frameworks call this feature *Sprite Sheets.*)

**Creating the Atlas**

The positions of each *SubTexture* are defined in an XML file like this one:

```xml
<TextureAtlas imagePath="atlas.png">
  <SubTexture name="moon" x="0" y="0" width="30" height="30"/>
  <SubTexture name="jupiter" x="30" y="0" width="65" height="78"/>
  ...
</TextureAtlas>
```

As you can see, the XML references one big texture and defines multiple named *SubTextures*, each
pointing to an area within that texture. At runtime, you can reference these *SubTextures* by their name and they will act just as if they were independent textures.

But how do you combine all your textures into such an atlas? Thankfully, you don’t have to do that manually; there are lots of tools around that will help you with that task. Here are two candidates, but *Google* will bring up many more.

- **TexturePacker** is my personal favorite. You won’t find any tool that allows for so much control over your sprite sheets, and its Starling support is excellent (ATF textures, anyone?).
- **Shoebox** is a free tool built with *AIR*. While it doesn’t have as many options for atlas creation as *TexturePacker*, it contains lots of related functionality, like bitmap font creation or sprite extraction.

### Using the Atlas

Okay: you’ve got a texture atlas now. But how do you use it? Let’s start with embedding the texture and XML data.

```javascript
[Embed(source="atlas.xml", mimeType="application/octet-stream")] ①
public static const AtlasXml:Class;

[Embed(source="atlas.png")] ②
public static const AtlasTexture:Class;
```

① Embed the atlas XML. Don’t forget to specify the `mimeType`.

② Embed the atlas texture.

Alternatively, you can also load these files from an URL or from the disk (if we are talking about *AIR*). We will look at that in detail when we discuss Starling’s `AssetManager`.

With those two objects available, we can create a new `TextureAtlas` instance and access all `SubTextures` through the method `getTexture()`. Create the atlas object once when the game is initialized and reference it throughout its lifetime.

```javascript
var texture:Texture = Texture.fromEmbeddedAsset(AtlasTexture); ①
define texture:Texture = Texture.fromEmbeddedAsset(AtlasTexture); ①
var xml:XML = XML(new AtlasXml());
var atlas:TextureAtlas = new TextureAtlas(texture, xml);

var moonTexture:Texture = atlas.getTexture("moon"); ②
var moonTexture:Texture = atlas.getTexture("moon"); ②
define moonTexture:Texture = atlas.getTexture("moon"); ②
define moonTexture:Texture = atlas.getTexture("moon"); ②
define moonImage:Image = new Image(moonTexture);
```

① Create the atlas.

② Display a *SubTexture*.

It’s as simple as that!
2.3.5. Render Textures

The RenderTexture class allows creating textures dynamically. Think of it as a canvas on which you can paint any display object.

After creating a render texture, just call the drawObject method to render an object directly onto the texture. The object will be drawn onto the texture at its current position, adhering its current rotation, scale and alpha properties.

```javascript
var renderTexture:RenderTexture = new RenderTexture(512, 512); ①
var brush:Sprite = getBrush(); ②
brush.x = 40;
brush.y = 120;
brush.rotation = 1.41;
renderTexture.draw(brush); ③
```

① Create a new RenderTexture with the given size (in points). It will be initialized with fully transparent pixels.

② In this sample, we're referencing a display object depicting a brush. We move it to a certain location.

③ The brush object will be drawn to the texture with its current position and orientation settings.

Drawing is done very efficiently, as it is happening directly in graphics memory. After you have drawn objects onto the texture, the performance will be just like that of a normal texture — no matter how many objects you have drawn.

```javascript
var image:Image = new Image(renderTexture);
addChild(image); ①
```

① The texture can be used like any other texture.

If you draw lots of objects at once, it is recommended to bundle the drawing calls in a block via the drawBundled method, as shown below. This allows Starling to skip a few rather costly operations, speeding up the process immensely.

```javascript
renderTexture.drawBundled(function():void ① {
    for (var i:int=0; i<numDrawings; ++i)
    {
        image.rotation = (2 * Math.PI / numDrawings) * i;
        renderTexture.draw(image); ②
    }
});
```

① Activate bundled drawing by encapsulating all draw-calls in a function.
Inside the function, call `draw` just like before.

You can use any display object like an eraser by setting its blend mode to `BlendMode.ERASE`. That way, you can selectively remove parts of the texture.

```javascript
brush.blendMode = BlendMode.ERASE;
renderTexture.draw(brush);
```

To wipe it completely clean, use the `clear` method.

---

**Context Loss**

Unfortunately, render textures have one big disadvantage: they lose all their contents when the render context is lost. Context Loss is discussed in detail in a later chapter; in a nutshell, it means that Stage3D may lose the contents of all its buffers in certain situations. (Yes, that is as nasty as it sounds.)

Thus, if it is really important that the texture's contents is persistent (i.e. it's not just eye candy), you will need to make some arrangements. We will look into possible strategies in the mentioned chapter—I just wanted to mention this fact here so it doesn't hit you by surprise.

---

### 2.4. Dynamic Text

Text is an important part of every application. You can only convey so much information with images; some things simply need to be described with words, dynamically at run-time.

#### 2.4.1. TextFields

Starling makes it easy to display dynamic text. The `TextField` class should be quite self explanatory!

```javascript
var textField:TextField = new TextField(100, 20, "text"); ①
textField.format.setTo("Arial", 12, Color.RED); ②
textField.format.horizontalAlign = Align.RIGHT; ③
textField.border = true; ④
```

① Create a TextField with a size of 100×20 points, displaying the text “text”.

② We set the format to "Arial" with a size of 12 points, in red.

③ The text is aligned to the right.

④ The border property is mainly useful during development: it will show the boundaries of the TextField.

Note that the style of the text is set up via the `format` property, which points to a `starling.text.TextFormat` instance.

Once created, you can use a `TextField` just like you’d use an image or quad.
2.4.2. TrueType Fonts

Per default, Starling will use system fonts to render text. For example, if you set up your TextField to use "Arial", it will use the one installed on the system (if it is).

However, the rendering quality of that approach is not optimal; for example, the font might be rendered without anti-aliasing.

For a better output, you should embed your TrueType fonts directly into the SWF file. Use the following code to do that:

```
[Embed(source="Arial.ttf", embedAsCFF="false", fontFamily="Arial")]
private static const Arial:Class; ①

[Embed(source="Arial Bold.ttf", embedAsCFF="false", fontFamily="Arial", fontWeight="bold")]
private static const ArialBold:Class; ②

[Embed(source="Arial Italic.ttf", embedAsCFF="false", fontFamily="Arial", fontStyle="italic")]
private static const ArialItalic:Class; ③

[Embed(source="Arial.ttf", embedAsCFF="false", fontFamily="Arial", unicodeRange = "U+0020-U+007e")]
private static const ArialJustLatin:Class; ④
```

① Embedding the standard Arial font. Note the embedAsCFF part: don’t skip it! Otherwise, the font simply won’t show up.

② Bold and italic styles must be embedded separately. Note the fontWeight attribute here,

③ and the fontStyle attribute here.

④ You can also define which glyphs to include, which is useful for big fonts when you don’t need all Unicode letters. The range shown here is for basic Latin (upper- and lowercase chars, numerals and common symbols/punctuations).
After embedding the font, any TextField that is set up with a corresponding font name (font family) and weight will use it automatically. There's nothing else to set up or configure.

Beware of the big footprint when embedding all glyphs of a font. The "unicodeRange" shown above mitigates this problem. You can generate the ranges using e.g. Unicode Range Generator.

If your text is clipped or does not appear at the correct position, have a look at your current stage.quality setting. A low quality value often causes Flash/AIR to report incorrect values regarding the text bounds, and Starling depends on those values when it draws the text. (I'm talking about the Flash stage here; and this applies only to TrueType fonts.)

2.4.3. Bitmap Fonts

Using TrueType fonts as shown above is adequate for text that does not change very often. However, if your TextField constantly changes its contents, or if you want to display a fancy font that's not available in TrueType format, you should use a bitmap font instead.

A bitmap font is a texture containing all the characters you want to display. Similar to a TextureAtlas, an XML file stores the positions of the glyphs inside the texture.

This is all Starling needs to render bitmap fonts. To create the necessary files, there are several options:

- Littera, a full-featured free online bitmap font generator.
- Bitmap Font Generator, a tool provided by AngelCode that lets you create a bitmap font out of any TrueType font. It is, however, only available for Windows.
- Glyph Designer for macOS, an excellent tool that allows you to add fancy special effects to your fonts.
- bmGlyph, also exclusive for macOS, available on the App Store.

The tools are all similar to use, allowing you to pick one of your system fonts and optionally apply some special effects. On export, there are a few things to consider:

- Starling requires the XML variant of the .fnt format.
- Make sure to pick the right set of glyphs; otherwise, your font texture will grow extremely big.

The result is a .fnt file and an associated texture containing the characters.
To make such a font available to Starling, you can embed it in the SWF and register it at the TextField class.

```actionscript
[Embed(source="font.png")] public static const FontTexture:Class;

[Embed(source="font.fnt", mimeType="application/octet-stream")]
public static const FontXml:Class;

var texture:Texture = Texture.fromEmbeddedAsset(FontTexture);
var xml:XML = XML(new FontXml());
var font:BitmapFont = new BitmapFont(texture, xml); ①

TextField.registerCompositor(font, font.name); ②
```

① Create an instance of the BitmapFont class.

② Register the font at the TextField class.

Once the bitmap font instance has been registered at the TextField class, you don't need it any longer. Starling will simply pick up that font when it encounters a TextField that uses a font with that name. Like here:

```actionscript
var textField:TextField = new TextField(100, 20, "Hello World");
textField.format.font = "fontName"; ①
textField.format.size = BitmapFont.NATIVE_SIZE; ②
```

① To use the font, simply reference it by its name. By default, that's what is stored in the face attribute within the XML file.

② Bitmap fonts look best when they are displayed in the exact size that was used to create the font texture. You could assign that size manually—but it's smarter to let Starling do that, via the NATIVE_SIZE constant.
Gotchas

There's one more thing you need to know: if your bitmap font uses just a single color (like a normal TrueType font, without any color effects), your glyphs need to be exported in pure white. The `format.color` property of the TextField can then be used to tint the font into an arbitrary color at runtime (simply by multiplication with the RGB channels of the texture).

On the other hand, if your font does contains colors (like the sample image above), it's the TextField's `format.color` property that needs to be set to white (`Color.WHITE`). That way, the color tinting of the TextField will not affect the texture color.

For optimal performance, you can even add bitmap fonts to your texture atlas! That way, your texts may be batched together with regular images, reducing draw calls even more.

To do that, simply add the font's PNG image to your atlas, just like the other textures. Then initialize the bitmap font with the `SubTexture` from the atlas and the regular `.fnt` file.

When you support multiple scale factors (a concept we will look at in the Mobile Development chapter), the process becomes a little difficult, though. You cannot simply create one high resolution font and have the atlas generator scale it down; this would result in occasional graphical glitches. Each scaled font must be created separately by the bitmap font creator.

The MINI Font

Starling actually comes with one very lightweight bitmap font included. It probably won't win any beauty contests—but it's perfect when you need to display text in a prototype, or maybe for some debug output.

The quick brown fox jumps over the lazy dog.

*Figure 13. The "MINI" bitmap font.*

When I say lightweight, I mean it: each letter is only 5 pixels high. There is a trick, though, that will scale it up to exactly 200% its native size.

```javascript
var textField:TextField = new TextField(100, 10, "The quick brown fox ...");
textField.format.font = BitmapFont.MINI; ①
textField.format.size = BitmapFont.NATIVE_SIZE * 2; ②
```

① Use the MINI font.
② Use exactly twice the native size. Since the font uses nearest neighbor scaling, it will stay crisp!

2.5. Event Handling

You can think of events as occurrences of any kind that are of interest to you as a programmer.
• For example, a mobile app might notify you that the device orientation has changed, or that the user just touched the screen.

• On a lower level, a button might indicate that it was triggered, or a knight that he has run out of health points.

That's what Starling's event mechanism is for.

### 2.5.1. Motivation

The event mechanism is a key feature of Starling’s architecture. In a nutshell, events allow objects to communicate with each other.

You might think: we already have a mechanism for that — methods! That’s true, but methods only work in one direction. For example, look at a `MessageBox` that contains a `Button`.

![MessageBox with Button](image)

The message box owns the button, so it can use its methods and properties, e.g.

```java
public class MessageBox extends DisplayObjectContainer {
    private var _yesButton:Button;

    private function disableButton():void {
        _yesButton.enabled = false; ①
    }
}
```

① Communicate with the `Button` via a property.

The `Button` instance, on the other hand, does not own a reference to the message box. After all, a button can be used by any component — it's totally independent of the `MessageBox` class. That's a good thing, because otherwise, you could only use buttons inside message boxes, and nowhere else. Ugh!

Still: the button is there for a reason — if triggered, it needs to tell somebody about it! In other words: the button needs to be able to send messages to its owner, whoever that is.
2.5.2. Event & EventDispatcher

I have something to confess: when I showed you the class hierarchy of Starling's display objects, I omitted the actual base class: EventDispatcher.

This class equips all display objects with the means to dispatch and handle events. It's not a coincidence that all display objects inherit from EventDispatcher; in Starling, the event system is tightly integrated with the display list. This has some advantages we will see later.

Events are best described by looking at an example.

Imagine for a moment that you've got a dog; let's call him Einstein. Several times each day, Einstein will indicate to you that he wants to go out for a walk. He does so by barking.
class Dog extends Sprite {
    function advanceTime():void {
        if (timeToPee) {
            var event:Event = new Event("bark"); ①
dispatchEvent(event); ②
        }
    }
}

var einstein:Dog = new Dog();
einstein.addEventListener("bark", onBark); ③

function onBark(event:Event):void ④ {
einstein.walk();
}

① The string bark will identify the event. It's encapsulated in an Event instance.
② Dispatching event will send it to everyone who subscribed to bark events.
③ Here, we do subscribe by calling addEventListener. The first argument is the event type, the second the listener (a function).
④ When the dog barks, this method will be called with the event as parameter.

You just saw the three main components of the event mechanism:

• Events are encapsulated in instances of the Event class (or subclasses thereof).
• To dispatch an event, the sender calls dispatchEvent, passing the Event instance along.
• To listen to an event, the client calls addEventListener, indicating which type of event he is interested in and the function or method to be called.

From time to time, your aunt takes care of the dog. When that happens, you don’t mind if the dog barks — your aunt knows what she signed up for! So you remove the event listener, which is a good practice not only for dog owners, but also for Starling developers.

einstein.removeListener("bark", onBark); ①
einstein.removeListener("bark"); ②

① This removes the specific onBark listener.
② This removes all listeners of that type.

So much for the bark event. Of course, Einstein could dispatch several different event types, for example howl or growl events. It's recommended to store such strings in static constants, e.g. right in the Dog class.
class Dog extends Sprite {
    public static const BARK:String = "bark";
    public static const HOWL:String = "howl";
    public static const GROWL:String = "growl";
}
einstein.addEventListener(Dog.GROWL, burglar.escape);
einstein.addEventListener(Dog.HOWL, neighbor.complain);

Starling predefines several very useful event types right in the Event class. Here’s a selection of the most popular ones:

- **Event.TRIGGERED**: a button was triggered
- **Event.ADDED**: a display object was added to a container
- **Event.ADDED_TO_STAGE**: a display object was added to a container that is connected to the stage
- **Event.REMOVED**: a display object was removed from a container
- **Event.REMOVED_FROM_STAGE**: a display object lost its connection to the stage
- **Event.ENTER_FRAME**: some time has passed, a new frame is rendered (we’ll get to that later)
- **Event.COMPLETE**: something (like a MovieClip instance) just finished

2.5.3. Custom Events

Dogs bark for different reasons, right? Einstein might indicate that he wants to pee, or that he is hungry. It might also be a way to tell a cat that it’s high time to make an exit.

Dog people will probably hear the difference (I’m a cat person; I won’t). That’s because smart dogs set up a *BarkEvent* that stores their intent.

```
public class BarkEvent extends Event {
    public static const BARK:String = "bark"; ①

    private var _reason:String; ②

    public function BarkEvent(type:String, reason:String, bubbles:Boolean=false) {
        super(type, bubbles); ③
        _reason = reason;
    }

    public function get reason():Boolean { return _reason; } ④
}
```

① It’s a good practice to store the event type right at the custom event class.
The reason for creating a custom event: we want to store some information with it. Here, that’s the `reason` String.

Call the super class in the constructor. (We will look at the meaning of `bubbles` shortly.)

Make `reason` accessible via a property.

The dog can now use this custom event when barking:

```javascript
class Dog extends Sprite {
    function advanceTime():void {
        var reason:String = this.hungry ? "hungry" : "pee";
        var event:BarkEvent = new BarkEvent(BarkEvent.BARK, reason);
        dispatchEvent(event);
    }
}

var einstein:Dog = new Dog();
einstein.addEventListener(BarkEvent.BARK, onBark);

function onBark(event:BarkEvent):void {
    if (event.reason == "hungry")
        einstein.feed();
    else
        einstein.walk();
}
```

Note that the parameter is of type `BarkEvent`.

That’s why we can now access the `reason` property and act accordingly.

That way, any dog owners familiar with the `BarkEvent` will finally be able to truly understand their dog. Quite an accomplishment!

### 2.5.4. Simplifying

Agreed: it’s a little cumbersome to create that extra class just to be able to pass on that `reason` string. After all, it’s very often just a single piece of information we are interested in. Having to create additional classes for such a simple mechanism feels somewhat inefficient.

That’s why you won’t actually need the subclass-approach very often. Instead, you can make use of the `data` property of the `Event` class, which can store arbitrary references (its type: `Object`).

Replace the `BarkEvent` logic with this:
// create & dispatch event
var event:Event = new Event(Dog.BARK);
event.data = "hungry"; ①
dispatchEvent(event);

// listen to event
einstein.addEventListener(Dog.BARK, onBark);
function onBark(event:Event):void
{
    trace("reason: " + event.data as String); ②
}

① Store the reason for barking inside the data property.
② To get the reason back, cast data to String.

The downside of this approach is that we lose some type-safety. But in my opinion, I’d rather have that cast to String than implement a complete class.

Furthermore, Starling has a few shortcuts that simplify this code further! Look at this:

// create & dispatch event
dispatchEventWith(Dog.BARK, false, "hungry"); ①

// listen to event
einstein.addEventListener(Dog.BARK, onBark);
function onBark(event:Event, reason:String):void
{
    trace("reason: " + reason); ②
}

① Creates an event of type Dog.BARK, populates the data property, and dispatches the event — all in one line.
② The data property is passed to the (optional) second argument of the event handler.

We got rid of quite an amount of boiler plate code that way! Of course, you can use the same mechanism even if you don’t need any custom data. Let’s look at the most simple event interaction possible:

// create & dispatch event
dispatchEventWith(Dog.HOWL); ①

// listen to event
dog.addEventListener(Dog.HOWL, onHowl);
function onHowl():void ②
{
    trace("hoooh!");
}
① Dispatch an event by only specifying its type.

② Note that this function doesn’t contain any parameters! If you don’t need them, there’s no need to specify them.

The simplified `dispatchEventWith` call is actually even more memory efficient, since Starling will pool the `Event` objects behind the scenes.

### 2.5.5. Bubbling

In our previous examples, the event dispatcher and the event listener were directly connected via the `addEventListener` method. But sometimes, that’s not what you want.

Let’s say you created a complex game with a deep display list. Somewhere in the branches of this list, *Einstein* (the protagonist-dog of this game) ran into a trap. He howls in pain, and in his final breaths, dispatches a `GAME_OVER` event.

Unfortunately, this information is needed far up the display list, in the game’s root class. On such an event, it typically resets the level and returns the dog to its last save point. It would be really cumbersome to hand this event up from the dog over numerous display objects until it reaches the game root.

That’s a very common requirement — and the reason why events support something that is called *bubbling*.

Imagine a real tree (it’s your display list) and turn it around by 180 degrees, so that the trunk points upwards. The trunk, that’s your stage, and the leaves of the tree are your display objects. Now, if a leaf creates a bubbling event, that event will move upwards just like the bubbles in a glass of soda, traveling from branch to branch (from parent to parent) until it finally reaches the trunk.

![Figure 14. An event bubbles all the way up to the stage.](image)

Any display object along this route can listen to this event. It can even pop the bubble and stop it from traveling further. All that is required to do that is to set the `bubbles` property of an event to true.
// classic approach:
var event:Event = new Event("gameOver", true); ①
dispatchEvent(event);

// one-line alternative:
dispatchEventWith("gameOver", true); ②

① Passing true as second parameter of the Event constructor activates bubbling.
② Alternatively, dispatchEventWith takes the exact same parameters.

Anywhere along its path, you can listen to this event, e.g. on the dog, its parent, or the stage:

dog.addEventListener("gameOver", onGameOver);
dog.parent.addEventListener("gameOver", onGameOver);
stage.addEventListener("gameOver", onGameOver);

This feature comes in handy in numerous situations; especially when it comes to user input via mouse or touch screen.

### 2.5.6. Touch Events

While typical desktop computers are controlled with a mouse, most mobile devices, like smartphones or tablets, are controlled with your fingers.

Starling unifies those input methods and treats all "pointing-device" input as TouchEvent. That way, you don’t have to care about the actual input method your game is controlled with. Whether the input device is a mouse, a stylus, or a finger: Starling will always dispatch touch events.

First things first: if you want to support multitouch, make sure to enable it before you create your Starling instance.

```javascript
Starling.multitouchEnabled = true;

var starling:Starling = new Starling(Game, stage);
starling.simulateMultitouch = true;
```

Note the property simulateMultitouch. If you enable it, you can simulate multitouch input with your mouse on your development computer. Press and hold the Ctrl or Cmd keys (Windows or Mac) when you move the mouse cursor around to try it out. Add Shift to change the way the alternative cursor is moving.
To react to touch events (real or simulated), you need to listen for events of the type `TouchEvent.TOUCH`.

```plaintext
sprite.addEventListener(TouchEvent.TOUCH, onTouch);
```

You might have noticed that I’ve just added the event listener to a `Sprite` instance. `Sprite`, however, is a container class; it doesn’t have any tangible surface itself. Is it even possible to touch it, then?

Yes, it is — thanks to bubbling.

To understand that, think back to the `MessageBox` class we created a while ago. When the user clicks on its text field, anybody listening to touches on the text field must be notified — so far, so obvious. But the same is true for somebody listening for touch events on the message box itself; the text field is part of the message box, after all. Even if somebody listens to touch events on the stage, he should be notified. Touching any object in the display list means touching the stage!

Thanks to bubbling events, Starling can easily represent this type of interaction. When it detects a touch on the screen, it figures out which leaf object was touched. It creates a `TouchEvent` and dispatches it on that object. From there, it will bubble up along the display list.

**Touch Phases**

Time to look at an actual event listener:

```plaintext
private function onTouch(event:TouchEvent):void
{
    var touch:Touch = event.getTouch(this, TouchPhase.BEGAN);
    if (touch)
    {
        var localPos:Point = touch.getLocation(this);
        trace("Touched object at position: " + localPos);
    }
}
```

That’s the most basic case: Find out if somebody touched the screen and trace out the coordinates. The method `getTouch` is provided by the `TouchEvent` class and helps you find the touches you are interested in.
The `Touch` class encapsulates all information of a single touch: where it occurred, where it was in the previous frame, etc.

As first parameter, we passed `this` to the `getTouch` method. Thus, we're asking the event to return any touches that occurred on `this` or its children.

Touches go through a number of phases within their lifetime:

- **TouchPhase.HOVER**
  - Only for mouse input; dispatched when the cursor moves over the object with the mouse button up.
- **TouchPhase.BEGAN**
  - The finger just hit the screen, or the mouse button was pressed.
- **TouchPhase.MOVED**
  - The finger moves around on the screen, or the mouse is moved while the button is pressed.
- **TouchPhase.STATIONARY**
  - The finger or mouse (with pressed button) has not moved since the last frame.
- **TouchPhase.ENDED**
  - The finger was lifted from the screen or from the mouse button.

Thus, the sample above (which looked for phase `BEGAN`) will write trace output at the exact moment the finger touches the screen, but not while it moves around or leaves the screen.

### Multitouch

In the sample above, we only listened to single touches (i.e. one finger only). Multitouch is handled very similarly; the only difference is that you call `touchEvent.getTouches` instead (note the plural).

```javascript
var touches:Vector.<Touch> = event.getTouches(this, TouchPhase.MOVED);

if (touches.length == 1)
{
    // one finger touching (or mouse input)
    var touch:Touch = touches[0];
    var movement:Point = touch.getMovement(this);
}
else if (touches.length >= 2)
{
    // two or more fingers touching
    var touch1:Touch = touches[0];
    var touch2:Touch = touches[1];
    // ...
}
```

The `getTouches` method returns a vector of touches. We can base our logic on the length and contents of that vector.
• In the first if-clause, only a single finger is on the screen. Via `getMovement`, we could e.g. implement a drag-gesture.

• In the else-clause, two fingers are on the screen. By accessing both touch objects, we could e.g. implement a pinch-gesture.

The demo application that's part of the Starling download contains the `TouchSheet` class, which is used in the `Multitouch` scene. It shows a sample implementation of a touch handler that allows dragging, rotation and scaling a sprite.

**Mouse Out and End Hover**

There's a special case to consider when you want to detect that a mouse was moved away from an object (with the mouse button in "up"-state). (This is only relevant for mouse input.)

If the target of a hovering touch changed, a `TouchEvent` is dispatched to the previous target to notify it that it's no longer being hovered over. In this case, the `getTouch` method will return `null`. Use that knowledge to catch what could be called a `mouse out` event.

```javascript
var touch:Touch = event.getTouch(this);
if (touch == null)
    resetButton();
```

### 2.6. Animations

Animations are not only a fundamental part of any game; even modern business apps are expected to provide smooth and dynamic transitions. Some well placed animations go a long way towards providing a responsive and intuitive interface. To help with that, Starling offers a very flexible animation engine.

If you think about it, there are two types of animations.

• On the one hand, you've got animations that are so dynamic that you don't know beforehand what exactly will happen. Think of an enemy that's moving towards the player: its direction and speed need to be updated each frame, depending on the environment. Or physics: each additional force or collision changes everything.

• Then there are animations that follow a meticulous plan; you know from the beginning exactly what will happen. Think of fading in a message box or transitioning from one screen to another.

We will look at both of these types in the following sections.

#### 2.6.1. EnterFrameEvent

In some game engines, you have what is called a `run-loop`. That's an endless loop which constantly updates all elements of the scene.

In Starling, due to the display list architecture, such a run loop would not make much sense. You
separated your game into numerous different custom display objects, and each should know for itself what to do when some time has passed.

That's exactly the point of the `EnterFrameEvent`: allowing a display object to update itself over time. Every frame, that event is dispatched to all display objects that are part of the display list. Here is how you use it:

```javascript
public function CustomObject()
{
    addEventListener(EnterFrameEvent.ENTER_FRAME, onEnterFrame); ①
}

private function onEnterFrame(event:EnterFrameEvent):void ②
{
    trace("Time passed since last frame: " + event.passedTime);
    bird.advanceTime(passedTime);
}
```

① You can add a listener to this event anywhere, but the constructor is a good candidate.

② That's what the corresponding event listener looks like.

The method `onEnterFrame` is called once per frame, and it's passed along the exact time that has elapsed since the previous frame. With that information, you can move your enemies, update the height of the sun, or do whatever else is needed.

The power behind this event is that you can do completely different things each time it occurs. You can dynamically react to the current state of the game.

For example, you could let an enemy take one step towards the player; a simple form of enemy AI, if you will!
Simplified Version

Actually, I tend to use the following, slightly condensed code instead:

```javascript
public function CustomObject() {
    addEventListener(Event.ENTER_FRAME, onEnterFrame); ①
}

private function onEnterFrame(event:Event, passedTime:Number):void ② {
    trace("Time passed since last frame: " + event.passedTime);
}
```

① Since the **ENTER_FRAME** event is used so often, Starling defines the type constant on the **Event** class, too.

② You can tell Starling to send **passedTime** as the second parameter of the event handler. Since you don't need any other properties of the **EnterFrameEvent**, there is no need to fully qualify the class.

This is exactly equivalent to the original code—the sole advantage being that you save a couple of key strokes. For lazy developers like me, that's all the reason I need to prefer this variant!

### 2.6.2. Tweens

Now to predefined animations. They are very common and have names such as **movement**, **scale**, **fade**, etc. Starling's approach on these kinds of animations is simple—but at the same time very flexible. Basically, you can animate any property of any object, as long as it is numeric (**Number**, **int**, **uint**). Those animations are described in an object called **Tween**.

The term "Tween" comes from hand drawn animations, where a lead illustrator would draw important key frames, while the rest of the team drew the frames in-between those frames.

Figure 16. The different frames of a tween.

Enough theory, let's go for an example:
This tween describes an animation that moves the ball object to \( x = 20 \), scales it to twice its size and reduces its opacity until it is invisible. All those changes will be carried out simultaneously over the course of half a second. The start values are simply the current values of the specified properties.

This sample showed us that

- you can animate arbitrary properties of an object, and that
- you can combine multiple animations in one tween object.

Apropos: since scaling, fading and movement are done so frequently, the Tween class provides specific methods for that, too. So you can write the following instead:

```javascript
var tween:Tween = new Tween(ball, 0.5);

tween.animate("x", 20);
tween.animate("scale", 2.0);
tween.animate("alpha", 0.0);
```

An interesting aspect of tweens is that you can change the way the animation is executed, e.g. letting it start slow and get faster over time. That's done by specifying a transition type.
Figure 17. The available transition types. The default, `linear`, was omitted.

The following example shows how to specify such a transition and introduces a few more tricks the class is capable of.

```javascript
var tween:Tween = new Tween(ball, 0.5, Transitions.EASE_IN); ①
tween.onStart = function():void { /* ... */ }; ②
tween onUpdate = function():void { /* ... */ }; ②
tween.onComplete = function():void { /* ... */ }; ②
tween.delay = 2; ③
tween.repeatCount = 3; ④
tween.reverse = true;
tween.nextTween = explode; ⑤
```
Specify the transition via the third constructor argument.

These callbacks are executed when the tween has started, each frame, or when it has finished, respectively.

Wait two seconds before starting the animation.

Repeat the tween three times, optionally in yoyo-style (reverse). If you set repeatCount to zero, the tween will be repeated indefinitely.

Specify another tween to start right after this one is complete.

We just created and configured a tween — but nothing is happening yet. A tween object describes the animation, but it does not execute it.

You could do that manually via the tweens advanceTime method:

```javascript
ball.x = 0;
tween = new Tween(ball, 1.0);
tween.animate("x", 100);

tween.advanceTime(0.25); // -> ball.x = 25
tween.advanceTime(0.25); // -> ball.x = 50
tween.advanceTime(0.25); // -> ball.x = 75
tween.advanceTime(0.25); // -> ball.x = 100
```

Hm, that works, but it's a little cumbersome, isn't it? Granted, one could call advanceTime in an ENTER_FRAME event handler, but still: as soon as you've got more than one animation, it's bound to become tedious.

Don't worry: I know just the guy for you. He's really good at handling such things.

### 2.6.3. Juggler

The juggler accepts and executes any number of animatable objects. Like any true artist, it will tenaciously pursue its true passion, which is: continuously calling advanceTime on everything you throw at it.

There is always a default juggler available on the active Starling instance. The easiest way to execute an animation is through the line below — just add the animation (tween) to the default juggler and you are done.

```javascript
Starling.juggler.add(tween);
```

When the tween has finished, it will be thrown away automatically. In many cases, that simple approach will be all you need!

In other cases, though, you need a little more control. Let's say your stage contains a game area where the main action takes place. When the user clicks on the pause button, you want to pause the game and show an animated message box, maybe providing an option to return to the menu.
When that happens, the game should freeze completely: none of its animations should be advanced any longer. The problem: the message box itself uses some animations too, so we can’t just stop the default juggler.

In such a case, it makes sense to give the game area its own juggler. As soon as the exit button is pressed, this juggler should just stop animating anything. The game will freeze in its current state, while the message box (which uses the default juggler, or maybe another one) animates just fine.

When you create a custom juggler, all you have to do is call its `advanceTime` method in every frame. I recommend using jugglers the following way:

```javascript
public class Game {  
    private var _gameArea:GameArea;

    private function onEnterFrame(event:Event, passedTime:Number):void {
        if (activeMsgBox) 
            trace("waiting for user input");
        else 
            _gameArea.advanceTime(passedTime);  
    }
}

public class GameArea {
    private var _juggler:Juggler;

    public function advanceTime(passedTime:Number):void {
        _juggler.advanceTime(passedTime);  
    }
}
```

① In your Game’s root class, listen to `Event.ENTER_FRAME`.
② Advance the `gameArea` only when there is no active message box.
③ The `GameArea` contains its own juggler. It will manage all in-game animations.
④ The juggler is advanced in its `advanceTime` method (called by `Game`).

That way, you have neatly separated the animations of the game and the message box.

By the way: the juggler is not restricted to Tweens. As soon as a class implements the `IAnimatable` interface, you can add it to the juggler. That interface has only one method:

```javascript
function advanceTime(time:Number):void;
```

By implementing this method, you could e.g. create a simple MovieClip-class yourself. In its
The `advanceTime` method, it would constantly change the texture that is displayed. To start the movie clip, you’d simply add it to a juggler.

This also opens up another strategy for handling custom jugglers. Since the `Juggler` class implements `IAnimatable` as well, jugglers can be added to other jugglers!

```javascript
Starling.juggler.add(_juggler);
```

That way, you don’t have to set up any `ENTER_FRAME` event listeners; just add your custom juggler to the default juggler. When you want to pause this group of animations, simply remove the juggler again.

This leaves one question, though: when and how is an object removed from the juggler?

**Stopping Animations**

When a tween finishes, it is removed from the juggler automatically. If you want to abort the animation before it is finished, you simply remove it from the juggler.

Let’s say you just created a tween that animates a ball and added it to the default juggler:

```javascript
tween:Tween = new Tween(ball, 1.5);
tween.moveTo(x, y);
Starling.juggler.add(tween);
```

There are several ways you can abort that animation. Depending on the circumstances, simply pick the one that suits your game logic best.

```javascript
var animID:uint = Starling.juggler.add(tween);

Starling.juggler.remove(tween); ①  
Starling.juggler.removeTweens(ball); ②  
Starling.juggler.removeByID(animID); ③  
Starling.juggler.purge(); ④
```

① Remove the tween directly. This works with any `IAnimatable` object.  
② Remove all tweens that affect the ball. Only works for tweens!  
③ Remove the tween by its ID. Useful when you don’t have access to the `Tween` instance.  
④ If you want to abort everything, purge the juggler.

Be a little careful with the `purge` method, though: if you call it on the default juggler, another part of your code might suddenly be faced with an aborted animation, bringing the game to a halt. I recommend you use `purge` only on your custom jugglers.
Automatic Removal

You might have asked yourself how the Tween class manages to have tweens removed from the juggler automatically once they are completed. That's done with the REMOVE_FROM_JUGGLER event.

Any object that implements IAnimatable can dispatch such an event; the juggler listens to those events and will remove the object accordingly.

```java
public class MyAnimation extends EventDispatcher implements IAnimatable {
    public function stop():void {
        dispatchEventWith(Event.REMOVE_FROM_JUGGLER);
    }
}
```

Single-Command Tweens

While the separation between tween and juggler is very powerful, it sometimes just stands in the way, forcing you to write a lot of code for simple tasks. That's why there is a convenience method on the juggler that allows you to create and execute a tween with a single command. Here's a sample:

```javascript
juggler.tween(msgBox, 0.5, {
    transition: Transitions.EASE_IN,
    onComplete: function():void { button.enabled = true; },
    x: 300,
    rotation: deg2rad(90)
});
```

This will create a tween for the msgBox object with a duration of 0.5 seconds, animating both the x and rotation properties. As you can see, the {} parameter is used to list all the properties you want to animate, as well as the properties of the Tween itself. A huge time-saver!

2.6.4. Delayed Calls

Technically, we have now covered all the animation types Starling supports. However, there's actually another concept that's deeply connected to this topic.

Remember Einstein, our dog-hero who introduced us to the event system? The last time we saw him, he had just lost all his health points and was about to call gameOver. But wait: don't call that method immediately — that would end the game too abruptly. Instead, call it with a delay of, say, two seconds (time enough for the player to realize the drama that is unfolding).

To implement that delay, you could use a native Timer or the setTimeout method. However, you can also use the juggler, and that has a huge advantage: you remain in full control.

It becomes obvious when you imagine that the player hits the "Pause" button right now, before
those two seconds have passed. In that case, you not only want to stop the game area from animating; you want this delayed `gameOver` call to be delayed even more.

To do that, make a call like the following:

```javascript
juggler.delayCall(gameOver, 2);
```

The `gameOver` function will be called two seconds from now (or longer if the juggler is disrupted). It's also possible to pass some arguments to that method. Want to dispatch an event instead?

```javascript
juggler.delayCall(dispatchEventWith, 2, "gameOver");
```

Another handy way to use delayed calls is to perform periodic actions. Imagine you want to spawn a new enemy once every three seconds.

```javascript
juggler.repeatCall(spawnEnemy, 3);
```

Behind the scenes, both `delayCall` and `repeatCall` create an object of type `DelayedCall`. Just like the `juggler.tween` method is a shortcut for using tweens, those methods are shortcuts for creating delayed calls.

To abort a delayed call, use one of the following methods:

```javascript
var animID:uint = juggler.delayCall(gameOver, 2);

juggler.removeByID(animID);
juggler.removeDelayedCalls(gameOver);
```

### 2.6.5. Movie Clips

You might have noticed the `MovieClip` class already when we looked at the class diagram surrounding `Mesh`. That's right: a `MovieClip` is actually just a subclass of `Image` that changes its texture over time. Think of it as Starling's equivalent of an animated GIF!

**Acquiring Textures**

It is recommended that all frames of your movie clip are from one texture atlas, and that all of them have the same size (if they have not, they will be stretched to the size of the first frame). You can use tools like Adobe Animate to create such an animation; it can export directly to Starling’s texture atlas format.

This is a sample of a texture atlas that contains the frames of a movie clip. First, look at the XML with the frame coordinates. Note that each frame starts with the prefix `flight_`. 
Here is the corresponding texture:

![Figure 18. The frames of our MovieClip.](image)

Creating the MovieClip

Now let's create the MovieClip. Supposing that the `atlas` variable points to a `TextureAtlas` containing all our frames, that's really easy.

```javascript
var frames:Vector.<Texture> = atlas.getTextures("flight_"); ①
var movie:MovieClip = new MovieClip(frames, 10); ②
addChild(movie);

movie.play();
movie.pause(); ③
didStop();

Starling.juggler.addButton(movie); ④
```

① The `getTextures` method returns all textures starting with a given prefix, sorted alphabetically.
② That's ideal for our `MovieClip`, because we can pass those textures right to its constructor. The second parameter depicts how many frames will be played back per second.
③ Those are the methods controlling playback of the clip. It will be in "play" mode per default.
④ Important: just like any other animation in Starling, the movie clip needs to be added to the juggler!

Did you notice how we referenced the textures from the atlas by their prefix `flight_`? That allows you to create a mixed atlas that contains other movie clips and textures, as well. To group the frames of one clip together, you simply use the same prefix for all of them.

The class also supports executing a sound or an arbitrary callback whenever a certain frame is reached. Be sure to check out its API reference to see what's possible!

More Complex Movies

A downside of this animation technique has to be mentioned, though: you will quickly run out of
texture memory if your animations are either very long or if the individual frames are very big. If your animations take up several big texture atlases, they might not fit into memory.

For these kinds of animations, you need to switch to a more elaborate solution: skeletal animation. This means that a character is split up into different parts (bones); those parts are then animated separately (according to the character’s skeleton). This is extremely flexible.

Support for such animations isn’t part of Starling itself, but there are several other tools and libraries coming to the rescue. All of the following work really well with Starling:

- Spine
- Flump
- Dragon Bones
- Generic Animation Format

While Spine is a standalone application (built specifically with game development in mind), the others are all based on Adobe Animate (either via plug-in or by parsing SWF data).

Apropos Adobe Animate: a recent update actually added a brand-new export option. To use it, right-click on a symbol in the Library panel and choose "Generate Texture Atlas". Don’t be fooled by the misleading name: this format has nothing to do with Starling’s standard texture atlases. It’s a completely different format that efficiently describes animations.

Best of all, loading these animations in Starling is really easy if you add this extension to your project. Once the data is loaded (which is taken care of by a custom AssetManager), you can instantiate Animation objects that work just like the MovieClip class we just encountered. Creative Cloud users should definitely give this a try!

![Image of animated ninja characters]

*Figure 19. The Adobe Animate extension is demoed with this cute sample animation by Chris Georgenes.*

### 2.7. Asset Management

One thing should be clear by now: textures make up a big part of every application’s resources. Especially when we are talking about games, there is a lot to manage; from the user interface to the characters, items, backgrounds, etc. But that’s not all: you will probably need to deal with sounds and configuration files, too.

For referencing these assets, you’ve got several choices.

- Embed them right inside the application (via the [Embed] meta data).
- Load them from disk (only possible for AIR applications).
• Load them from an URL, e.g. from a webserver.

Since every option requires different code (depending on the asset type and loading mechanism), it's difficult to access the assets in a uniform way. Thankfully, Starling contains a class that helps you with that: the AssetManager.

It supports the following types of assets:

• Textures (either from Bitmaps or ATF data)
• Texture atlases
• Bitmap Fonts
• Sounds
• XML data
• JSON data
• ByteArrays

To accomplish this, the AssetManager uses a three-step approach:

1. You add pointers to your assets to a queue, e.g. File objects or URLs.
2. You tell the AssetManager to process the queue.
3. As soon as the queue finishes processing, you can access all assets with corresponding get methods.

The AssetManager contains a verbose property. If enabled, all steps of the enqueuing and loading process will be traced to the console. That's very useful for debugging, or if you don't understand why a certain asset is not showing up! For that reason, the latest Starling versions have it enabled per default.

2.7.1. AssetManager versions

Starling 2.4 features a new, much enhanced AssetManager. All the code examples that follow were written with this new version in mind.

The new AssetManager class can be found in the starling.assets package. For compatibility reasons, the old version is still available at its previous location (in starling.utils), but it's now marked as deprecated. Upgrading to the new version should be really painless — the changes in the public API were kept at a minimum. In exchange, you can profit from the new version's much higher flexibility.

2.7.2. Enqueuing the Assets

The first step is to enqueue all the assets you want to use. How that's done exactly depends on the type and origin of each asset.
Assets from disk or from the network

Enqueuing files from disk or from a remote server is rather straight-forward:

```javascript
// Enqueue an asset from a remote URL
assets.enqueue("http://gamua.com/img/starling.jpg");

// Enqueue an asset from disk (AIR only)
var appDir:File = File.applicationDirectory;
assets.enqueue(appDir.resolvePath("sounds/music.mp3"));

// Enqueue all contents of a directory, recursively (AIR only).
assets.enqueue(appDir.resolvePath("textures"));
```

To load a texture atlas, just enqueue both its XML file and the corresponding texture. Make sure that the `imagePath` attribute in the XML file contains the correct filename, because that's what the AssetManager will look for when it creates the atlas later.

```javascript
assets.enqueue(appDir.resolvePath("textures/atlas.xml"));
assets.enqueue(appDir.resolvePath("textures/atlas.png"));
```

Bitmap Fonts work just the same. In this case, you need to make sure that the `file` attribute in the XML (the `.fnt` file) is set up correctly.

```javascript
assets.enqueue(appDir.resolvePath("fonts/desyrel.fnt"));
assets.enqueue(appDir.resolvePath("fonts/desyrel.png"));
```

Assets that are embedded

For embedded assets, I recommend you put all the embed statements into one dedicated class. Declare them as `public static const` and follow these naming conventions:

- Classes for embedded images should have the exact same name as the file, without extension. This is required so that references from XMLs (atlas, bitmap font) won't break.
- Atlas and font XML files can have an arbitrary name, since they are never referenced by file name.

Here's a sample of such a class:
```java
public class EmbeddedAssets {

    /* PNG texture */
    [Embed(source = "/textures/bird.png")]
    public static const bird:Class;

    /* ATF texture */
    [Embed(source = "textures/1x/atlas.atf",
            mimeType = "application/octet-stream")]
    public static const atlas:Class;

    /* XML file */
    [Embed(source = "textures/1x/atlas.xml",
            mimeType = "application/octet-stream")]
    public static const atlas_xml:Class;

    /* MP3 sound */
    [Embed(source = "/audio/explosion.mp3")]
    public static const explosion:Class;
}
```

When you enqueue that class, the asset manager will later instantiate all the assets that are embedded within.

```
var assets:AssetManager = new AssetManager();
assets.enqueue(EmbeddedAssets); ①
```

① Enqueues bird texture, explosion sound, and a texture atlas.

**Per-Asset Configuration**

When you create a texture manually (via the `Texture.from…()` factory methods), you've got a chance to fine-tune how it is created. For example, you can decide on a texture format or scale factor.

The problem with those settings: once the texture is created, you cannot change them any more. So you need to make sure the correct settings are applied right when the texture is created. The asset manager supports this kind of configuration, too:

```
var assets:AssetManager = new AssetManager();
assets.textureOptions.format = Context3DTextureFormat.BGRA_PACKED;
assets.textureOptions.scale = 2;
assets.enqueue(EmbeddedAssets);
```

The asset manager will adhere to these settings for all the textures it creates. However, it seems that this would only allow one set of properties for all the loaded textures, right? Actually, no: you just need to enqueue them in several steps, assigning the right settings prior to each call to `enqueue`. 
assets.textureOptions.scale = 1;
assets.enqueue(appDir.resolvePath("textures/1x"));

assets.textureOptions.scale = 2;
assets.enqueue(appDir.resolvePath("textures/2x"));

This will make the textures from the 1x and 2x folders use scale factors of one and two, respectively.

### 2.7.3. Loading the Assets

Now that the assets are enqueued, you can load all of them at once. Depending on the number and size of assets you are loading, this can take a while.

The `loadQueue` method accepts up to three different parameters, with separate callbacks for completion, error handling and progress. Only the first parameter (`onComplete`) is obligatory.

```javascript
assets.loadQueue(onComplete, onError, onProgress);

function onComplete():void { trace("done!"); } ①
function onError(error:String):void { trace("error:", error); } ②
function onProgress(ratio:Number):void { trace("progress:", ratio); } ③
```

① You can always rely on `onComplete` to be called once at the very end.

② `onError` may be executed multiple times (once for every asset that can’t be loaded).

③ Use `onProgress` if you want to show some kind of progress bar or loading indicator.

That’s the one part where you need to be careful when migrating to the new AssetManager. Previously, there was just a single callback parameter in the `loadQueue` method that signaled both progress and completion. Make sure to update your call so that it uses the new separate callbacks.

With an enabled `verbose` property, you’ll see the names with which the assets can be accessed.

```javascript
[AssetManager] Adding sound 'explosion'
[AssetManager] Adding texture 'bird'
[AssetManager] Adding texture 'atlas'
[AssetManager] Adding texture atlas 'atlas'
```

### 2.7.4. Accessing the Assets

Finally: now that the queue finished processing, you can access your assets with the various `get...` methods of the AssetManager. Each asset is referenced by a name, which is the file name of the asset (without extension) or the class name of embedded objects.
var texture:Texture = assets.getTexture("bird"); ①
var textures:Vector.<Texture> = assets.getTextures("animation"); ②
var explosion:SoundChannel = assets.playSound("explosion"); ③

① This will first search named textures, then atlases.
② Same as above, but returns all (sub) textures starting with the given String.
③ Plays a sound and returns the SoundChannel that controls it.

If you enqueued a bitmap font along the way, it will already be registered and ready to use.

In my games, I typically store a reference to the asset manager at my root class, accessible through a static property. That makes it super easy to access my assets from anywhere in the game, simply by calling Game.assets.get…() (assuming the root class is called Game).

The Starling Wiki contains two extensions for the AssetManager that are worth checking out:

• Zipped Assets: Unzip compressed assets on the fly. Useful e.g. for big JSON or XML files.
• Asset Caching: Caches remote assets locally. Subsequent loads will be executed from the disk.

2.8. Fragment Filters

Up to this point, all the drawable objects we discussed were meshes with (or without) textures mapped onto them. You can move meshes around, scale them, rotate them, and maybe tint them in a different color. All in all, however, the possibilities are rather limited — the look of the game is solely defined by its textures.

At some point, you will run into the limits of this approach; perhaps you need a variant of an image in multiple colors, blurred, or with a drop shadow. If you add all of those variants into your texture atlas, you will soon run out of memory.

Fragment filters can help with that. A filter can be attached to any display object (including containers) and can completely change its appearance.

For example, let’s say you want to add a Gaussian blur to an object:

var filter:BlurFilter = new BlurFilter(); ①
object.filter = filter; ②

① Create and configure an instance of the desired filter class.
② Assign the filter to the filter property of a display object.

With a filter assigned, rendering of a display object is modified like this:
• Each frame, the target object is rendered into a texture.
• That texture is processed by a fragment shader (directly on the GPU).
• Some filters use multiple passes, i.e. the output of one shader is fed into the next.
• Finally, the output of the last shader is drawn to the back buffer.

This approach is extremely flexible, allowing to produce all kinds of different effects (as we will see shortly). Furthermore, it makes great use of the GPU's parallel processing abilities; all the expensive per-pixel logic is executed right on the graphics chip.

That said: filters break batching, and each filter step requires a separate draw call. They are not exactly cheap, both regarding memory usage and performance. So be careful and use them wisely.

2.8.1. Showcase

Out of the box, Starling comes with a few very useful filters.

BlurFilter

Applies a Gaussian blur to an object. The strength of the blur can be set for x- and y-axis separately.

• Per blur direction, the filter requires at least one render pass (draw call).
• Per strength unit, the filter requires one render pass (a strength of 1 requires one pass, a strength of 2 two passes, etc).
• Instead of raising the blur strength, it's often better to lower the filter resolution. That has a similar effect, but is much cheaper.
**ColorMatrixFilter**

Dynamically alters the color of an object. Change an object's brightness, saturation, hue, or invert it altogether.

This filter multiplies the color and alpha values of each pixel with a $4 \times 5$ matrix. That's a very flexible concept, but it's also quite cumbersome to get to the right matrix setup. For this reason, the class contains several helper methods that will set up the matrix for the effects you want to achieve (e.g. changing hue or saturation).

- You can combine multiple color transformations in just one filter instance. For example, to change both brightness and saturation, call both of the corresponding methods on the filter.
- This filter always requires exactly one pass.

**DropShadow- and GlowFilter**

These two filters draw the original object in the front and add a blurred and tinted variant behind it.

- That also makes them rather expensive, because they add an additional render pass to what's required by a pure **BlurFilter**.

**DisplacementMapFilter**

Displaces the pixels of the target object depending on the colors in a *map texture*.

- Not exactly easy to use, but very powerful!
- Reflection on water, a magnifying glass, the shock wave of an explosion — this filter can do it.
FilterChain

To combine several filters on one display object, you can chain them together via the FilterChain class. The filters will be processed in the given order; the number of draw calls per filter are simply added up.

2.8.2. Performance Tips

I mentioned it above: while the GPU processing part is very efficient, the additional draw calls make fragment filters rather expensive. However, Starling does its best to optimize filters.

• When an object does not change its position relative to the stage (or other properties like scale and color) for two successive frames, Starling recognizes this and will automatically cache the filter output. This means that the filter won't need to be processed any more; instead, it behaves just like a single image.

• On the other hand, when the object is constantly moving, the last filter pass is always rendered directly to the back buffer instead of a texture. That spares one draw call.

• If you want to keep using the filter output even though the object is moving, call filter.cache(). Again, this will make the object act just like a static image. However, for any changes of the target object to show up, you must call cache again (or uncache).

• To save memory, experiment with the resolution and textureFormat properties. This will reduce image quality, though.

2.8.3. More Filters

Would you like to know how to create your own filters? Don’t worry, we will investigate that topic a little later (see Custom Filters).

In the meantime, you can try out filters created by other Starling developers. An excellent example is the filter collection by Devon O. Wolfgang.
2.9. Meshes

Mesh is the basic building block of all tangible display objects. By "tangible", I mean a "leaf" object in the display list: an object that is not a container but is rendered directly to the back buffer. Since it is so important, I want to look at this class in a little more detail.

In a nutshell, a Mesh represents a list of triangles to be rendered via Stage3D. It was mentioned a few times already, since it's the base class of Quad and Image. As a reminder, here is the class hierarchy we are talking about:

Mesh is not an abstract class; nothing prevents you from instantiating it directly. Here's how:

```javascript
var vertexData:VertexData = new VertexData();
vertexData.setPoint(0, "position", 0, 0);
vertexData.setPoint(1, "position", 10, 0);
vertexData.setPoint(2, "position", 0, 10);

var indexData:IndexData = new IndexData();
indexData.addTriangle(0, 1, 2);

var mesh:Mesh = new Mesh(vertexData, indexData);
addChild(mesh);
```

As you can see, we first needed to instantiate two more classes: VertexData and IndexData. They represent collections of vertices and indices, respectively.

- VertexData efficiently stores the attributes of each vertex, e.g. its position and color.
- IndexData stores indices to those vertices. Every three indices will make up a triangle.

That way, the code above created the most basic drawing primitive: a triangle. We did this by defining three vertices and referencing them clockwise. After all, that's what a GPU can do best: drawing triangles — lots of them.
2.9.1. Extending Mesh

Working directly with VertexData and IndexData would be quite bothersome over time. It makes sense to encapsulate the code in a class that takes care of setting everything up.

To illustrate how to create custom meshes, we will now write a simple class called NGon. Its task: to render a regular n-sided polygon with a custom color.

We want the class to act just like a built-in display object. You instantiate it, move it to a certain position and then add it to the display list.

```javascript
var ngon:NGon = new NGon(100, 5, Color.RED);  // ①
ngon.x = 60;
ngon.y = 60;
addChild(ngon);
```

① The constructor arguments define radius, number of edges, and color.

Let’s look at how we can achieve this feat.

2.9.2. Vertex Setup

Like all other shapes, our regular polygon can be built from just a few triangles. Here’s how we could set up the triangles of a pentagon (an n-gon with n=5).

```javascript
var ngon:NGon = new NGon(100, 5, Color.RED);  // ①
ngon.x = 60;
ngon.y = 60;
addChild(ngon);
```

① The constructor arguments define radius, number of edges, and color.

Let’s look at how we can achieve this feat.
The pentagon is made up of six vertices spanning up five triangles. We give each vertex a number between 0 and 5, with 5 being in the center.

As mentioned, the vertices are stored in a `VertexData` instance. `VertexData` defines a set of named attributes for each vertex. In this sample, we need two standard attributes:

- `position` stores a two-dimensional point (x, y).
- `color` stores an RGBA color value.

The `VertexData` class defines a couple of methods referencing those attributes. That allows us to set up the vertices of our polygon.

Create a new class called `NGon` that extends `Mesh`. Then add the following instance method:

```javascript
private function createVertexData(
    radius:Number, numEdges:int, color:uint):VertexData {
    var vertexData:VertexData = new VertexData();
    vertexData.setPoint(numEdges, "position", 0.0, 0.0); ①
    vertexData.setColor(numEdges, "color", color);
    for (var i:int=0; i<numEdges; ++i) ② {
        var edge:Point = Point.polar(radius, i*2*Math.PI / numEdges);
        vertexData.setPoint(i, "position", edge.x, edge.y);
        vertexData.setColor(i, "color", color);
    }
    return vertexData;
}
```

① Set up center vertex (last index).
② Set up edge vertices.

Since our mesh has a uniform color, we assign the same color to each vertex. The positions of the edge vertices (the corners) are distributed along a circle with the given radius.

### 2.9.3. Index Setup

That’s it for the vertices. Now we need to define the triangles that make up the polygon.

`Stage3D` wants a simple list of indices, with each three successive indices referencing one triangle. It’s a good practice to reference the indices clockwise; that convention indicates that we are looking at the front side of the triangle. Our pentagon’s list would look like this:

```
5, 0, 1, 5, 1, 2, 5, 2, 3, 5, 3, 4, 5, 4, 0
```
In Starling, the IndexData class is used to set up such a list. The following method will fill an IndexData instance with the appropriate indices.

```javascript
private function createIndexData(numEdges:int):IndexData {
    var indexData:IndexData = new IndexData();
    for (var i:int=0; i<numEdges; ++i)
        indexData.addTriangle(numEdges, i, (i+1) % numEdges);
    return indexData;
}
```

### 2.9.4. NGon constructor

This is actually all we need for our NGon class! Now we just need to make use of the above methods in the constructor. All the other responsibilities of a display object (hit testing, rendering, bounds calculations, etc.) are handled by the superclass.

```javascript
public class NGon extends Mesh {
    public function NGon(
        radius:Number, numEdges:int, color:uint=0xffffff)
    {
        var vertexData:VertexData = createVertexData(radius, numEdges, color);
        var indexData:IndexData = createIndexData(numEdges);
        super(vertexData, indexData);
    }
    // ...
}
```

That's rather straight-forward, isn't it? This approach works for any shape you can think of.

When working with custom meshes, also look at the Polygon class (in the starling.geom package). It helps with converting an arbitrary, closed shape (defined by a number of vertices) into triangles. We look at it in more detail in the Masks section.

### 2.9.5. Adding a Texture

Wouldn't it be nice if we were able to map a texture onto this polygon, as well? The base class, Mesh, already defines a texture property; we're only lacking the required texture coordinates.

Through texture coordinates, you define which part of a texture gets mapped to a vertex. They are often called UV-coordinates, which is a reference to the names that are typically used for their
coordinate axes \((u\) and \(v\)). Note that the UV range is defined to be within \(0\) and \(1\), regardless of the actual texture dimensions.

![Figure 29. The texture coordinates of the polygon are in the range 0-1.](image)

With this information, we can update the \texttt{createVertexData} method accordingly.

```javascript
function createVertexData(radius: Number, numEdges: int, color: uint): VertexData
{
    var vertexData:VertexData = new VertexData(null, numEdges + 1);
    vertexData.setPoint(numEdges, "position", 0.0, 0.0);
    vertexData.setColor(numEdges, "color", color);
    vertexData.setPoint(numEdges, "texCoords", 0.5, 0.5); ①

    for (var i:int=0; i<numEdges; ++i)
    {
        var edge:Point = Point.polar(radius, i*2*Math.PI / numEdges);
        vertexData.setPoint(i, "position", edge.x, edge.y);
        vertexData.setColor(i, "color", color);

        var u:Number = (edge.x + radius) / (2 * radius); ②
        var v:Number = (edge.y + radius) / (2 * radius);
        vertexData.setPoint(i, "texCoords", u, v);
    }

    return vertexData;
}
```

① The texture coordinates of the center vertex: \(0.5, 0.5\).

② The origin of the \(n\)-gon is in the center, but the texture coordinates must be all positive. So we move the vertex coordinates to the right (by \(radius\)) and divide them by \(2 \times radius\) to end up in the range \(0\text{-}1\).

When a texture is assigned, the rendering code will automatically pick up those values.

```javascript
var ngon:NGon = new NGon(100, 5);
ngon.texture = assets.getTexture("brick-wall");
addChild(ngon);
```
2.9.6. Anti-Aliasing

If you look closely at the edges of our n-gon, you will see that the edges are quite jagged. That’s because the GPU treats a pixel either within the n-gon, or outside — there are no in-betweens. To fix that, you can enable anti-aliasing: there’s a property with that name on the Starling class.

```javascript
starling.antiAliasing = 2;
```

The value correlates to the number of subsamples Stage3D uses on rendering. Using more subsamples requires more calculations to be performed, making anti-aliasing a potentially very expensive option. Furthermore, Stage3D doesn’t support anti-aliasing on all platforms.

Full-screen anti-aliasing is available on all desktop platforms, except when using software rendering. Support for iOS was added in AIR 24; Android is currently only supported via RenderTextures.

Thus, it’s not an ideal solution. Luckily, however, the typical pixel-density of mobile devices is constantly on the rise. On modern, high end mobile phones, the pixels are now so small that aliasing is much less an issue than it was in the past.

2.9.7. Mesh Styles

You now know how to create textured meshes with arbitrary shapes. For this, you are using the standard rendering mechanics built into Starling.

However, what if you want to customize the rendering process itself? The properties and methods
of the Mesh class provide a solid foundation—but sooner or later, you will want more than that.

Coming to the rescue: Starling’s mesh styles.

Styles are a brand new addition to Starling (introduced in version 2.0) and are the recommended way to create custom, high performance rendering code. In fact, all rendering in Starling is now done via mesh styles.

- A style can be assigned to any mesh (instances of the Mesh class or its subclasses).
- Per default, the style of each mesh is an instance of the base MeshStyle class.
- The latter provides the standard rendering capabilities of Starling: drawing colored and textured triangles.

To teach your meshes new tricks, you can extend MeshStyle. This allows you to create custom shader programs for all kinds of interesting effects. For example, you could implement fast color transformations or multi-texturing.

One of the most impressive samples of a style is the Dynamic Lighting extension. With the help of a normal map (a texture encoding surface normals), it can provide realistic real-time lighting effects. Be sure to check out this extension in the Starling Wiki to see it in action!

To use a style, instantiate it and assign it to the style property of the mesh:

```javascript
var image: Image = new Image(texture);
var lightStyle: LightStyle = new LightStyle(normalTexture);
image.style = lightStyle;
```

Figure 32. The Dynamic Lighting extension in action.

Styles are extremely versatile; their possible applications are almost without limit. And since meshes with the same style can be batched together, you do not sacrifice performance in any way. In this respect, they are much more efficient than fragment filters (which serve a similar purpose).

The main downsides of styles are simply that they can only be assigned to a mesh (not, say, a sprite), and that they can only act within the actual mesh area (making things like a blur impossible).
Furthermore, it's not possible to combine several styles on one mesh.

Still: styles are a powerful tool that any Starling developer should be familiar with. Stay tuned: in the section **Custom Styles**, I will show you how to create your own mesh style from scratch, shaders and all!

If you're still a little confused about the differences between a *Mesh* and a *MeshStyle*, think of it like this: the *Mesh* is nothing more than a list of vertices, and how those vertices spawn up triangles.

A style may add additional data to each vertex and use it on rendering. The standard *MeshStyle* provides color and texture coordinates; a *MultiTextureStyle* might add an additional set of texture coordinates, etc. But a style should never modify the original shape of the object; it won't add or remove vertices or change their positions.

### 2.10. Masks

Masks can be used to cut away parts of a display object. Think of a mask as a "hole" through which you can look at the contents of another display object. That hole can have an arbitrary shape.

If you've used the "mask" property of the classic display list, you'll feel right at home with this feature. Just assign a display object to the new mask property, as shown below. Any display object can act as a mask, and it may or may not be part of the display list.

```javascript
var sprite:Sprite = createSprite();
var mask:Quad = new Quad(100, 100);
mask.x = mask.y = 50;
sprite.mask = mask; //← use the quad as a mask
```

This will yield the following result:

![Figure 33. Using a rectangular mask.](image)

The logic behind masks is simple: a pixel of a masked object will only be drawn if it is within the mask's polygons. This is crucial: the shape of the mask is defined by its polygons — not its texture! Thus, such a mask is purely binary: a pixel is either visible, or it is not.
Masks and AIR

For masks to work in an AIR application, you will need to activate the stencil buffer in the application descriptor. Add the following setting to the `initialWindow` element:

```xml
<depthAndStencil>true</depthAndStencil>
```

But don't worry, Starling will print a warning to the console if you forget to do so.

2.10.1. Canvas and Polygon

"This mask feature looks really nice", you might say, "but how the heck am I going to create those arbitrary shapes you were talking about?!" Well, I'm glad you ask!

Indeed: since masks rely purely on geometry, not on any textures, you need a way to draw your mask-shapes. In a funny coincidence, there are actually two classes that can help you with exactly this task: Canvas and Polygon. They go hand-in-hand with stencil masks.

The API of the Canvas class is similar to Flash's `Graphics` object. This will e.g. draw a red circle:

```javascript
var canvas:Canvas = new Canvas();
canvas.beginFill(0xff0000);
canvas.drawCircle(0, 0, 120);
canvas.endFill();
```

There are also methods to draw an ellipse, a rectangle or an arbitrary polygon.

Other than those basic methods, the Canvas class is rather limited; don't expect a full-blown alternative to the Graphics class just yet. This might change in a future release, though!

That brings us to the Polygon class. A Polygon (package starling.geom) describes a closed shape defined by a number of straight line segments. It's a spiritual successor of Flash's Rectangle class, but supporting (mostly) arbitrary shapes.

Since Canvas contains direct support for polygon objects, it's the ideal companion of Polygon. This pair of classes will solve all your mask-related needs.

```javascript
var polygon:Polygon = new Polygon(); ①
polygon.addVertices(0,0, 100,0, 0,100);

var canvas:Canvas = new Canvas();
canvas.beginFill(0xff0000);
canvas.drawPolygon(polygon); ②
canvas.endFill();
```
This polygon describes a triangle.

Draw the triangle to a canvas.

There are a few more things about masks I want to note:

**Visibility**

The mask itself is never visible. You always only see it indirectly via its effect on the masked display object.

**Positioning**

If the mask is not part of the display list (i.e. it has no parent), it will be drawn in the local coordinate system of the masked object: if you move the object, the mask will follow. If the mask *is* part of the display list, its location will be calculated just as usual. Depending on the situation, you might prefer one or the other behavior.

**Stencil Buffer**

Behind the scenes, masks use the *stencil buffer* of the GPU, making them very lightweight and fast. One mask requires two draw calls: one to draw the mask into the stencil buffer and one to remove it when all the masked content has been rendered.

**Scissor Rectangle**

If the mask is an untextured *Quad* parallel to the stage axes, Starling can optimize its rendering. Instead of the *stencil buffer*, it will then use the *scissor rectangle* — sparing you one draw call.

**Texture Masks**

If a simple vector shape just doesn’t cut it, there is an extension that allows you to use the alpha channel of a texture as a stencil mask. It’s called *Texture Mask* and is found in the Starling Wiki.

### 2.11. Sprite3D

All display objects that we looked at in the previous sections represent pure two-dimensional objects. That’s to be expected — Starling is a 2D engine, after all. However, even in a 2D game, it’s sometimes nice to add a simple 3D effect, e.g. for transitioning between two screens or to show the backside of a playing card.

For this reason, Starling contains a class that makes it easy to add basic 3D capabilities: *Sprite3D*. It allows you to move your 2D objects around in a three dimensional space.

#### 2.11.1. Basics

Just like a conventional *Sprite*, you can add and remove children to this container, which allows you to group several display objects together. In addition to that, however, *Sprite3D* offers several interesting properties:

- **z** — Moves the sprite along the z-axis (which points away from the camera).
- **rotationX** — Rotates the sprite around the x-axis.
- **rotationY** — Rotates the sprite around the y-axis.
• `scaleZ` — Scales the sprite along the z-axis.
• `pivotZ` — Moves the pivot point along the z-axis.

With the help of these properties, you can place the sprite and all its children in the 3D world.

```javascript
var sprite: Sprite3D = new Sprite3D(); ①
sprite.addChild(image1); ②
sprite.addChild(image2);

sprite.x = 50; ③
sprite.y = 20;
sprite.z = 100;
sprite.rotationX = Math.PI / 4.0;

addChild(sprite); ④
```

① Create an instance of `Sprite3D`.
② Add a few conventional 2D objects to the sprite.
③ Set up position and orientation of the object in 3D space.
④ As usual, add it to the display list.

As you can see, it’s not difficult to use a `Sprite3D`: you simply have a few new properties to explore. Hit-testing, animations, custom rendering — everything works just like you’re used to from other display objects.

### 2.11.2. Camera Setup

Of course, if you’re displaying 3D objects, you also want to be able to configure the perspective with which you’re looking at those objects. That’s possible by setting up the camera; and in Starling, the camera settings are found on the stage.

The following stage properties set up the camera:

• `fieldOfView` — Specifies an angle (radian, between zero and PI) for the field of view (FOV).
• `focalLength` — The distance between the stage and the camera.
• `projectionOffset` — A vector that moves the camera away from its default position, which is right in front of the center of the stage.
Figure 34. Those are the properties that set up the camera.

Starling will always make sure that the stage will fill the entire viewport. If you change the field of view, the focal length will be modified to adhere to this constraint, and the other way round. In other words: _fieldOfView_ and _focalLength_ are just different representations of the same property.

Here’s an example of how different _fieldOfView_ values influence the look of the cube from the Starling demo:

![Figure 35. Different values for fieldOfView (in degrees).](image)

Per default, the camera will always be aligned so that it points towards the center of the stage. The _projectionOffset_ allows you to change the perspective away from this point; use it if you want to look at your objects from another direction, e.g. from the top or bottom. Here’s the cube again, this time using different settings for _projectionOffset.y_:

![Figure 36. Different values for projectionOffset.y.](image)
2.11.3. Limitations

Starling is still a 2D engine at its heart, and this means that there are a few limitations you should be aware of:

- Starling does not make any depth tests. Visibility is determined solely by the order of children.
- You need to be careful about the performance. Each Sprite3D instance interrupts batching.

However, there's a trick that mitigates the latter problem in many cases: when the object is not actually 3D transformed, i.e. you're doing nothing that a 2D sprite couldn't do just as well, then Starling treats it just like a 2D object — with the same performance and batching behavior.

This means that you don't have to avoid having a huge number of Sprite3D instances; you just have to avoid that too many of them are 3D-transformed at the same time.

2.11.4. Sample Project

I created a video tutorial that demonstrates how this feature can be used in a real-life project. It shows you how to move a 2D game of concentration into the third dimension.

- Watch the video on Vimeo.
- Get the complete source code from GitHub.

2.12. Utilities

The starling.utils package contains several useful little helpers that shouldn't be overlooked.

2.12.1. Colors

In both conventional Flash and Starling, colors are specified in hexadecimal format. Here are a few examples:

```javascript
// format:         0xRRGGBB
var red:Number    = 0xff0000;
var green:Number  = 0x00ff00;  // or 0xff00
var blue:Number   = 0x0000ff;  // or 0xff
var white:Number  = 0xffffff;
var black:Number  = 0x000000;  // or simply 0
```

The Color class contains a list of named color values; furthermore, you can use it to easily access the components of a color.
```javascript
var purple:uint = Color.PURPLE; ①
var lime:uint = Color.LIME;
var yellow:uint = Color.YELLOW;

var color:uint = Color.rgb(64, 128, 192); ②

var red:int = Color.getRed(color); // -> 64 ③
var green:int = Color.getGreen(color); // -> 128
var blue:int = Color.getBlue(color); // -> 192
```

① A few common colors are predefined.
② Any other color can be created with this method. Just pass the RGB values to this method (range: 0 - 255).
③ You can also extract the integer value of each channel.

### 2.12.2. Angles

Starling expects all angles in radians (different to Flash, which uses degrees in some places and radians in others). To convert between degrees and radians, you can use the following simple functions.

```javascript
var degrees:Number = rad2deg(Math.PI); // -> 180
var radians:Number = deg2rad(180); // -> PI
```

### 2.12.3. StringUtil

You can use the `format` method to format Strings in .Net/C# style.

```javascript
StringUtil.format("{0} plus {1} equals {2}", 4, 3, "seven");
// -> "4 plus 3 equals seven"
```

The same class also contains methods that trim whitespace from the start and end of a string—a frequent operation whenever you need to process user input.

```javascript
StringUtil.trim("  hello world\n"); // -> "hello world"
```

### 2.12.4. SystemUtil

It's often useful to find out information about the environment an app or game is currently executed in. The `SystemUtil` contains some methods and properties helping with that task.
2.12.5. MathUtil

While that class is mainly designed to help with some geometric problems, it also contains the following very useful helper methods:

```javascript
var min: Number = MathUtil.min(1, 10); ①
var max: Number = MathUtil.max(1, 10); ②
var inside: Number = MathUtil.clamp(-5, 1, 10); ③
```

① Get the smallest of two numbers. Result: 1
② Get the biggest of two numbers. Result: 10
③ Move the number (first argument) into a specific range. Result: 1

If you have worked with AS3 in the past, you might wonder why I made the effort of writing those methods when similar ones are already provided in the native Math class.

Unfortunately, those equivalent methods have a side effect: each time you call e.g. Math.min, it creates a temporary object (at least when you compile your app for iOS, that is). Those alternatives do not have this side effect, so you should always prefer them.

2.12.6. Pooling

Now that we touched the topic of temporary objects, it’s the perfect time to introduce you to the Pool class.

Experienced AS3 developers know that any object allocation comes at a price: the object needs to be garbage collected later. This happens completely behind the scenes; you won’t even notice it most of the time.

However, when the cleanup process takes up too much time, your app will freeze for a short moment. If that happens often, it quickly becomes a nuisance to your users.

One tactic to avoid this problem is to recycle your objects and use them repeatedly. For example, classes like Point and Rectangle are often just needed for a short moment: you create them, fill them with some data, and then throw them away.

From now on, let Starling’s Pool class handle those objects.
```javascript
var point:Point = Pool.getPoint(); ①
doSomethingWithPoint(point);
Pool.putPoint(point); ②

var rect:Rectangle = Pool.getRectangle(); ①
doSomethingWithRectangle(rect);
Pool.putRectangle(rect); ②
```

① Get an object from the pool. That replaces calling `new` on the class.
② Put it back into the pool when you do not need it any longer.

The class also supports `Vector3D`, `Matrix`, and `Matrix3D`, in a similar style.

Always make sure that the `get` and `put`-calls are balanced. If you put too many objects into the pool and never retrieve them, it will fill up over time, using more and more memory.

2.12.7. Furthermore ...

The `starling.utils` package contains more helpers than I can possibly list here. For a complete list of methods and classes, refer to the API Reference. It will definitely pay off to take a look!

2.13. Summary

You are now familiar with all the basic concepts of the Starling Framework. This is all the knowledge you need to get started with that game you have in your head, or with that app that’s waiting to be created.

On the other hand, there are still a few tricks up the sleeves of our little bird (birds have sleeves?). If you’re ready to jump into those advanced topics, please follow my lead.
Chapter 3. Advanced Topics

With all the things we learned in the previous chapters, you are already well equipped to start using Starling in real projects. However, while doing so, you might run into a few things that might be puzzling. For example,

• your textures are quickly consuming all available memory.
• you are encountering a context loss from time to time. WTF!? [1: My editor said it's not polite to swear, but (1) I used an acronym and (2) context loss really s*cks.]
• you are actually a little disappointed by the performance of your application. You want more speed!
• or you might be one of those masochists who likes to write their own vertex and fragment shaders, but didn't know where to start.

Oddly enough, that perfectly summarizes what this chapter is about. Buckle up Dorothy, we are now jumping into some advanced topics!

3.1. ATF Textures

In conventional Flash, most developers use the PNG format for their images, or JPG if they don’t need transparency. Those are very popular in Starling, too. However, Stage3D offers an alternative that has several unique advantages: the Adobe Texture Format, which can store compressed textures.

• Compressed textures require just a fraction of their conventional counterparts.
• Decompression is done directly on the GPU.
• Uploading to graphics memory is faster.
• Uploading can be done asynchronously: you can load new textures without interrupting gameplay. (Beginning with AIR 24, this is possible with conventional textures, as well.)

3.1.1. Graphics Memory

Before we go on, it might be interesting to know how much memory is required by a texture, anyway.

A PNG image stores 4 channels for every pixel: red, green, blue, and alpha, each with 8 bit (that makes 256 values per channel). It’s easy to calculate how much space a 512 × 512 pixel texture takes up:

Memory footprint of a 512 × 512 RGBA texture:
512 × 512 pixels × 4 bytes = 1,048,576 bytes ≈ 1 MB

When you've got a JPG image, it's similar; you just spare the alpha channel.
Memory footprint of a $512 \times 512$ RGB texture:
$512 \times 512 \text{ pixels} \times 3 \text{ bytes} = 786,432 \text{ bytes} \approx 768 \text{ kB}$

Quite a lot for such a small texture, right? Beware that the built-in file compression of PNG and JPG does not help: the image has to be decompressed before Stage3D can handle it. In other words: the file size does not matter; the memory consumption is always calculated with the above formula.

Nevertheless: if your textures easily fit into graphics memory that way — go ahead and use them! Those formats are very easy to work with and will be fine in many situations, especially if your application is targeting desktop hardware.

However, there might come a moment in the development phase where your memory consumption is higher than what is available on the device. This is the right time to look at the ATF format.

### 3.1.2. Compressed Textures

Above, we learned that the file size of a conventional texture has nothing to do with how much graphics memory it uses; a massively compressed JPG will take up just as much space as the same image in pure BMP format.

This is not true for compressed textures: they can be processed directly on the GPU. This means that, depending on the compression settings, you can load up to ten times as many textures. Quite impressive, right?

Unfortunately, each GPU vendor thought he could do better than the others, and so there are several different formats for compressed textures. In other words: depending on where your game is running, it will need a different kind of texture. How should you know beforehand which file to include?

This is where ATF comes to the rescue. It is a format that Adobe created especially for Stage3D; actually, it is a container file that can include up to four different versions of a texture.

- **PVRTC** (PowerVR Texture Compression) is used in PowerVR GPUs. It is supported by all generations of the iPhone, iPod Touch, and iPad.
- **DXT1/5** (S3 Texture Compression) was originally developed by S3 Graphics. It is now supported by both Nvidia and AMD GPUs, and is thus available on most desktop computers, as well as some Android phones.
- **ETC1** (Ericsson Texture Compression) is used on many mobile phones, most notably on Android.
- **ETC2** provides higher quality RGB and RGBA compression. It is supported by all Android and iOS devices that also support OpenGL ES 3.

I wrote before that ATF is a *container* format. That means that it can include any combination of the above formats.
When you include all formats (which is the default), the texture can be loaded on any Stage3D-supporting device, no matter if your application is running on iOS, Android, or on the Desktop. You don’t have to care about the internals!

However, if you know that your game will only be deployed to, say, iOS devices, you can omit all formats except PVRC. Or if you’re only targeting high end mobile devices (with at least *OpenGL ES 3*), include only ETC2; that works on both Android and iOS. That way, you can optimize the download size of your game.

The difference between DXT1 and DXT5 is just that the latter supports an alpha channel. Don’t worry about this, though: the ATF tools will choose the right format automatically.

ETC1 actually does not support an alpha channel, but Stage3D works around this by using two textures internally. Again, this happens completely behind the scenes.

### 3.1.3. Creating an ATF texture

Adobe provides a set of command line tools to convert to and from ATF and to preview the generated files. They are part of the AIR SDK (look for the `atftools` folder).

Probably the most important tool is `png2atf`. Here is a basic usage example; it will compress the texture with the standard settings in all available formats.

```
png2atf -c -i texture.png -o texture.atf
```

If you tried that out right away, you probably received the following error message, though:

```
Dimensions not a power of 2!
```

That’s a limitation I have not mentioned yet: ATF textures are required to always have side-lengths that are powers of two. While this is a little annoying, it’s actually rarely a problem, since you will almost always use them for atlas textures.
Most atlas generators can be configured so that they create power-of-two textures.

When the call succeeds, you can review the output in the ATFViewer.

![ATFViewer](image)

**Figure 38. The ATFViewer tool.**

In the list on the left, you can choose which internal format you want to view. Furthermore, you see that, per default, all mipmap variants have been created.

We will discuss mipmaps in the Memory Management chapter.

You will probably also notice that the image quality has suffered a bit from the compression. This is because all those compression formats are *lossy*: the smaller memory footprint comes at the prize of a reduced quality. How much the quality suffers is depending on the type of image: while organic, photo-like textures work well, comic-like images with hard edges can suffer quite heavily.

The tool provides a lot of different options, of course. E.g. you can let it package only the PVRTC format, perfect for iOS:

```
png2atf -c p -i texture.png -o texture.atf
```

Or you can tell it to omit mipmaps in order to save memory:
Another useful utility is called `atfinfo`. It displays details about the data that's stored in a specific ATF file, like the included texture formats, the number of mipmaps, etc.

```plaintext
> atfinfo -i texture.atf

File Name          : texture.atf
ATF Version        : 2
ATF File Type      : RAW Compressed With Alpha (DXT5+ETC1/ETC1+PVRTV4bpp)
Size               : 256x256
Cube Map           : no
Empty Mipmaps      : no
Actual Mipmaps     : 1
Embedded Levels    : X........ (256x256)
AS3 Texture Class  : Texture (flash.display3D.Texture)
AS3 Texture Format : Context3DTextureFormat.COMPRESSED_ALPHA
```

### 3.1.4. Using ATF Textures

Using a compressed texture in Starling is just as simple as any other texture. Pass the byte array with the file contents to the factory method `Texture.fromAtfData()`.

```javascript
var atfData:ByteArray = getATFBytes(); ①
var texture:Texture = Texture.fromAtfData(atfData); ②
var image:Image = new Image(texture); ③
```

① Get the raw data e.g. from a file.

② Create the ATF texture.

③ Use it like any other texture.

That's it! This texture can be used like any other texture in Starling. It's also a perfectly suitable candidate for your atlas texture.

However, the code above will upload the texture synchronously, i.e. AS3 execution will pause until that's done. To load the texture asynchronously instead, pass a callback to the method:

```javascript
Texture.fromAtfData(atfData, 1, true,
    function(texture:Texture):void
    {
        var image:Image = new Image(texture);
    });
```

Parameters two and three control the scale factor and if mipmaps should be used, respectively. The fourth one, if passed a callback, will trigger asynchronous loading: Starling will be able to continue
rendering undisturbed while that happens. As soon as the callback has been executed (but not any sooner!), the texture will be usable.

Of course, you can also embed the ATF file directly in the AS3 source.

```
[Embed(source="texture.atf", mimeType="application/octet-stream")]
public static const CompressedData:Class;

var texture:Texture = Texture.fromEmbeddedAsset(CompressedData);
```

Note, however, that asynchronous upload is not available in this case.

### 3.1.5. Other Resources

You can find out more about this topic in the following sources:

- Introducing Compressed Textures
- ATF Tools User's Guide

### 3.2. Context Loss

All Stage3D rendering happens through a so called "render context" (an instance of the `Context3D` class). It stores all current settings of the GPU, like the list of active textures, pointers to the vertex data, etc. The render context is your connection to the GPU — without it, you can’t do any Stage3D rendering.

And here comes the problem: that context can sometimes get lost. This means that you lose references to all data that was stored in graphics memory; most notably: textures.

Such a context loss doesn’t happen equally frequently on all systems; it’s rare on iOS and macOS, happens from time to time on Windows and very often on Android (rotating the screen? Bam!). So there’s no way around it: we need to expect the worst and prepare for a context loss.

**How to trigger a context loss**

There is an easy way to check if your application can handle a context loss: simply dispose the current context via `Starling.context.dispose()`. It will immediately be recreated, which is just what happens after the real thing.

### 3.2.1. Default Behavior

When Starling recognizes that the current render context has been lost, it initiates the following procedures:

- Starling will automatically create a new context and initialize it with the same settings as before.
- All vertex- and index buffers will be restored.
• All vertex- and fragment programs (shaders) will be recompiled.

• Textures will be restored by whatever means possible (from memory/disk/etc.)

Restoring buffers and programs is not problematic; Starling has all data that’s required and it
doesn’t take much time. Textures, however, are a headache. To illustrate that, let’s look at the worst
case example: a texture created from an embedded bitmap.

```javascript
[Embed(source="hero.png")]
public static const Hero:Class;

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

The moment you call `Texture.fromBitmap`, the bitmap is uploaded to GPU memory, which means it’s
now part of the context. If we could rely on the context staying alive forever, we’d be done now.

However, we cannot rely on that: the texture data could be lost anytime. That’s why Starling will
keep a copy of the original bitmap. When the worst happens, it will use it to recreate the texture. All
of that happens behind the scenes.

Lo and behold! That means that the texture is in memory three times.

• The "Hero" class (conventional memory)
• The backup bitmap (conventional memory)
• The texture (graphics memory)

Given the tight memory constraints we’re facing on mobile, this is a catastrophe. You don’t want
this to happen!

It becomes a little better if you change your code slightly:

```javascript
// use the 'fromEmbeddedAsset' method instead
var texture:Texture = Texture.fromEmbeddedAsset(Hero);
```

That way, Starling can recreate the texture directly from the embedded class (calling `new Hero()`),
which means that the texture is in memory only two times. For embedded assets, that’s your best
bet.

Ideally, though, we want to have the texture in memory only once. For this to happen, you must not
embed the asset; instead, you need to load it from an URL that points to a local or remote file. That
way, only the URL needs to be stored; the actual data can then be reloaded from the original
location.

There are two ways to make this happen:

• Use the `AssetManager` to load your textures.
• Restore the texture manually.
My recommendation is to use the AssetManager whenever possible. It will handle a context loss without wasting any memory; you don’t have to add any special restoration logic whatsoever.

Nevertheless, it’s good to know what’s happening behind the scenes. Who knows — you might run into a situation where a manual restoration is your only choice.

### 3.2.2. Manual Restoration

You might wonder how Texture.fromEmbeddedAsset() works internally. Let’s look at a possible implementation of that method:

```javascript
public static function fromEmbeddedAsset(assetClass:Class):Texture
{
  var texture:Texture = Texture.fromBitmap(new assetClass());
  texture.root.onRestore = function():void
  {
    texture.root.uploadFromBitmap(new assetClass());
  }
  return texture;
}
```

You can see that the magic is happening in the root.onRestore callback. Wait a minute: what’s root?

You might not know it, but when you’ve got a Texture instance, that’s actually often not a concrete texture at all. In reality, it might be just a pointer to a part of another texture (a SubTexture). Even the fromBitmap call could return such a texture! (Explaining the reasoning behind that would be beyond the scope of this chapter, though.)

In any case, texture.root will always return the ConcreteTexture object, and that’s where the onRestore callback is found. This callback will be executed directly after a context loss, and it gives you the chance of recreating your texture.

In our case, that callback simply instantiates the bitmap once again and uploads it to the root texture. Voilà, the texture is restored!

The devil lies in the details, though. You have to construct your onRestore-callback very carefully to be sure not to store another bitmap copy without knowing it. Here’s one innocent looking example that’s actually totally useless:
Can you spot the error?

The problem is that the method creates a `Bitmap` object and uses it in the callback. That callback is actually a so-called *closure*; that's an inline function that will be stored together with some of the variables that accompany it. In other words, you've got a function object that stays in memory, ready to be called when the context is lost. And the bitmap instance is stored inside it, even though you never explicitly said so. (Well, in fact you did, by using `bitmap` inside the callback.)

In the original code, the bitmap is not referenced, but *created* inside the callback. Thus, there is no `bitmap` instance to be stored with the closure. Only the `assetClass` object is referenced in the callback — and that is in memory, anyway.

That technique works in all kinds of scenarios:

- If your texture originates from an URL, you pass only that URL to the callback and reload it from there.
- For ATF textures, the process is just the same, except that you need to upload the data with `root.uploadATFData` instead.
- For a bitmap containing a rendering of a conventional display object, just reference that display object and draw it into a new bitmap in the callback. (That's just what Starling's `TextField` class does.)

Let me emphasize: the `AssetManager` does all this for you, so that's the way to go. I just wanted to show you how that is achieved.

### 3.2.3. Render Textures

Another area where a context loss is especially nasty: render textures. Just like other textures, they will lose all their contents — but there's no easy way to restore them. After all, their content is the result of any number of dynamic draw operations.

If the `RenderTexture` is just used for eye candy (say, footprints in the snow), you might be able to just live with it getting cleared. If its content is crucial, on the other hand, you need a solution for this problem.
There’s no way around it: you will need to manually redraw the texture’s complete contents. Again, the `onRestore` callback could come to the rescue:

```javascript
renderTexture.root.onRestore = function():void
{
    var contents:Sprite = getContents();
    renderTexture.clear(); // required on texture restoration
    renderTexture.draw(contents);
};
```

I hear you: it’s probably more than just one object, but a bunch of draw calls executed over a longer period. For example, a drawing app with a `RenderTexture`-canvas, containing dozens of brush strokes.

In such a case, you need to store sufficient information about all draw commands to be able to reproduce them.

If we stick with the drawing app scenario, you might want to add support for an undo/redo system, anyway. Such a system is typically implemented by storing a list of objects that encapsulate individual commands. You can re-use that system in case of a context loss to restore all draw operations.

Now, before you start implementing this system, there is one more gotcha you need to be aware of. When the `root.onRestore` callback is executed, it’s very likely that not all of your textures are already available. After all, they need to be restored, too, and that might take a while!

If you loaded your textures with the `AssetManager`, however, it has got you covered. In that case, you can listen to its `TEXTURES_RESTORED` event instead. Also, make sure to use `drawBundled` for optimal performance.

```javascript
assetManager.addEventListener(Event.TEXTURES_RESTORED, function():void
{
    renderTexture.drawBundled(function():void
    {
        for each (var command:DrawCommand in listOfCommands)
        {
            command.redraw(); // executes `renderTexture.draw()`
        }
    });
});
```

This time, there is no need to call `clear`, because that’s the default behavior of `onRestore`, anyway—and we did not modify that. Remember, we are in a different callback here (Event.TEXTURES_RESTORED), and `onRestore` has not been modified from its default implementation.

### 3.3. Memory Management

Many Starling developers use the framework to create apps and games for mobile devices. And
almost all of those developers will sooner or later find out (the hard way) that mobile devices are
notoriously low on memory. Why is that?

- Most mobile devices have screens with extremely high resolutions.
- 2D games for such devices require equally high resolution textures.
- The available RAM is too small to hold all that texture data.

In other words, a really vicious combination.

What happens if you do run out of memory? Most of the time, you will get the famous error 3691
("Resource limit for this resource type exceeded") and your app will crash. The following hints will
show you ways to avoid this nasty error!

3.3.1. Dispose your Waste

When you don't need an object any longer, don't forget to call `dispose` on it. Different to
conventional Flash objects, the garbage collector will not clean up any Stage3D resources! You are
responsible for that memory yourself.

Textures

Those are the most important objects you need to take care of. Textures will always take up the
biggest share of your memory.

Starling tries to help you with this, of course. For example, when you load your textures from an
atlas, you only need to dispose the atlas, not the actual SubTextures. Only the atlas requires GPU
memory, the "offspring" textures will just reference the atlas texture.

```javascript
var atlas:TextureAtlas = ...;
var hero:Texture = atlas.getTexture("hero");
atlas.dispose(); // will invalidate "hero" as well.
```

Display Objects

While display objects themselves do not require a lot of graphics memory (some do not require any
at all), it's a good practice to dispose them, too. Be especially careful with "heavy" objects like
TextFields.

Display object containers will take care of all their children, as is to be expected. When you dispose
a container, all children will be disposed automatically.
```javascript
var parent:Sprite = new Sprite();
var child1:Quad = new Quad(100, 100, Color.RED);
var child2:Quad = new Quad(100, 100, Color.GREEN);

parent.addChild(child1);
parent.addChild(child2);

parent.dispose(); // will dispose the children, too
```

All in all, though, recent Starling versions have become more forgiving when it comes to disposing display objects. Most display objects do not store Stage3D resources any longer, so it’s not a catastrophe if you forget to dispose one.

**Images**

Here’s the first pitfall: disposing an image will *not* dispose its texture.

```javascript
var texture:Texture = Texture.fromBitmap(/* ... */);
var image:Image = new Image(texture);

image.dispose(); // will NOT dispose texture!
```

That’s because Starling can’t know if you’re using this texture anywhere else! After all, you could have other images that use the same texture.

On the other hand, if you know that the texture is not used anywhere else, get rid of it.

```javascript
image.texture.dispose();
image.dispose();
```

**Filters**

Fragment filters are a little delicate, too. When you dispose an object, the filter will be disposed, as well:

```javascript
var object:Sprite = createCoolSprite();
object.filter = new BlurFilter();
object.dispose(); // will dispose filter
```

But watch out: the following similar code will not dispose the filter:

```javascript
var object:Sprite = createCoolSprite();
object.filter = new BlurFilter();
object.filter = null; // filter will *not* be disposed
```
Again, the reason is that Starling can’t know if you want to use the filter elsewhere.

However, in practice, this is not a problem. The filter is not disposed, but Starling will still clean up all its resources. So you won’t create a memory leak.

In previous Starling versions (< 2.0), this *did* create a memory leak.

### 3.3.2. Do not Embed Textures

ActionScript developers have always been used to embedding their bitmaps directly into the SWF file, using `Embed` metadata. This is great for the web, because it allows you to combine all your game’s data into one file.

We already saw in the Context Loss section that this approach has some serious downsides in Starling (or Stage3D in general). It comes down to this: the texture will be in memory at least two times: once in conventional memory, once in graphics memory.

```actionscript
private static var Hero:Class; ①

var texture:Texture = Texture.fromEmbeddedAsset(Hero); ②
```

① The class is stored in conventional memory.

② The texture is stored in graphics memory.

Note that this sample uses `Texture.fromEmbeddedAsset` to load the texture. For reasons discussed in Context Loss, the alternative (`Texture.fromBitmap`) uses even more memory.

The only way to guarantee that the texture is really only stored in graphics memory is by loading it from an URL. If you use the `AssetManager` for this task, that’s not even a lot of work.

```actionscript
var appDir:File = File.applicationDirectory;
var assets:AssetManager = new AssetManager();

assets.enqueue(appDir.resolvePath("assets/textures"));
assets.loadQueue(...);

var texture:Texture = assets.getTexture("hero");
```

### 3.3.3. Use RectangleTextures

Starling’s `Texture` class is actually just a wrapper for two Stage3D classes:

- `flash.display3D.textures.Texture`
  - Available in all profiles. Supports mipmaps and wrapping, but requires side-lengths that are powers of two.
flash.display3D.textures.RectangleTexture

Available beginning with BASELINE profile. No mipmaps, no wrapping, but supports arbitrary side-lengths.

The former (Texture) has a strange and little-known side effect: it will always allocate memory for mipmaps, whether you need them or not. That means that you will waste about one third of texture memory!

Thus, it’s preferred to use the alternative (RectangleTexture). Starling will use this texture type whenever possible.

However, it can only do that if you run at least in BASELINE profile, and if you disable mipmaps. The first requirement can be fulfilled by picking the best available Context3D profile. That happens automatically if you use Starling’s default constructor.

```java
// init Starling like this:
... = new Starling(Game, stage);

// that's equivalent to this:
... = new Starling(Game, stage, null, null, "auto", "auto");
```

The last parameter (auto) will tell Starling to use the best available profile. This means that if the device supports RectangleTextures, Starling will use them.

As for mipmaps: they will only be created if you explicitly ask for them. Some of the Texture.from… factory methods contain such a parameter, and the AssetManager features a useMipMaps property. Per default, they are always disabled.

### 3.3.4. Use ATF Textures

We already talked about ATF Textures previously, but it makes sense to mention them again in this section. Remember, the GPU cannot make use of JPG or PNG compression; those files will always be decompressed and uploaded to graphics memory in their uncompressed form.

Not so with ATF textures: they can be rendered directly from their compressed form, which saves a lot of memory. So if you skipped the ATF section, I recommend you take another look!

The downside of ATF textures is the reduced image quality, of course. But while it’s not feasible for all types of games, you can try out the following trick:

1. Create your textures a little bigger than what’s actually needed.
2. Now compress them with the ATF tools.
3. At runtime, scale them down to their original size.

You’ll still save a quite a bit of memory, and the compression artifacts will become less apparent.
3.3.5. Use 16 bit Textures

If ATF textures don’t work for you, chances are that your application uses a comic-style with a limited color palette. I’ve got good news for you: for these kinds of textures, there’s a different solution!

- The default texture format (`Context3DTextureFormat.BGRA`) uses 32 bits per pixel (8 bits for each channel).
- There is an alternative format (`Context3DTextureFormat.BGRA_PACKED`) that uses only half of that: 16 bits per pixel (4 bits for each channel).

You can use this format in Starling via the `format` argument of the `Texture.from...` methods, or via the AssetManager’s `textureFormat` property. This will save you 50% of memory!

Naturally, this comes at the price of a reduced image quality. Especially if you’re making use of gradients, 16 bit textures might become rather ugly. However, there’s a solution for this: dithering!

![Dithering Comparison](image)

*Figure 39. Dithering can conceal a reduced color depth.*

To make it more apparent, the gradient in this sample was reduced to just 16 colors (4 bits). Even with this low number of colors, dithering manages to deliver an acceptable image quality.

Most image processing programs will use dithering automatically when you reduce the color depth. `TexturePacker` has you covered, as well.

The `AssetManager` can be configured to select a suitable color depth on a per-file basis.

```javascript
var assets:AssetManager = new AssetManager();

// enqueue 16 bit textures
assets.textureOptions.format = Context3DTextureFormat.BGRA_PACKED;
assets.enqueue(/* ... */);

// enqueue 32 bit textures
assets.textureOptions.format = Context3DTextureFormat.BGRA;
assets.enqueue(/* ... */);

// now start the loading process
assets.loadQueue(/* ... */);
```
3.3.6. Avoid Mipmaps

Mipmaps are downsampled versions of your textures, intended to increase rendering speed and reduce aliasing effects.

![Sample of a texture with mipmaps.](image)

Since version 2.0, Starling doesn’t create any mipmaps by default. That turned out to be the preferable default, because without mipmaps:

- Textures load faster.
- Textures require less texture memory (just the original pixels, no mipmaps).
- Blurry images are avoided (mipmaps sometimes become fuzzy).

On the other hand, activating them will yield a slightly faster rendering speed when the object is scaled down significantly, and you avoid aliasing effects (i.e. the effect contrary to blurring). To enable mipmaps, use the corresponding parameter in the `Texture.from...` methods.

3.3.7. Use Bitmap Fonts

As already discussed, TextFields support two different kinds of fonts: TrueType fonts and Bitmap Fonts.

While TrueType fonts are very easy to use, they have a few downsides.

- Whenever you change the text, a new texture has to be created and uploaded to graphics memory. This is slow.
- If you've got many TextFields or big ones, this will require a lot of texture memory.

Bitmap Fonts, on the other hand, are

- updated very quickly and
- require only a constant amount of memory (just the glyph texture).

That makes them the preferred way of displaying text in Starling. My recommendation is to use them whenever possible!
Bitmap Font textures are a great candidate for 16 bit textures, because they are often just pure white that’s tinted to the actual TextField color at runtime.

### 3.3.8. Optimize your Texture Atlas

It should be your top priority to pack your texture atlases as tightly as possible. Tools like TexturePacker have several options that will help with that:

- Trim transparent borders away.
- Rotate textures by 90 degrees if it leads to more effective packing.
- Reduce the color depth (see above).
- Remove duplicate textures.
- etc.

Make use of this! Packing more textures into one atlas not only reduces your overall memory consumption, but also the number of draw calls (more on that in the next chapter).

### 3.3.9. Use Adobe Scout

Adobe Scout is a lightweight but comprehensive profiling tool for ActionScript and Stage3D. Any Flash or AIR application, regardless of whether it runs on mobile devices or in browsers, can be quickly profiled with no change to the code—and Adobe Scout quickly and efficiently detects problems that could affect performance.

With Scout, you can not only find performance bottlenecks in your ActionScript code, but you’ll also find a detailed roundup of your memory consumption over time, both for conventional and graphics memory. This is priceless!

Adobe Scout is part of the free version of Adobe’s Creative Cloud membership. You don’t have to become a paying subscriber of CC to get it.

Here is a great tutorial from Thibault Imbert that explains in detail how to work with Adobe Scout: [Getting started with Adobe Scout](#).
3.3.10. Keep an Eye on the Statistics Display

The statistics display (available via `starling.showStats`) includes information about both conventional memory and graphics memory. It pays off to keep an eye on these values during development.

Granted, the conventional memory value is often misleading—you never know when the garbage collector will run. The graphics memory value, on the other hand, is extremely accurate. When you create a texture, the value will rise; when you dispose a texture, it will decrease—immediately.

Actually, just about five minutes after I added this feature to Starling, I had already found the first memory leak—in the demo app. I used the following approach:

- In the main menu, I noted down the used GPU memory.
- Then I entered the demos scenes, one after another.
- Each time I returned to the main menu, I checked if the GPU memory had returned to the original value.
- After returning from one of the scenes, that value was not restored, and indeed: a code review showed that I had forgotten to dispose one of the textures.

Needless to say: Scout offers far more details on memory usage. But the simple fact that the statistics display is always available makes it possible to find things that would otherwise be easily overlooked.
3.4. Performance Optimization

While Starling mimics the classic display list of Flash, what it does behind the scenes is quite different. To achieve the best possible performance, you have to understand some key concepts of its architecture. Here is a list of best practices you can follow to have your game run as fast as possible.

3.4.1. General AS3 Tips

Always make a Release Build

The most important rule right at the beginning: always create a release build when you test performance. Unlike conventional Flash projects, a release build makes a huge difference when you use a Stage3D framework. The speed difference is immense; depending on the platform you’re working on, you can easily get a multiple of the framerate of a debug build.

- In Flash Builder, release builds are created by clicking on Project › Export Release Build.
- In Flash Develop, choose the "Release" configuration and build the project; then choose "ipa-ad-hoc" or "ipa-app-store" option when you execute the "PackageApp.bat" script.
- In IntelliJ IDEA, select Build › Package AIR Application; choose "release" for Android and "ad hoc distribution" for iOS. For non-AIR projects, deselect "Generate debuggable SWF" in the module’s compiler options.
- If you build your Starling project from command line, make sure -optimize is true and -debug is false.

![Figure 43. Don’t get confused by this Flash Builder dialog.](image)

Check your Hardware

Be sure that Starling is indeed using the GPU for rendering. That’s easy to check: if Starling.current.context.driverInfo contains the string Software, then Stage3D is in software fallback mode, otherwise it’s using the GPU.

Furthermore, some mobile devices can be run in a Battery Saving Mode. Be sure to turn that off when making performance tests.

Set the Framerate

Your framerate is somehow stuck at 24 frames per second, no matter how much you optimize? Then you probably never set your desired framerate, and you’ll see the Flash Player’s default setting.
To change that, either use the appropriate metadata on your startup class, or manually set the framerate at the Flash stage.

```java
[SWF(frameRate="60", backgroundColor="#000000")]
public class Startup extends Sprite
{
/* ... */
}
```

// or anywhere else
Starling.current.nativeStage.frameRate = 60;

**Use Adobe Scout**

*Adobe Scout* is not only useful for memory analysis; it’s just as powerful when it comes to performance profiling.

It allows you to actually see how much time is actually spent in each of your (and Starling’s) *ActionScript* methods. This is extremely useful, because it shows you where you can gain most from any optimizations. Without it, you might end up optimizing areas of your code that are actually not relevant to the framerate at all!

💡 Remember, premature optimization is the root of all evil!

What’s nice compared to classic profilers is that it also works in release mode, with all optimizations in place. That ensures that its output is extremely accurate.

**Decode Loaded Images Asynchronously**

By default, if you use a *Loader* to load a PNG or JPEG image, the image data is not decoded right away, but when you first use it. This happens on the main thread and can cause your application to stutter on texture creation. To avoid that, set the image decoding policy flag to `ON_LOAD`. This will cause the image to be decoded directly in the Loader’s background thread.

```java
loaderContext.imageDecodingPolicy = ImageDecodingPolicy.ON_LOAD;
loader.load(url, loaderContext);
```

On the other hand, you are probably using Starling’s *AssetManager* to load your textures, aren’t you? In that case, don’t worry: it makes use of this practice, anyway.

**Avoid "for each"**

When working with loops that are repeated very often or are deeply nested, it’s better to avoid `for each`; the classic `for i yields a better performance. Furthermore, beware that the loop condition is executed once per loop, so it’s faster to save it into an extra variable.
Avoid creating a lot of temporary objects. They take up memory and need to be cleaned up by the garbage collector, which might cause small hiccups when it's running.

Actually, Starling contains a class that helps with that: Pool. It provides a pool of objects that are often required, like Point, Rectangle and Matrix. You can "borrow" objects from that pool and return them when you're done.

3.4.2. Starling Specific Tips
Minimize State Changes

As you know, Starling uses Stage3D to render the display list. This means that all drawing is done by the GPU.

Now, Starling could send one quad after the other to the GPU, drawing one by one. In fact, this is how the very first Starling release worked! For optimal performance, though, GPUs prefer to get a huge pile of data and draw all of it at once.

That's why newer Starling versions batch as many quads together as possible before sending them to the GPU. However, it can only batch quads that have similar properties. Whenever a quad with a different "state" is encountered, a "state change" occurs, and the previously batched quads are drawn.

I use Quad and Image synonymously in this section. Remember, Image is just a subclass of Quad that adds a few methods. Besides, Quad extends Mesh, and what you read below is true for meshes, as well.

These are the crucial properties that make up a state:

- The texture (different subtextures from the same atlas are fine, though)
- The blendMode of display objects
- The textureSmoothing value of meshes/quads/images
- The textureRepeat mode of meshes/quads/images

If you set up your scene in a way that creates as little state changes as possible, your rendering performance will profit immensely.

Again, Starling's statistics display provides useful data. It shows exactly how many draw calls are executed per frame. The more state changes you have, the higher this number will be.

The Painter's Algorithm

To know how to minimize state changes, you need to know the order in which Starling processes your objects.

Like Flash, Starling uses the Painter's algorithm to process the display list. This means that it draws
your scene like a painter would do it: starting at the object at the bottom layer (e.g. the background image) and moving upwards, drawing new objects on top of previous ones.

**Figure 45. Drawing a scene with the Painter's algorithm.**

If you’d set up such a scene in Starling, you could create three sprites: one containing the mountain range in the distance, one with the ground, and one with the vegetation. The mountain range would be at the bottom (index 0), the vegetation at the top (index 2). Each sprite would contain images that contain the actual objects.

**Figure 46. The scene graph of the landscape from above.**

On rendering, Starling would start at the left with "Mountain 1" and continue towards the right, until it reaches "Tree 2". If all those objects have a different state, this would mean six draw calls. That's exactly what will happen if you load each object's texture from a separate Bitmap.

**The Texture Atlas**

That's one of the reasons why texture atlases are so important. If you load all those textures from one single atlas, Starling will be able to draw all objects at once! (At least if the other properties listed above do not change.)
The consequence of this is that you should always use an atlas for your textures. Here, each image uses the same atlas (depicted by all nodes having the same color).

Sometimes, though, not all of your textures will fit into a single atlas. The size of textures is limited, so you’ll run out of space sooner or later. But this is no problem, as long as you arrange your textures in a smart way.

Both those examples use two atlases (again, one color per atlas). But while the display list on the left will force a state change for each object, the version on the right will be able to draw all objects in just two batches.

**Use the MeshBatch class**

The fastest way to draw a huge number of quads or other meshes at once is to use the `MeshBatch` class. That's the same class that is used internally by Starling for all rendering, so it's heavily optimized. It works like this:
var meshBatch:MeshBatch = new MeshBatch();
var image:Image = new Image(texture);

for (var i:int=0; i<100; ++i)
{
    meshBatch.addMesh(image);
    image.x += 10;
}
addChild(meshBatch);

Did you notice? You can add the same image as often as you want! Furthermore, adding it is a very fast operation; e.g. no event will be dispatched (which is the case when you add an object to a container).

As expected, this has some downsides, though:

• All the objects you add must have the same state (i.e. use textures from the same atlas). The first image you add to the MeshBatch will decide on its state. You can’t change the state later, except by resetting it completely.

• You can only add instances of the Mesh class or its subclasses (that includes Quad, Image, and even MeshBatch).

• Object removal is quite tricky: you can only remove meshes by trimming the number of vertices and indices of the batch. However, you can overwrite meshes at a certain index.

For these reasons, it’s only suitable for very specific use-cases (the BitmapFont class, for example, uses a mesh batch internally). In those cases, it’s definitely the fastest option, though. You won’t find a more efficient way to render a huge number of objects in Starling.

If you’re still using Starling 1.x, look for the QuadBatch class instead.

**Batch your TextField**

Per default, a TextField will require one draw call, even if your glyph texture is part of your main texture atlas. That’s because long texts require a lot of CPU time to batch; it’s faster to simply draw them right away (without copying them to a MeshBatch).

However, if your text field contains only a few letters (rule of thumb: below 16), you can enable the batchable property on the TextField. With that enabled, your texts will be batched just like other display objects.

**Use BlendMode.NONE**

If you’ve got totally opaque, rectangular textures, help the GPU by disabling blending for those textures. This is especially useful for large background images.

```actionscript
backgroundImage.blendMode = BlendMode.NONE;
```
Naturally, this will also mean an additional state change, so don’t overuse this technique. For small images, it’s probably not worth the effort (except if they’d cause a state change, anyway, for some other reason).

**Use stage.color**

Oftentimes, the actual stage color is actually never seen in your game, because there are always images or meshes on top of the stage.

In that case, always set it to clear black (0x0) or white (0xffffff). There seems to be a fast hardware optimization path for a `context.clear` on some mobile hardware when it is called with either all 1’s or all 0’s. Some developers reported a full millisecond of spared rendering time per frame, which is a very nice gain for such a simple change!

```swf(backgroundColor="#0")
public class Startup extends Sprite
{
  // ...
}
```

On the other hand, if the background of your game is a flat color, you can make use of that, too: just set the stage color to that value instead of displaying an image or a colored quad. Starling has to clear the stage once per frame, anyway — thus, if you change the stage color, that operation won’t cost anything.

```swf(backgroundColor="#ff2255")
public class Startup extends Sprite
{
  // ...
}
```

**Avoid querying width and height**

The `width` and `height` properties are more expensive than one would guess intuitively, especially on sprites. A matrix has to be calculated, and each vertex of each child will be multiplied with that matrix.

For that reason, avoid accessing them repeatedly, e.g. in a loop. In some cases, it might even make sense to use a constant value instead.
// bad:
for (var i:int=0; i<numChildren; ++i)
{
    var child:DisplayObject = getChildAt(i);
    if (child.x > wall.width)
        child.removeFromParent();
}

// better:
var wallWidth:Number = wall.width;
for (var i:int=0; i<numChildren; ++i)
{
    var child:DisplayObject = getChildAt(i);
    if (child.x > wallWidth)
        child.removeFromParent();
}

Make containers non-touchable

When you move the mouse/finger over the screen, Starling has to find out which object is hit. This can be an expensive operation, because it requires a hit-test on each and every display object (in the worst case).

Thus, it helps to make objects untouchable if you don’t care about them being touched, anyway. It’s best to disable touches on complete containers: that way, Starling won’t even have to iterate over their children.

// good:
for (var i:int=0; i<container.numChildren; ++i)
    container.getChildAt(i).touchable = false;

// even better:
container.touchable = false;

Hide objects that are outside the Stage bounds

Starling will send any object that is part of the display list to the GPU. This is true even for objects that are outside the stage bounds!

You might wonder: why doesn’t Starling simply ignore those invisible objects? The reason is that checking the visibility in a universal way is quite expensive. So expensive, in fact, that it’s faster to send objects up to the GPU and let it do to the clipping. The GPU is actually very efficient with that and will abort the whole rendering pipeline very early if the object is outside the screen bounds.

However, it still takes time to upload that data, and you can avoid that. Within the high level game logic, it’s often easier to make visibility checks (you can e.g. just check the x/y coordinates against a constant). If you’ve got lots of objects that are outside those bounds, it’s worth the effort. Remove those elements from the stage or set their visible property to false.
Make use of Event Pooling

Compared to classic Flash, Starling adds an additional method for event dispatching:

```javascript
// classic way:
object.dispatchEvent(new Event("type", bubbles));

// new way:
object.dispatchEventWith("type", bubbles);
```

The new approach will dispatch an event object just like the first one, but behind the scenes, it will pool event objects for you. That means that you will save the garbage collector some work.

In other words, it's less code to write and is faster — thus, it's the preferred way to dispatch events. (Except if you need to dispatch a custom subclass of Event; they cannot be dispatched with that method.)

3.5. Custom Filters

Are you ready to get your hands dirty? We are now entering the realm of custom rendering code, starting with a simple fragment filter.

Yes, this will involve some low level code; heck, you'll even write a few lines of assembler! But fear not, it's not rocket science. As my old math teacher used to say: a drilled monkey could do that!

Remember: filters work on the pixel level of display objects. The filtered object is rendered into a texture, which is then processed by a custom fragment shader (hence the name fragment filter).

3.5.1. The Goal

Even though we're picking a simple goal, it should be a useful one, right? So let's create a ColorOffsetFilter.

You probably know that you can tint any vertex of a mesh by assigning it a color. On rendering, the color will be multiplied with the texture color, which provides a very simple (and fast) way to modify the color of a texture.

```javascript
var image:Image = new Image(texture);
image.color = 0x808080; // R = G = B = 0.5
```

The math behind that is extremely simple: on the GPU, each color channel (red, green, blue) is represented by a value between zero and one. Pure red, for example, would be:

```
R = 1, G = 0, B = 0
```
On rendering, this color is then multiplied with the color of each pixel of the texture (also called “texel”). The default value for an image color is pure white, which is a 1 on all channels. Thus, the texel color appears unchanged (a multiplication with 1 is a no-op). When you assign a different color instead, the multiplication will yield a new color, e.g.

\[
R = 1, \quad G = 0.8, \quad B = 0.6 \times \\
R = 0.5, \quad G = 0.5, \quad B = 0.5 \\
\text{-------------------------} \\
R = 0.5, \quad G = 0.4, \quad B = 0.3
\]

And here’s the problem: this will only ever make an object darker, never brighter. That’s because we can only multiply with values between 0 and 1; zero meaning the result will be black, and one meaning it remains unchanged.

![Figure 49. Tinting an image with a gray color.](image)

That’s what we want to fix with this filter! We’re going to include an offset to the formula. (In classic Flash, you would do that with a `ColorTransform`.)

- New red value = (old red value \times redMultiplier) + redOffset
- New green value = (old green value \times greenMultiplier) + greenOffset
- New blue value = (old blue value \times blueMultiplier) + blueOffset
- New alpha value = (old alpha value \times alphaMultiplier) + alphaOffset

We already have the multiplier, since that’s handled in the base `Mesh` class; our filter just needs to add the offset.

![Figure 50. Adding an offset to all channels.](image)

So let’s finally start, shall we?!
3.5.2. Extending FragmentFilter

All filters extend the class starling.filters.FragmentFilter, and this one is no exception. Now hold tight: I’m going to give you the complete ColorOffsetFilter class now; this is not a stub, but the final code. We won’t modify it any more.

```java
public class ColorOffsetFilter extends FragmentFilter {
    public function ColorOffsetFilter(redOffset:Number=0, greenOffset:Number=0, blueOffset:Number=0, alphaOffset:Number=0):void {
        colorOffsetEffect.redOffset = redOffset;
        colorOffsetEffect.greenOffset = greenOffset;
        colorOffsetEffect.blueOffset = blueOffset;
        colorOffsetEffect.alphaOffset = alphaOffset;
    }

    override protected function createEffect():FilterEffect {
        return new ColorOffsetEffect();
    }

    private function get colorOffsetEffect():ColorOffsetEffect {
        return effect as ColorOffsetEffect;
    }

    public function get redOffset():Number {
        return colorOffsetEffect.redOffset;
    }

    public function set redOffset(value:Number):void {
        colorOffsetEffect.redOffset = value;
        setRequiresRedraw();
    }

    public function get/set greenOffset():Number;
    public function get/set blueOffset():Number;
    public function get/set alphaOffset():Number;
}
```

That's surprisingly compact, right? Well, I have to admit it: this is just half of the story, because we’re going to have to write another class, too, which does the actual color processing. Still, it’s worthwhile to analyze what we see above.
The class extends `FragmentFilter`, of course, and it overrides one method: `createEffect`. You probably haven’t run into the `starling.rendering.Effect` class before, because it’s really only needed for low-level rendering. From the API documentation:

> An effect encapsulates all steps of a Stage3D draw operation. It configures the render context and sets up shader programs as well as index- and vertex-buffers, thus providing the basic mechanisms of all low-level rendering.

The `FragmentFilter` class makes use of this class, or actually its subclass called `FilterEffect`. For this simple filter, we just have to provide a custom effect, which we’re doing by overriding `createEffect()`. The properties do nothing else than configuring our effect. On rendering, the base class will automatically use the effect to render the filter. That’s it!

If you’re wondering what the `colorOffsetEffect` property does: that’s just a shortcut to be able to access the effect without constantly casting it to `ColorOffsetEffect`. The base class provides an `effect` property, too, but that will return an object of type `FilterEffect` — and we need the full type, `ColorOffsetEffect`, to access our `offset` properties.

More complicated filters might need to override the `process` method as well; that’s e.g. necessary to create multi-pass filters. For our sample filter, though, that’s not necessary.

Finally, note the calls to `setRequiresRedraw`: they make sure the effect is re-rendered whenever the settings change. Otherwise, Starling wouldn’t know that it has to redraw the object.

### 3.5.3. Extending `FilterEffect`

Time to do some actual work, right? Well, our `FilterEffect` subclass is the actual workhorse of this filter. Which doesn’t mean that it’s very complicated, so just bear with me.

Let’s start with a stub:
public class ColorOffsetEffect extends FilterEffect {
    private var _offsets:Vector.<Number>;

    public function ColorOffsetEffect() {
        _offsets = new Vector.<Number>(4, true);
    }

    override protected function createProgram():Program {
        // TODO
    }

    override protected function beforeDraw(context:Context3D):void {
        // TODO
    }

    public function get redOffset():Number { return _offsets[0]; }
    public function set redOffset(value:Number):void { _offsets[0] = value; }

    public function get greenOffset():Number { return _offsets[1]; }
    public function set greenOffset(value:Number):void { _offsets[1] = value; }

    public function get blueOffset():Number { return _offsets[2]; }
    public function set blueOffset(value:Number):void { _offsets[2] = value; }

    public function get alphaOffset():Number { return _offsets[3]; }
    public function set alphaOffset(value:Number):void { _offsets[3] = value; }
}

Note that we’re storing the offsets in a Vector, because that will make it easy to upload them to the GPU. The offset properties read from and write to that vector. Simple enough.

It gets more interesting when we look at the two overridden methods.

createProgram

This method is supposed to create the actual Stage3D shader code.

I’ll show you the basics, but explaining Stage3D thoroughly is beyond the scope of this manual. To get deeper into the topic, you can always have a look at one of the following tutorials:

- How Stage3D works
- Introduction to AGAL
- List of AGAL operations
All Stage3D rendering is done through vertex- and fragment-shaders. Those are little programs that are executed directly by the GPU, and they come in two flavors:

- **Vertex Shaders** are executed *once for each vertex*. Their input is made up from the vertex attributes we typically set up via the `VertexData` class; their output is the position of the vertex in screen coordinates.

- **Fragment Shaders** are executed *once for each pixel* (fragment). Their input is made up of the *interpolated* attributes of the three vertices of their triangle; the output is simply the color of the pixel.

- Together, a fragment and a vertex shader make up a **Program**.

The language filters are written in is called AGAL, an assembly language. (Yes, you read right! This is as low-level as it gets.) Thankfully, however, typical AGAL programs are very short, so it’s not as bad as it sounds.

A quick tip: be sure to always enable `starling.enableErrorChecking` while writing AGAL code. That way, the AGAL compiler will immediately fail when there’s a problem with the code (and print useful debug information), instead of silently ignoring any problems. Trust me: this will save you from hours of frustration and head-scratching.

Just be sure to disable the property again for release builds, as it has a negative impact on performance.

Good news: we only need to write a fragment shader. The vertex shader is the same for most fragment filters, so Starling provides a standard implementation for that. Let’s look at the code:

```javascript
override protected function createProgram():Program {
    var vertexShader:String = STD_VERTEX_SHADER;
    var fragmentShader:String =
        "tex ft0, v0, fs0 <2d, linear> \n" +
        "add oc, ft0, fc0";

    return Program.fromSource(vertexShader, fragmentShader);
}
```

As promised, the vertex shader is taken from a constant; the fragment shader is just two lines of code. Both are combined into one `Program` instance, which is the return value of the method.

The fragment shader requires some further elaboration, of course.

**AGAL in a Nutshell**

In AGAL, each line contains a simple method call.

```
[opcode] [destination], [argument 1], ([argument 2])
```

---

122
• The first three letters are the name of the operation (tex, add).
• The next argument defines where the result of the operation is saved.
• The other arguments are the actual arguments of the method.
• All data is stored in predefined registers; think of them as Vector3D instances (with properties for x, y, z and w).

There are several types of registers, e.g. for constants, temporary data or for the output of a shader. In our shader, some of them already contain data; they were set up by other methods of the filter (we'll come to that later).

• v0 contains the current texture coordinates (varying register 0)
• fs0 points to the input texture (fragment sampler 0)
• fc0 contains the color offset this is all about (fragment constant 0)

The result of a fragment shader must always be a color; that color is to be stored in the oc register.

Code Review

Let's get back to the actual code of our fragment shader. The first line reads the color from the texture:

tex ft0, v0, fs0 <2d, linear>

We're reading the texture fs0 with the texture coordinates read from register v0, and some options (2d, linear). The reason that the texture coordinates are in v0 is just because the standard vertex shader (STD_VERTEX_SHADER) stores them there; just trust me on this one. The result is stored in the temporary register ft0 (remember: in AGAL, the result is always stored in the first argument of an operation).

Now wait a minute. We never created any texture, right? What is this?

As I wrote above, a fragment filter works at the pixel level; its input is the original object, rendered into a texture. Our base class (FilterEffect) sets that up for us; when the program runs, you can be sure that the texture sampler fs0 will point to the pixels of the object being filtered.

You know what, actually I'd like to change this line a little. You probably noticed the options at the end, indicating how the texture data should be interpreted. Well, it turns out that these options depend on the texture type we're accessing. To be sure the code works for every texture, let's use a helper method to write that AGAL operation.

tex("ft0", "v0", 0, this.texture)

That does just the same (the method returns an AGAL string), but we don’t need to care about the options any longer. Always use this method when accessing a texture; it will let you sleep much
better at night.

The second line is doing what we actually came here for: it adds the color offsets to the texel color. The offset is stored in fc0, which we'll look at shortly; that's added to the ft0 register (the texel color we just read) and stored in the output register (oc).

```
add oc, ft0, fc0
```

That's it with AGAL for now. Let's have a look at the other overridden method.

**beforeDraw**

The beforeDraw method is executed directly before the shaders are executed. We can use them to set up all the data required by our shader.

```java
override protected function beforeDraw(context:Context3D):void {
    context.setProgramConstantsFromVector(Context3DProgramType.FRAGMENT, 0, _offsets);
    super.beforeDraw(context);
}
```

This is where we pass the offset values to the fragment shader. The second parameter, 0, defines the register that data is going to end up in. If you look back at the actual shader code, you'll see that we read the offset from fc0, and that's exactly what we're filling up here: fragment constant 0.

The super call sets up all the rest, e.g. it assigns the texture (fs0) and the texture coordinates.

Before you ask: yes, there is also an afterDraw() method, usually used to clean up one's resources. But for constants, this is not necessary, so we can ignore it in this filter.

### 3.5.4. Trying it out

Our filter is ready, actually (download the complete code [here](#))! Time to give it a test ride.

```javascript
var image:Image = new Image(texture);
var filter:ColorOffsetFilter = new ColorOffsetFilter();
filter.redOffset = 0.5;
image.filter = filter;
addChild(image);
```
Our filter seems to have an ugly side effect.

Blimey! Yes, the red value is definitely higher, but why is it now extending beyond the area of the bird!? We didn’t change the alpha value, after all!

Don’t panic. You just created your first filter, and it didn’t blow up on you, right? That must be worth something. It’s to be expected that there’s some fine-tuning to do.

It turns out that we forgot to consider “premultiplied alpha” (PMA). All conventional textures are stored with their RGB channels premultiplied with the alpha value. So, a red with 50% alpha, like this:

\[
R = 1, \quad G = 0, \quad B = 0, \quad A = 0.5
\]

would actually be stored like this:

\[
R = 0.5, \quad G = 0, \quad B = 0, \quad A = 0.5
\]

And we didn’t take that into account. What he have to do is multiply the offset values with the alpha value of the current pixel before adding it to the output. Here’s one way to do that:

```asm
tex("ft0", "v0", 0, texture) // get color from texture
mov ft1, fc0 // copy complete offset to ft1
mul ft1.xyz, fc0.xyz, ft0.www // multiply offset.rgb with alpha (pma!)
add oc, ft0, ft1 // add offset, copy to output
```

As you can see, we can access the \texttt{xyzw} properties of the registers to access individual color channels (they correspond with our \texttt{rgba} channels).

What if the texture is not stored with PMA? The \texttt{tex} method makes sure that we always receive the value with PMA, so no need to worry about that.

Second Try

When you give the filter another try now (complete code: \texttt{ColorOffsetFilter.as}), you’ll see correct alpha values:
Congratulations! You just created your first filter, and it works flawlessly. (Yes, you could have just used Starling's `ColorMatrixFilter` instead — but hey, this one is a tiny little bit faster, so it was well worth the effort.)

If you're feeling brave, you could now try to achieve the same with a mesh style instead. It's not *that* different, promised!

### 3.6. Custom Styles

Now that we have tapped the raw power of `Stage3D`, let's continue on this road! In this section, we will write a simple mesh style. In Starling 2, all rendering is done through styles; by creating your own style, you can create special effects without sacrificing performance in any way.

Before you continue, please make sure you have read through the section [Custom Filters](#), as well. Filters and styles share many concepts, so it makes sense to start with the simpler of the two. Below, I'll assume that you are familiar with everything that's shown in that other section.

#### 3.6.1. The Goal

The goal is just the same as the one we were shooting for with the `ColorOffsetFilter`; we want to allow adding an offset to the color value of every rendered pixel. Only this time, we're doing it with style! We'll call it ... `ColorOffsetStyle`.

![Figure 53. Applying a color offset with style.](image)

Before we continue, it's crucial that you understand the difference between a filter and a style.
Filters vs. Styles

As mentioned before, a filter works on a per-pixel-level: the object is rendered into a texture, and the filter processes that texture in some way. A style, on the other hand, has access to all the original geometry of the object, or to be more precise: to the object’s vertices.

While that limits styles in some ways (e.g. you can’t achieve a blur effect with a style), it makes them much more efficient. First, because you don’t need that first step of drawing the object into a texture. Second, and most importantly: this allows styled meshes to be batched.

As you know, keeping the number of draw calls down is very important for a high frame rate. To make sure that happens, Starling batches as many objects together as possible before drawing them. Question is, how to decide which objects may be batched together? This is where the style comes into play: only objects with the same style can be batched together.

If you add three images to the stage that have a `ColorOffsetFilter` applied to them, you’ll see at least three draw calls. Add three objects with a `ColorOffsetStyle` instead, and you’ll have just one. That makes styles a little more difficult to write — but that’s also what makes it worth the effort!

3.6.2. Extending MeshStyle

The base class for all styles is `starling.styles.MeshStyle`. This class provides all the infrastructure we need. Let’s look at a stub first:

```javascript
public class ColorOffsetStyle extends MeshStyle {

    public static const VERTEX_FORMAT:VertexDataFormat =
        MeshStyle.VERTEX_FORMAT.extend("offset:float4");

    private var _offsets:Vector.<Number>;

    public function ColorOffsetStyle(
        redOffset:Number=0, greenOffset:Number=0,
        blueOffset:Number=0, alphaOffset:Number=0):void
    {
        _offsets = new Vector.<Number>(4, true);
        setTo(redOffset, greenOffset, blueOffset, alphaOffset);
    }

    public function setTo(
        redOffset:Number=0, greenOffset:Number=0,
        blueOffset:Number=0, alphaOffset:Number=0):void
    {
        _offsets[0] = redOffset;
        _offsets[1] = greenOffset;
        _offsets[2] = blueOffset;
        _offsets[3] = alphaOffset;

        updateVertices();
    }
}
```
override public function copyFrom(meshStyle:MeshStyle):void
{
    // TODO
}

override public function createEffect():MeshEffect
{
    return new ColorOffsetEffect();
}

override protected function onTargetAssigned(target:Mesh):void
{
    updateVertices();
}

override public function get vertexFormat():VertexDataFormat
{
    return VERTEX_FORMAT;
}

private function updateVertices():void
{
    // TODO
}

public function get redOffset():Number { return _offsets[0]; }
public function set redOffset(value:Number):void
{
    _offsets[0] = value;
    updateVertices();
}

// the other offset properties need to be implemented accordingly.

public function get/set greenOffset():Number;
public function get/set blueOffset():Number;
public function get/set alphaOffset():Number;
}

That's our starting point. You'll see that there's already going on a little more than in our initial filter class from the last example. So let's have a look at the individual parts of that code.

**Vertex Formats**

The first thing that's notable is the vertex format constant at the very top of the class. I mentioned already that styles work on a vertex level, giving you access to all the geometry of an object. The VertexData class stores that geometry, but we never actually discussed how that class knows which data is stored in this class, and how. That's defined by the VertexDataFormat.
The default format used by MeshStyle is the following:

```
position:float2, texCoords:float2, color:bytes4
```

The syntax of this string should seem familiar; it’s a list of attributes with certain data types.

- The **position** attribute stores two floats (for the x- and y-coordinates of a vertex).
- The **texCoords** attribute stores two floats, as well (for the texture coordinates of the vertex).
- The **color** attribute stores four bytes for the color of the vertex (one byte for each channel).

A VertexData instance with this format will store those attributes for each vertex of the mesh, using the exact same order as in the format string. This means that each vertex will take up 20 bytes (8 + 8 + 4).

When you create a mesh and don’t assign any style in particular, it will be rendered by the standard **MeshStyle**, forcing exactly this format onto its vertices. That’s all the information you need to draw a textured, colored mesh, after all.

But for our **ColorOffsetStyle**, that’s not enough: we need to store our color offset as well. Thus, we need to define a new format that adds an **offset** attribute consisting of four float values.

```javascript
MeshStyle.VERTEX_FORMAT.extend("offset:float4");
// => position:float2, texCoords:float2, color:bytes4, offset:float4
```

Now, you may ask: **Why do we need this?** The filter worked just fine without a custom vertex format, after all.

That’s a very good question, I’m glad you ask! The answer lies in Starling’s batching code. When we assign our style to some subsequent meshes, they will be batched together — that’s the whole reason we make this effort, right?

But what does batching mean? It just means that we’re copying the vertices of all individual meshes to one bigger mesh and render that. Somewhere inside Starling’s rendering internals, you’ll find code that will look similar to this:

```javascript
var batch:Mesh = new Mesh();
batch.add(meshA);
batch.add(meshB);
batch.add(meshC);

batch.style = meshA.style; // ← !!!
batch.render();
```

Do you see the problem? The big mesh (**batch**) receives a copy of the style of the mesh that was first added. Those three styles will probably use different settings, though. If those settings are just
stored in the style, all but one will be lost on rendering. Instead, the style must store its data in the VertexData of its target mesh! Only then will the big batch mesh receive all the offsets individually.

⚠️ Since it’s so important, I’ll rephrase that: A style’s settings must always be stored in the target mesh’s vertex data.

Per convention, the vertex format is always accessible as a static constant in the style’s class, and also returned by the vertexFormat property. When the style is assigned to a mesh, its vertices will automatically be adapted to that new format.

When you have understood that concept, you’re already halfway through all of this. The rest is just updating the code so that the offset is read from the vertex data instead of fragment constants.

But I’m getting ahead of myself.

**Member Variables**

You’ll note that even though I just insisted that all data is to be stored in the vertices, there’s still a set of offsets stored in a member variable:

```javascript
private var _offsets:Vector.<Number>;
```

That’s because we want developers to be able to configure the style before it’s assigned to a mesh. Without a target object, there’s no vertex data we could store these offsets in, right? So we’ll use this vector instead. As soon as a target is assigned, the values are copied over to the target’s vertex data (see onTargetAssigned).

**copyFrom**

During batching, styles sometimes have to be copied from one instance to another (mainly to be able to re-use them without annoying the garbage collector). Thus, it’s necessary to override the method `copyFrom`. We’ll do that like this:

```javascript
override public function copyFrom(meshStyle:MeshStyle):void
{
    var colorOffsetStyle:ColorOffsetStyle = meshStyle as ColorOffsetStyle;
    if (colorOffsetStyle)
    {
        for (var i:int=0; i<4; ++i)
        _offsets[i] = colorOffsetStyle._offsets[i];
    }
    super.copyFrom(meshStyle);
}
```

This is rather straight-forward; we just check if the style we’re copying from has the correct type and then duplicate all of its offsets on the current instance. The rest is done by the super-class.
createEffect

This looks very familiar, right?

```javascript
override public function createEffect():MeshEffect {
    return new ColorOffsetEffect();
}
```

It works just like in the filter class; we return the `ColorOffsetEffect` we're going to create later. No, it's not the same as the one used in the filter (since the offset values are read from the vertices), but it would be possible to create an effect that works for both.

onTargetAssigned

As mentioned above, we need to store our offsets in the vertex data of the target mesh. Yes, that means that each offset is stored on all vertices, even though this might seem wasteful. It's the only way to guarantee that the style supports batching.

When the filter is assigned a target, this callback will be executed—that is our cue to update the vertices. We're going to do that again elsewhere, so I moved the actual process into the `updateVertices` method.

```javascript
override protected function onTargetAssigned(target:Mesh):void {
    updateVertices();
}

private function updateVertices():void {
    if (target) {
        var numVertices:int = vertexData.numVertices;
        for (var i:int=0; i<numVertices; ++i) {
            vertexData.setPoint4D(i, "offset", _offsets[0], _offsets[1], _offsets[2], _offsets[3]);
            setRequiresRedraw();
        }
    }
}
```

You might wonder where that `vertexData` object comes from. As soon as the target is assigned, the `vertexData` property will reference the target's vertices (the style itself never owns any vertices). So the code above loops through all vertices of the target mesh and assigns the correct offset values, ready to be used during rendering.
3.6.3. Extending MeshEffect

We’re done with the style class now—time to move on to the effect, which is where the actual rendering takes place. This time, we’re going to extend the `MeshEffect` class. Remember, effects simplify writing of low-level rendering code. I’m actually talking about a group of classes with the following inheritance:

The base class (`Effect`) does only the absolute minimum: it draws white triangles. The `FilterEffect` adds support for textures, and the `MeshEffect` for color and alpha.

Those two classes could also have been named `TexturedEffect` and `ColoredTexturedEffect`, but I chose to baptize them with their usage in mind. If you create a filter, you need to extend `FilterEffect`; if you create a mesh style, `MeshEffect`.

So let’s look at the setup of our `ColorOffsetEffect`, with a few stubs we’re filling in later.
class ColorOffsetEffect extends MeshEffect {

    public static const VERTEX_FORMAT: VertexDataFormat = ColorOffsetStyle.VERTEX_FORMAT;

    public function ColorOffsetEffect() {}

    override protected function createProgram(): Program {
        // TODO
    }

    override public function get vertexFormat(): VertexDataFormat {
        return VERTEX_FORMAT;
    }

    override protected function beforeDraw(context: Context3D): void {
        super.beforeDraw(context);
        vertexFormat.setVertexBufferAt(3, vertexBuffer, "offset");
    }

    override protected function afterDraw(context: Context3D): void {
        context.setVertexBufferAt(3, null);
        super.afterDraw(context);
    }
}

If you compare that with the analog filter effect from the previous tutorial, you’ll see that all the offset properties were removed; instead, we’re now overriding vertexFormat, which ensures that we are using the same format as the corresponding style, ready to have our offset values stored with each vertex.

beforeDraw & afterDraw

The beforeDraw and afterDraw methods now configure the context so that we can read the offset attribute from the shaders as va3 (vertex attribute 3). Let’s have a look at that line from beforeDraw:

vertexFormat.setVertexBufferAt(3, vertexBuffer, "offset");

That’s equivalent to the following:

context.setVertexBufferAt(3, vertexBuffer, 5, "float4");
That third parameter (\(5 \rightarrow \text{bufferOffset}\)) indicates the position of the color offset inside the vertex format, and the last one (\(\text{float4} \rightarrow \text{format}\)) the format of that attribute. So that we don’t have to calculate and remember those values, we can ask the \text{vertexFormat} object to set that attribute for us. That way, the code will continue to work if the format changes and we add, say, another attribute before the offset.

Vertex buffer attributes should always be cleared when drawing is finished, because following draw calls probably use a different format. That’s what we’re doing in the \text{afterDraw} method.

\textbf{createProgram}

It’s finally time to tackle the core of the style; the AGAL code that does the actual rendering. This time, we have to implement the vertex-shader as well; it won’t do to use a standard implementation, because we need to add some custom logic. The fragment shader, however, is almost identical to the one we wrote for the filter. Let’s take a look!

```agal
override protected function createProgram():Program {
    var vertexShader:String = [
        "m44 op, va0, vc0", // 4x4 matrix transform to output clip-space
        "mov v0, va1", // pass texture coordinates to fragment program
        "mul v1, va2, vc4", // multiply alpha (vc4) with color (va2), pass to fp
        "mov v2, va3" // pass offset to fp
    ].join("\n");

    var fragmentShader:String = [
        tex("ft0", "v0", 0, texture) + // get color from texture
        "mul ft0, ft0, v1", // multiply color with texel color
        "mov ft1, v2", // copy complete offset to ft1
        "mul ft1.xyz, v2.xyz, ft0.www", // multiply offset.rgb with alpha (pma!)
        "add oc, ft0, ft1" // add offset, copy to output
    ].join("\n");

    return Program.fromSource(vertexShader, fragmentShader);
}
```

To understand what the vertex-shader is doing, you first have to understand the input it’s working with.

- The \(\text{va}\) registers ("vertex attribute") contain the attributes from the current vertex, taken from the vertex buffer. They are ordered just like the attributes in the vertex format we set up a little earlier: \(\text{va0}\) is the vertex position, \(\text{va1}\) the texture coordinates, \(\text{va2}\) the color, \(\text{va3}\) the offset.

- Two constants are the same for all our vertices: \(\text{vc0-3}\) contain the modelview-projection matrix, \(\text{vc4}\) the current alpha value.

The main task of any vertex shader is to move the vertex position into the so-called "clip-space". That’s done by multiplying the vertex position with the \text{mvpMatrix} (modelview-projection matrix). The first line takes care of that, and you’ll find it in any vertex shader in Starling. Suffice it to say that it is responsible for figuring out where the vertex is ending up on the screen.
Otherwise, we're more or less just forwarding data to the fragment shader via the "varying registers" $v0 - v2$.

The fragment shader is an almost exact replica of its filter-class equivalent. Can you find the difference? It's the register we're reading the offset from; before, that was stored in a constant, now in $v2$.

### 3.6.4. Trying it out

There you have it: we're almost finished with our style! Let's give it a test-ride. In a truly bold move, I'll use it on two objects right away, so that we'll see if batching works correctly.

```javascript
var image: Image = new Image(texture);
var style: ColorOffsetStyle = new ColorOffsetStyle();
style.redOffset = 0.5;
image.style = style;
addChild(image);

var image2: Image = new Image(texture);
image2.x = image.width;
var style2: ColorOffsetStyle = new ColorOffsetStyle();
style2.blueOffset = 0.5;
image2.style = style2;
addChild(image2);
```

*Figure 54. Two styled images, rendered with just one draw call.*

Hooray, this actually works! Be sure to look at the draw count at the top left, which is an honest and constant "1".

There's a tiny little bit more to do, though; our shaders above were created assuming that there's always a texture to read data from. However, the style might also be assigned to a mesh that doesn't use any texture, so we have to write some specific code for that case (which is so simple I'm not going to elaborate on it right now).

The complete class, including this last-minute fix, can be found here: [ColorOffsetStyle.as](#).

### 3.6.5. Where to go from here

That's it with our style! I hope you're as thrilled as I am that we succeeded on our task. What you
see above is the key to extending Starling in ways that are limited only by your imagination. The MeshStyle class even has a few more tricks up its sleeve, so be sure to read through the complete class documentation.

I'm looking forward to seeing what you come up with!

### 3.7. Distance Field Rendering

As mentioned multiple times, bitmap fonts are the fastest way to render text in Starling. However, if you need to display text in multiple sizes, you will soon discover that bitmap fonts do not scale well. Scaling up makes them blurry, scaling down introduces aliasing problems. Thus, for best results, one has to embed the font in all the sizes used within the application.

Distance Field Rendering solves this issue: it allows bitmap fonts and other single-color shapes to be drawn without jagged edges, even at high magnifications. The technique was first introduced in a SIGGRAPH paper by Valve Software. To support this feature, Starling contains a special MeshStyle implementation.

To understand how it works, I will start by showing you how to use it on a single image. This could e.g. be an icon you want to use throughout your application.

#### 3.7.1. Rendering a single Image

We had plenty of birds in this manual already, so let’s go for a predator this time! My cat qualifies for the job. I’ve got her portrait as a black vector outline, which is perfect for this use-case.

![Figure 55. Say hello to “Seven of Nine”, my cat!](image)

Unfortunately, Starling can’t display vector images; we need Seven as a bitmap texture (PNG format). That works great as long as we want to display the cat in roughly the original size (scale == 1). However, when we enlarge the image, it quickly becomes blurry.
This is exactly what we can avoid by converting this image into a distance field texture. Starling actually contains a handy little tool that takes care of this conversion process. It’s called the "Field Agent" and can be found in the `util` directory of the Starling repository.

You need both Ruby and ImageMagick installed to use the field agent. Look at the accompanying README file to find out how to install those dependencies. The tool works both on Windows and macOS.

I started with a high-resolution PNG version of the cat and passed that to the field agent.

```
ruby field_agent.rb cat.png cat-df.png --scale 0.25 --auto-size
```

This will create a distance field texture with 25% of the original size. The field agent works best if you pass it a high resolution texture and let it scale that down. The distance field encodes the details of the shape, so it can be much smaller than the input texture.

The original, sharp outline of the cat has been replaced with a blurry gradient. That’s what a distance field is about: in each pixel, it encodes the distance to the closest edge in the original shape.

This texture is actually just pure white on a transparent background; I colored the background black just so you can see the result better.

The amount of blurriness is called spread. The field agent uses a default of eight pixels, but you can customize that. A higher spread allows better scaling and makes it easier to add special effects (more on those later), but its possible range depends on the input image. If the input contains very thin lines, there’s simply not enough room for a high spread.

To display this texture in Starling, we simply load the texture and assign it to an image. Assigning the `DistanceFieldStyle` will make Starling switch to distance field rendering.
With this style applied, the texture stays perfectly crisp even with high scale values. You only see small artifacts around very fine-grained areas (like Seven’s haircut).

![Figure 58. Scaling up a distance field texture.](image)

Depending on the "spread" you used when creating the texture, you might need to update the softness parameter to get the sharpness / smoothness you’d like to have. That’s the first parameter of the style’s constructor.

💡 Rule of thumb: softness = 1.0 / spread.

**Render Modes**

That’s actually just the most basic use of distance field textures. The distance field style supports a couple of different render modes; namely an outline, a drop shadow, and a glow. Those effects are all rendered in a specific fragment shader, which means that they do not require any additional draw calls. In other words, these effects are basically coming for free, performance wise!
The only limitation: you cannot combine two modes, e.g. to have both outline and drop shadow. You can still resort back to fragment filters for that, though.

### 3.7.2. Rendering Text

The characteristics of distance field rendering make it perfect for text. Good news: Starling’s standard bitmap font class works really well with the distance field style. By now, there are even great tools for creating suitable font textures.

Remember, a bitmap font consists of an atlas-texture that contains all the glyphs and an XML file describing the attributes of each glyph. You can’t simply use `field agent` to convert the texture in a post-processing step (at least not easily), since each glyph requires some padding around it to make up for the *spread*. Therefore, it’s best to use a bitmap font tool that supports distance field textures natively.

Some of the GUI-tools I described in the Bitmap Fonts chapter can be configured to create distance field output. However, to be blunt, none of them does that job particularly well. If you are not afraid of some command line magic, I recommend another one instead.

**msdf-bmfont-xml**

*Msdf-bmfont-xml* runs on both Windows and macOS (refer to its [GitHub page](https://github.com/) for installation instructions). The quality of the output is phenomenal, and it even supports an advanced type of distance field texture — we’ll look at that shortly.

For now, let’s start with the basics: let’s create a distance field font for, say, *Arial.ttf*:

```
msdf-bmfont -o arial.png -t sdf --smart-size Arial.ttf
```

- `-o arial.png` determines the filename for the font texture.
- `-t sdf` indicates that we want to create a *single-channel* distance field font. All distance field textures we discussed so far were single channel textures.
- `--smart-size` will make sure the texture uses the smallest possible size.
This command will create the familiar pair of \texttt{png} and \texttt{fnt} files that you can load in your app. What's more, it adds an additional data element to the \texttt{fnt} file that contains all relevant distance field settings. As a result, Starling is able to set everything up for you automatically.

That's right, there is no special set-up code to write — you create font and TextField just like you always do! Behind the scenes, Starling will automatically set up a \texttt{DistanceFieldStyle} instance with the correct settings.

**Multi-Channel Distance Field Textures**

A \textit{multi-channel} distance field texture uses all channels of an RGB-texture to encode the distance information. This leads to an even higher rendering quality.

As mentioned above, \texttt{msdf-bmfont-xml} supports this distance field type — in fact, it even defaults to it. Please run the command-line tool again, but omit the \texttt{-t sdf} parameter.

```
msdf-bmfont -o arial.png --smart-size Arial.ttf
```

This time, you will end up with a \textit{multi-channel} distance field texture. Here's what it looks like:

![Figure 60. A multi-channel distance field font.](image)

Again, Starling will recognize the texture and will render in "multi-channel" style automatically.
The reward for all this hard work: such a font can now be used at almost any scale, and with all the flexible render modes I showed above.

Figure 61. A bitmap font using distance field fonts looks great at any scale.

3.8. Summary

Pat yourself on the back: we just covered a lot of quite advanced topics.

- You are now familiar with ATF textures, which are not only very memory efficient, but load faster than standard PNGs.
- You know how to recover from a context loss: by relying on the `AssetManager` or providing your own restoration-code.
- You’ve got a feeling about how to make sure you are not wasting any memory, and how to avoid and find memory leaks.
- When performance becomes an issue, your first look is at the draw count. You know ways to make sure batching is not disrupted.
- The low-level rendering code is far less frightening that you thought at first. Heck, you even wrote your own filter and style!
- Distance field rendering is a useful technique to keep in mind for scalable fonts or other monochrome shapes.

This knowledge will save you a lot of time and trouble in any of your upcoming projects. And I bet some of them are going to have to run on mobile hardware, right ...?
Chapter 4. Mobile Development

Adobe AIR is one of the most powerful solutions available today when it comes to cross-platform development. And when somebody says "cross-platform" nowadays, it typically means: iOS and Android.

Developing for such mobile platforms can be extremely challenging—there's a plethora of different device types around, featuring screen resolutions that range from insult to the eye to insanely high and aspect ratios that defy any logic. To add insult to injury, some of them are equipped with CPUs that clearly were never meant to power anything else than a pocket calculator.

As a developer, you can only shrug your shoulders, roll up your sleeves, and jump right in. At least you know that fame and fortune lie on the other side of the journey! [2: Don't pin me down to that, though.]

4.1. Multi-Resolution Development

Oh, where are the days we developed games for a single screen? Back then, we had a small rectangular area in an HTML page, and that's where we placed our sprites, texts and images. A single resolution to work for — wasn't that nice?

Alas — the times they are a-changin'! Mobile phones come in all flavors and sizes, and even desktop computers and notebooks feature high density displays. This is great news for our consumer-selves, but it doesn't exactly make our lives as developers easier, that's for sure!

But don't give up hope: you can manage that. It's just a matter of thinking ahead and making use of a few simple mechanisms provided by Starling.

The problem is just that it's a little overwhelming at first. That's why we'll do this in small steps — and we will begin in 2007.

Yes, you heard right: step into the DeLorean, start up the Flux Capacitor™ and hold tight while we hit those eighty miles per hour.

4.1.1. iPhone

The iPhone is arguably the most popular platform for casual games. Back in 2007, it was also the only one you could easily develop for. This was the time of the big App Store gold rush!

With its fixed resolution of 320×480 pixels, the first iPhone was super easy to develop for. Granted, Starling wasn’t around back then, but you would have started it up like this:

```javascript
var screenWidth:int = stage.fullScreenWidth;
var screenHeight:int = stage.fullScreenHeight;
var viewPort:Rectangle = new Rectangle(0, 0, screenWidth, screenHeight);

starling = new Starling(Game, stage, viewPort);
```
We set the `viewPort` to the full size of the screen: 320×480 pixels. Per default, the stage will have exactly the same dimensions.
So far, so easy: this works just like in, say, a game for the browser. (That would be Internet Explorer 6, then, right?)

Next stop: 2010.

### 4.1.2. iPhone Retina

We park our DeLorean right around the corner of the old Apple campus and check out the App Store charts. Hurray! Apparently, our game was a huge success in 2007, and it's still in the top 10! There's no time to lose: we must make sure it looks well on the iPhone 4 that's going to come out in a few weeks.

Since we're coming from the future, we know about its major innovation, the high-resolution screen dubbed "Retina Display" by the Apple marketing team. To support this resolution in Adobe AIR, we need to activate the following setting in the application descriptor (the XML file that configures your AIR app):

```xml
<iPhone>
  <!-- Display Resolution ("standard" or "high"). Default "standard". -->
  <requestedDisplayResolution>high</requestedDisplayResolution>
</iPhone>
```

With that out of the way, we fire up our game from 2007 and start it up on this yet-to-be released device.
Damn, the game is now only taking up a quarter of the screen! Why is that?

If you look back at the code we wrote in 2007, you’ll see that we made the `viewport` just as big as the screen. With the iPhone 4, these values have doubled: its screen has 640×960 pixels. The code that placed display objects on the stage expected a coordinate system of just 320×480, though. So things that were placed on the very right (x=320) are now suddenly at the center instead.

That’s easily solved, though. Remember: Starling’s `viewport` and `stageWidth/Height` properties can be set independently.

- The `viewport` decides into which area of the screen Starling renders into. It is always specified in pixels.
- The stage size decides the size of the coordinate system that is displayed in that `viewport`. When your stage width is 320, any object with an x-coordinate between 0 and 320 will be within the stage, no matter the size of the `viewport`.

With that knowledge, upscaling is trivial:

```javascript
var screenWidth:int  = stage.fullScreenWidth;
var screenHeight:int = stage.fullScreenHeight;
var viewport:Rectangle = new Rectangle(0, 0, screenWidth, screenHeight);

starling = new Starling(Game, stage, viewport);
starling.stage.stageWidth  = 320;
starling.stage.stageHeight = 480;
```

The `viewport` is still dynamic, depending on the device the game is started on; but we added two lines at the bottom that hard-code the stage size to fixed values.

Since those values no longer indicate pixels, we are now calling them `points`: our stage size is now 320×480 points.

On the iPhone 4, the game now looks like this:
That's better: we are now using the full screen size. However, it's also a little blurry. We're not really making any use of the big screen. I can already see the bad reviews coming in ... we need to fix this!

**HD textures**

The solution for that problem is to provide special textures for the high resolution. Depending on the pixel density, we will use either the low- or high-resolution texture set. The advantage: except for the logic that picks the textures, we don't need to change any of our code.

It's not enough to simply load a different set of files, though. After all, bigger textures will return bigger values for *width* and *height*. With our fixed stage width of 320 points,

- an SD texture with a width of 160 pixels will fill half of the stage;
- a corresponding HD texture (width: 320 pixels) would fill the complete stage.

What we want instead is for the HD texture to report the same size as the SD texture, but provide more detail.

That's where Starling's *contentScaleFactor* comes in handy. We implicitly set it up when we configured Starling's *stage* and *viewport* sizes. With the setup shown above, run the following code on an iPhone 4:

```javascript
trace(starling.contentScaleFactor); // → 2
```

The *contentScaleFactor* returns the *viewport width* divided by the *stage width*. On a retina device, it will be "2"; on a non-retina device, it will be "1". This tells us which textures to load at runtime.

It's not a coincidence that the *contentScaleFactor* is a whole number. Apple exactly doubled the number of pixels per row / per column to avoid aliasing issues as much as possible.

The texture class has a similar property simply called *scale*. When set up correctly, the texture will work just like we want it to.

```javascript
var scale: Number = starling.contentScaleFactor; ①
var texturePath: String = "textures/" + scale + "x"; ②
var appDir: File = File.applicationDirectory;

assetManager.textureOptions.scale = scale; ③
assetManager.enqueue(appDir.resolvePath(texturePath));
assetManager.loadQueue(...);

var texture: Texture = assetManager.getTexture("penguin"); ④
trace(texture.scale); // → Either '1' or '2' ⑤
```
① Get the `contentScaleFactor` from the Starling instance.

② Depending on the scale factor, the textures will be loaded from the directory 1x or 2x.

③ By assigning the same scale factor to the `AssetManager`, all textures will be initialized with that value.

④ When accessing the textures, you don’t need to take care about the scale factor.

⑤ However, you can find out the scale of a texture anytime via the `scale` property.

> Not using the `AssetManager`? Don’t worry: all the `Texture.from...` methods contain an extra argument for the scale factor. It must be configured right when you create the texture; the value can’t be changed later.

The textures will now take the `scale factor` into account when you query their width or height. For example, here’s what will happen with the game’s full-screen background texture.

<table>
<thead>
<tr>
<th>File</th>
<th>Size in Pixels</th>
<th>Scale Factor</th>
<th>Size in Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>textures/1x/bg.jpg</td>
<td>320×480</td>
<td>1.0</td>
<td>320×480</td>
</tr>
<tr>
<td>textures/2x/bg.jpg</td>
<td>640×960</td>
<td>2.0</td>
<td>320×480</td>
</tr>
</tbody>
</table>

Now we have all the tools we need!

- Our graphic designer on the back seat (call him Biff) creates all textures in a high resolution (ideally, as vector graphics).

- In a preprocessing step, the textures are converted into the actual resolutions we want to support (1x, 2x).

- At runtime, we check Starling’s `contentScaleFactor` and load the textures accordingly.

This is it: now we’ve got a crisp-looking retina game! Our player’s will appreciate it, I’m sure of that.
Figure 65. Now we’re making use of the retina screen!

Tools like TexturePacker make this process really easy. Feed them with all your individual textures (in the highest resolution) and let them create multiple texture atlases, one for each scale factor.

We celebrate our success at a bar in Redwood, drink a beer or two, and move on.

4.1.3. iPhone 5

In 2012, the iPhone has another surprise in store for us: Apple changed the screen’s aspect ratio. Horizontally, it’s still 640 pixels wide; but vertically, it’s now a little bit longer (1136 pixels). It’s still a retina display, of course, so our new logical resolution is 320×568 points.

As a quick fix, we simply center our stage on the viewPort and live with the black bars at the top and bottom.

```javascript
var offsetY: int = (1136 - 960) / 2;
var viewport: Rectangle = new Rectangle(0, offsetY, 640, 960);
```

Mhm, that seems to work! It’s even a fair strategy for all those Android smartphones that are beginning to pop up in this time line. Yes, our game might look a little blurry on some devices, but it’s not too bad: the image quality is still surprisingly good. Most users won’t notice.
I call this the **Letterbox Strategy**.

- Develop your game with a fixed stage size (like 320×480 points).
- Add several sets of assets, depending on the scale factor (e.g. 1x, 2x, 3x).
- Then you scale up the application so that it fills the screen without any distortion.

This is probably the most pragmatic solution. It allows your game to run in an acceptable quality on all available display resolutions, and you don’t have to do any extra work other than setting the viewPort to the right size.

By the way, the latter is very easy when you use the `RectangleUtil` that comes with Starling. To "zoom" your viewPort up, just create it with the following code:

```javascript
const stageWidth: int = 320; // points
const stageHeight: int = 480;
const screenWidth: int = stage.fullScreenWidth; // pixels
const screenHeight: int = stage.fullScreenHeight;

var viewPort: Rectangle = RectangleUtil.fit(
    new Rectangle(0, 0, stageWidth, stageHeight),
    new Rectangle(0, 0, screenWidth, screenHeight),
    ScaleMode.SHOW_ALL);
```

Simple, yet effective! We definitely earned ourselves another trip with the time machine. Hop in!

### 4.1.4. iPhone 6 and Android

We’re in 2014 now and … Great Scott! Checking out the “App Store Almanac”, we find out that our sales haven’t been great after our last update. Apparently, Apple wasn’t too happy with our letterbox-approach and didn’t feature us this time. Damn.

Well, I guess we have no other choice now: let’s bite the bullet and make use of that additional screen space. So long, hard-coded coordinates! From now on, we need to use relative positions for all our display objects.

I will call this strategy **Smart Object Placement**. The startup-code is still quite similar:

```javascript
var viewPort: Rectangle = new Rectangle(0, 0, screenWidth, screenHeight);

starling = new Starling(Game, stage, viewPort);
starling.stage.stageWidth = 320;
starling.stage.stageHeight = isIPhone5() ? 568 : 480;
```

Yeah, I smell it too. Hard coding the stage height depending on the device we’re running … that’s not a very smart idea. Promised, we’re going to fix that soon.
For now, it works, though: both viewPort and stage have the right size. But how do we make use of that? Let’s look at the Game class now, the class acting as our Starling root.

```java
public class Game extends Sprite {
    public function Game() {
        addEventListener(Event.ADDED_TO_STAGE, onAddedToStage); ①
    }

    private function onAddedToStage():void {
        setup(stage.stageWidth, stage.stageHeight); ②
    }

    private function setup(width:Number, height:Number):void {
        // ...

        var lifeBar:LifeBar = new LifeBar(width); ③
        lifeBar.y = height - lifeBar.height;
        addChild(lifeBar);

        // ...
    }
}
```

① When the constructor of game is called, it’s not yet connected to the stage. So we postpone initialization until we are.

② We call our custom setup method and pass the stage size along.

③ Exemplary, we create a LifeBar instance (a custom user interface class) at the bottom of the screen.

All in all, that wasn’t too hard, right? The trick is to always take the stage size into account. Here, it pays off if you created your game in clean components, with separate classes responsible for different interface elements. For any element where it makes sense, you pass the size along (like in the LifeBar constructor above) and let it act accordingly.
That works really well on the iPhone 5. We should have done that in 2012, dammit! Here, in 2014, things have become even more complicated.

- Android is quickly gaining market share, with phones in all different sizes and resolutions.
- Even Apple introduced bigger screens with the iPhone 6 and iPhone 6 Plus.
- Did I mention tablet computers?

By organizing our display objects relative to the stage dimensions, we already laid the foundations to solve this. Our game will run with almost any stage size.

The remaining problem is which values to use for stage size and content scale factor. Looking at the range of screens we have to deal with, this seems like a daunting task!

<table>
<thead>
<tr>
<th>Device</th>
<th>Screen Size</th>
<th>Screen Density</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone 3</td>
<td>3,50&quot;</td>
<td>163 dpi</td>
<td>320×480</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>3,50&quot;</td>
<td>326 dpi</td>
<td>640×960</td>
</tr>
<tr>
<td>iPhone 5</td>
<td>4,00&quot;</td>
<td>326 dpi</td>
<td>640×1136</td>
</tr>
<tr>
<td>iPhone 6</td>
<td>4,70&quot;</td>
<td>326 dpi</td>
<td>750×1334</td>
</tr>
<tr>
<td>iPhone 6 Plus</td>
<td>5,50&quot;</td>
<td>401 dpi</td>
<td>1080×1920</td>
</tr>
<tr>
<td>Galaxy S1</td>
<td>4,00&quot;</td>
<td>233 dpi</td>
<td>480×800</td>
</tr>
<tr>
<td>Galaxy S3</td>
<td>4,80&quot;</td>
<td>306 dpi</td>
<td>720×1280</td>
</tr>
<tr>
<td>Galaxy S5</td>
<td>5,10&quot;</td>
<td>432 dpi</td>
<td>1080×1920</td>
</tr>
<tr>
<td>Galaxy S7</td>
<td>5,10&quot;</td>
<td>577 dpi</td>
<td>1440×2560</td>
</tr>
</tbody>
</table>

The key to figuring out the scale factor is to take the screen's density into account.

- The higher the density, the higher the scale factor. In other words: we can infer the scale factor from the density.
- From the scale factor, we can calculate the appropriate stage size. Basically, we reverse our previous approach.

The original iPhone had a screen density of about 160 dpi. We take that as the basis for our calculations: for any device, we divide the density by 160 and round the result to the next integer. Let's make a sanity check of that approach.

<table>
<thead>
<tr>
<th>Device</th>
<th>Screen Size</th>
<th>Screen Density</th>
<th>Scale Factor</th>
<th>Stage Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone 3</td>
<td>3,50&quot;</td>
<td>163 dpi</td>
<td>1.0</td>
<td>320×480</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>3,50&quot;</td>
<td>326 dpi</td>
<td>2.0</td>
<td>320×480</td>
</tr>
<tr>
<td>iPhone 5</td>
<td>4,00&quot;</td>
<td>326 dpi</td>
<td>2.0</td>
<td>320×568</td>
</tr>
<tr>
<td>Device</td>
<td>Screen Size</td>
<td>Screen Density</td>
<td>Scale Factor</td>
<td>Stage Size</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>iPhone 6</td>
<td>4.70&quot;</td>
<td>326 dpi</td>
<td>2.0</td>
<td>375×667</td>
</tr>
<tr>
<td>iPhone 6 Plus</td>
<td>5.50&quot;</td>
<td>401 dpi</td>
<td>3.0</td>
<td>414×736</td>
</tr>
<tr>
<td>Galaxy S1</td>
<td>4.00&quot;</td>
<td>233 dpi</td>
<td>1.5</td>
<td>320×533</td>
</tr>
<tr>
<td>Galaxy S3</td>
<td>4.80&quot;</td>
<td>306 dpi</td>
<td>2.0</td>
<td>360×640</td>
</tr>
<tr>
<td>Galaxy S5</td>
<td>5.10&quot;</td>
<td>432 dpi</td>
<td>3.0</td>
<td>360×640</td>
</tr>
<tr>
<td>Galaxy S7</td>
<td>5.10&quot;</td>
<td>577 dpi</td>
<td>4.0</td>
<td>360×640</td>
</tr>
</tbody>
</table>

Look at the resulting stage sizes: they are now ranging from 320×480 to 414×736 points. That's a moderate range, and it also makes sense: a screen that's physically bigger is supposed to have a bigger stage. The important thing is that, by choosing appropriate scale factors, we ended up with reasonable coordinate systems. This is a range we can definitely work with!

You might have noticed that the scale factor of the Galaxy S1 is not an integer value. This was necessary to end up with an acceptable stage size.

Let's see how I came up with those scale values. Create a class called `ScreenSetup` and start with the following contents:

```java
public class ScreenSetup
{
    private var _stageWidth:Number;
    private var _stageHeight:Number;
    private var _viewPort:Rectangle;
    private var _scale:Number;
    private var _assetScale:Number;

    public function ScreenSetup(
        fullScreenWidth:uint, fullScreenHeight:uint,
        assetScales:Array=null, screenDPI:Number=-1)
    {
        // ...
    }

    public function get stageWidth():Number { return _stageWidth; }
    public function get stageHeight():Number { return _stageHeight; }
    public function get viewPort():Rectangle { return _viewPort; }
    public function get scale():Number { return _scale; }
    public function get assetScale():Number { return _assetScale; }
}
```

This class is going to figure out the `viewPort` and `stage size` Starling should be configured with. Most properties should be self-explanatory — except for the `assetScale`, maybe.

The table above shows that we're going to end up with scale factors ranging from "1" to "4".
However, we probably don’t want to create our textures in all those sizes. The pixels of the densest screens are so small that your eyes can’t possibly differentiate them, anyway. Thus, you’ll often get away with just providing assets for a subset of those scale factors (say, 1-2 or 1-3).

- The `assetScales` argument in the constructor is supposed to be an array filled with the scale factors for which you created textures.
- The `assetScale` property will tell you which of those asset-sets you need to load.

Nowadays, it’s even rare for an application to require scale factor “1”. However, that size comes in handy during development, because you can preview your interface without requiring an extremely big computer screen.

Let’s get to the implementation of that constructor, then.

```javascript
public function ScreenSetup(
    fullScreenWidth:uint, fullScreenHeight:uint,
    assetScales:Array=null, screenDPI:Number=-1)
{
    if (screenDPI <= 0) screenDPI = Capabilities.screenDPI;
    if (assetScales == null || assetScales.length == 0) assetScales = [1];

    var iPad:Boolean = Capabilities.os.indexOf("iPad") != -1; ①
    var baseDPI:Number = iPad ? 130 : 160; ②
    var exactScale:Number = screenDPI / baseDPI;

    if (exactScale < 1.25) _scale = 1.0; ③
    else if (exactScale < 1.75) _scale = 1.5;
    else _scale = Math.round(exactScale);

    assetScales.sort(Array.NUMERIC | Array.DESCENDING);
    _assetScale = assetScales[0];

    for (var i:int=0; i<assetScales.length; ++i) ④
        if (assetScales[i] >= _scale) _assetScale = assetScales[i];

    _stageWidth  = int(fullScreenWidth / _scale); ⑤
    _stageHeight = int(fullScreenHeight / _scale);

    _viewPort = new Rectangle(0, 0, _stageWidth * _scale, _stageHeight * _scale);
}
```

① We need to add a small workaround for the Apple iPad. We want it to use the same set of scale factors you get natively on iOS.

② Our base density is 160 dpi (or 130 dpi on iPads). A device with such a density will use scale factor “1”.

③ Our scale factors should be integer values or 1.5. This code picks the closest one.

④ Here, we decide the set of assets that should be loaded.
5 The stage size depends on the scale factor we just created.

If you want to see the results of this code if run on the devices I used in the tables above, please refer to this Gist. You can easily add some more devices to this list and check out if you are pleased with the results.

Now that everything is in place, we can adapt the startup-code of Starling. This code presumes that you are providing assets with the scale factors ”1” and ”2”.

```javascript
var screen:ScreenSetup = new ScreenSetup(
    stage.fullScreenWidth, stage.fullScreenHeight, [1, 2]);

_startling = new Starling(Root, stage, screen.viewPort);
_startling.stage.stageWidth = screen.stageWidth;
_startling.stage.stageHeight = screen.stageHeight;
```

When loading the assets, make use of the `assetScale` property.

```javascript
var scale:Number = screen.assetScale;
var texturePath:String = "textures/" + scale + "x";
var appDir:File = File.applicationDirectory;

assetManager.textureOptions.scale = scale;
assetManager.enqueue(appDir.resolvePath(texturePath));
assetManager.loadQueue(...);
```

That's it! You still have to make sure to set up your user interface with the stage size in mind, but that's definitely manageable.

The Starling repository contains a project called *Mobile Scaffold* that contains all this code. It's the perfect starting point for any mobile application. (If you can't find the `ScreenSetup` class in your download yet, please have a look at the head revision of the GitHub project.)

If you are using *Feathers*, the class `ScreenDensityScaleFactorManager` will do the job of the `ScreenSetup` class we wrote above. In fact, the logic that's described here was heavily inspired by that class.

### 4.1.5. iPad and other Tablets

Back in the present, we're starting to wonder if it would make sense to port our game to tablets. The code from above will work just fine on a tablet; however, we will be facing a much larger stage, with much more room for content. How to handle that depends on the application you are creating.
Some games can simply be scaled up.

Games like *Super Mario Bros* or *Bejeweled* look great scaled to a big screen with detailed textures. In that case, you could ignore the screen density and calculate the scale factor based just on the amount of available pixels.

- The first iPad (resolution: 768×1024) would simply become a device with a stage size of 384×512 and a scale factor of "2".
- A retina iPad (resolution: 1536×2048) would also have a stage size of 384×512, but a scale factor of "4".

Others can display more content.

Think of *Sim City* or *Command & Conquer*: such games could show the user much more of the landscape. The user interface elements would take up less space compared to the game’s content.

Some will need you to rethink the complete interface.

This is especially true for productivity-apps. On the small screen of a mobile phone, an email client will show either a single mail, the inbox, or your mailboxes. A tablet, on the other hand, can display all three of those elements at once. Don’t underestimate the development effort this will cause.

### 4.2. Device Rotation

A very cool feature of today’s smartphones and tablets is that they recognize the orientation of the device in the physical world and may update the user interface accordingly.

To detect orientation changes in Starling, you first need to update your application’s AIR configuration file. Make sure that it includes the following settings:

```xml
<aspectRatio>any</aspectRatio> ①  
<autoOrients>true</autoOrients> ②
```

① The initial aspect ratio (*portrait*, *landscape* or *any*).
② Indicates whether the app will begin auto-orienting on launch.

When that’s in place, you can listen for a `RESIZE` event on the Starling stage. It is dispatched whenever the orientation changes. After all, an orientation change always causes the stage size to change, as well (exchanging *width* and *height*).

Update the dimensions of the Starling viewPort and stage in the corresponding event handler.
stage.addEventListener(Event.RESIZE, onResize);

private function onResize(event:ResizeEvent):void ①{
  updateViewPort(event.width, event.height);
  updatePositions(event.width, event.height);
}

private function updateViewPort(width:int, height:int):void ②{
  var current:Starling = Starling.current;
  var scale:Number = current.contentScaleFactor;

  stage.stageWidth = width / scale;
  stage.stageHeight = height / scale;

  current.viewPort.width = stage.stageWidth * scale;
  current.viewPort.height = stage.stageHeight * scale;
}

private function updatePositions(width:int, height:int):void ③{
  // Update the positions of the objects that make up your game.
}

① This event handler is called when the device rotates.
② Updates the size of stage and viewPort depending on the current screen size in pixels.
③ Updates your user interface so that it fits the new orientation.

Note that we had to update viewPort and stage size manually in the event listener. Per default, they will remain unchanged, which means that your application will appear cropped. The code above fixes that; it works for every scale factor.

The last part is going to be much harder: updating your user interface so that it fits into the new stage dimensions. This does not make sense for all games—but if it does, you should consider the additional effort. Your users will appreciate it!

The Scaffold project coming with Starling contains a possible implementation of this feature.

4.3. Summary

Make no mistake: developing for mobile platforms is not easy. The variety of the hardware makes it necessary to plan ahead and structure your code smartly. Furthermore, the market is extremely competitive, so your design needs to stand out from the masses.

Starling does what it can to help you with this process! With the tools you learned in this chapter, you will be able to master this challenge.
Chapter 5. Final Words

5.1. Achievement Unlocked!

You have successfully finished working through the *Starling Manual*. We covered a lot of ground, didn’t we? Pat yourself on the back and shave off that beard you’ve grown during the last hours.

Congratulations!

5.2. Getting Help

Want to share the knowledge you have just acquired? Or do you have any questions that haven’t been answered yet? The Starling community is eager to help you! Pay us a visit at the official [Starling Forum](#).

5.3. Want more?

If you want to become a true Starling master, don’t forget to get yourself a copy of the *Starling Handbook*! In addition to all the contents of this manual, it goes above and beyond to provide you with easy to follow recipes you can put to use in your own projects. And you’re supporting the continuing development of Starling at the same time!