Starling Handbook — Preview

Daniel Sperl

Table of Contents

Chapter 1. Getting Started

The first 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.

Chapter 2. Flappy Starling

In this chapter, we are going to jump (flap?) right into the development of an actual game. Along the way, you will get a peek at Starling's basic concepts. If you don't understand all the steps right away, don't worry: we will look at the details later. For now, I just want you to get a feeling for the framework, and to enjoy the sense of achievement that comes with finishing a game.

2.1. The Blueprint

Flappy Bird is a phenomenon. Developed over the course of just a few days by Vietnamese artist and programmer Dong Nguyen, it was released in 2013, sharing the fate of countless other games of its kind: nobody noticed it.

That changed in early 2014, however. For reasons that remain a mystery, it suddenly skyrocketed up the charts! At that time, the game was so successful that it earned about USD 50,000 per day from its in-app advertisements. The dream of every game developer come true, right?

At the height of this success, though, Dong pulled the game from the app stores. He said that he had a bad conscience about getting people addicted to his game. It only added to the hype, of course, and since then, countless clones of the game have appeared on the stores.

Well, it's time we made our own!

2.1.1. Setup and Gameplay

In all honesty: I admire this game. It demonstrates that it's often the most simple ideas that provide the most fun. Yes, it might be argued that it's not the most original game. Without question, though, it is definitely challenging and addictive!

If you haven't played it before, here's what *Flappy Bird* is about.

- A bird is flying through a side-scrolling landscape that could have been taken right out of Super Mario Bros.
- Each time you touch the screen, the bird flaps up a little bit, fighting against the constant pull of gravity.
- The player's job is to make it evade the obstacles that appear every few meters: green pipes that leave only a narrow gap for the bird to pass through.
- You score a point for every pair of pipes that the bird is passing. When you fail, you restart from the very beginning.

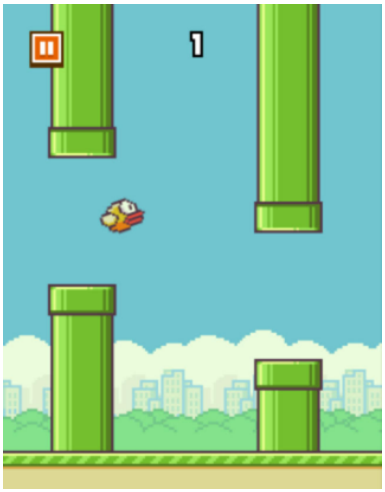


Figure 1. The Original: Flappy Bird.

As simple as this sounds, it's actually damn hard to score just a few points — and that's what this game is all about: the challenge is to stay focused and concentrated as long as possible, without making an error.

There's nothing else: no growing difficulty curve, no fancy graphics; it's just you and the bird. And since each session just takes a few seconds, it's perfect for a mobile game.

Furthermore, it's a game about a bird right? That's reason enough to make this the perfect example for our jump into Starling!

2.2. Starting Point

I set up a repository on *GitHub* that contains all the source code for this game. There's a *git tag* for each step we're going to take; that way you can follow along easily. First, clone the complete project to your development computer.

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

This will copy the game to the directory "Flappy-Starling". Navigate into this directory and checkout the tag `start`.

```
cd Flappy-Starling
git checkout start
```

This will revert the repository to the state we want to use to start off our project. To get an overview about what's there, take a look at the main folders within the project directory.

assets

The textures and fonts we are going to load at runtime. I prepared all the elements we need to set up the game world, as well as a colorful bitmap font that fits the style of the game.

lib

Contains the `starling.swc` library the game will use (v2.0.1). I added it directly to the repository

so that you can be sure the code will always compile.

src

The game's source code; we will spend most of our time here. Right now, it only contains a few very basic classes, but we will build on that soon.

We are working on a pure *Flash Player* project here (not *AIR*). That simplifies some things; after all, we want to focus completely on the code in this chapter.

You now have to set up your IDE for this project.

- If you are using *IntelliJ IDEA*, you're in luck: the repository contains suitable project and module files (in the root directory).
  - Open the complete project via ***File › Open*** or
  - import the module into an existing project via ***File › New › Module from Existing Sources***.
- For all other IDEs, please follow the setup procedures shown in the previous chapter.
  - Be sure to create a project for the *Flash Player* (not *AIR*), using pure *ActionScript 3*.
  - Link `lib/starling.swc` to the project, or use your local version of Starling.
  - The file `FlappyStarling.as` contains the main/startup class.

Once everything is set up, I recommend you compile and run the project. If your *Flash Player* fills up with a light blue color, you have succeeded.

The project already contains all the basic setup code for Starling so that we don't have to start from zero. Take a look around; it's really not much code yet! Basically, we're just setting up Starling (just as we did in the *Hello World* example) and load the assets.

2.2.1. The Game class

It's time to look at the root object that is going to host *Flappy Starling*.

Game.as

```
import starling.display.Sprite;
import starling.utils.AssetManager;

public class Game extends Sprite ①
{
    private static var sAssets:AssetManager;

    public function Game()
    { }

    public function start(assets:AssetManager):void ②
    {
        sAssets = assets;
    }

    public static function get assets():AssetManager ③
    {
        return sAssets;
    }
}
```

① The class extends `starling.display.Sprite`. I'll explain that in just a minute.

② The `start` method will be called from the startup-class. It also passes us all our assets.

③ For convenience, I added a static property that allows us to access the assets from anywhere via a simple call to `Game.assets`.

As you can see, there isn't that much going on yet. All we are doing at the moment is storing a reference to the `AssetManager` instance that's passed to us from the startup class.

The AssetManager

Wait, I haven't introduced you to the *AssetManager* yet! That's a very nice helper class that comes with Starling. You can stuff all kinds of game assets into it, for example textures, sounds, fonts, etc. That way, you don't have to manually load your assets when you need them, but can simply access them via their name.

At this state, our *AssetManager* instance already contains all the assets we need. The startup class (*FlappyStarling*) made those preparations for us.

Time to add some content to our game! I'm getting bored of this empty blue canvas. Modify the `start` method like this:

```
public function start(assets:AssetManager):void
{
    sAssets = assets;

    var skyTexture:Texture = assets.getTexture("sky"); ①
    var sky:Image = new Image(skyTexture); ②
    sky.y = stage.stageHeight - skyTexture.height; ③
    addChild(sky); ④
}
```

① Get a reference to the sky texture from the *AssetManager*.

② Create an image with that texture.

③ Move the image to the very bottom of the stage.

④ Add the image to the stage.

When you type that code, be careful when you come to the `Texture` keyword. There are several classes named `Texture`, and your IDE will probably ask you which one you want to import.

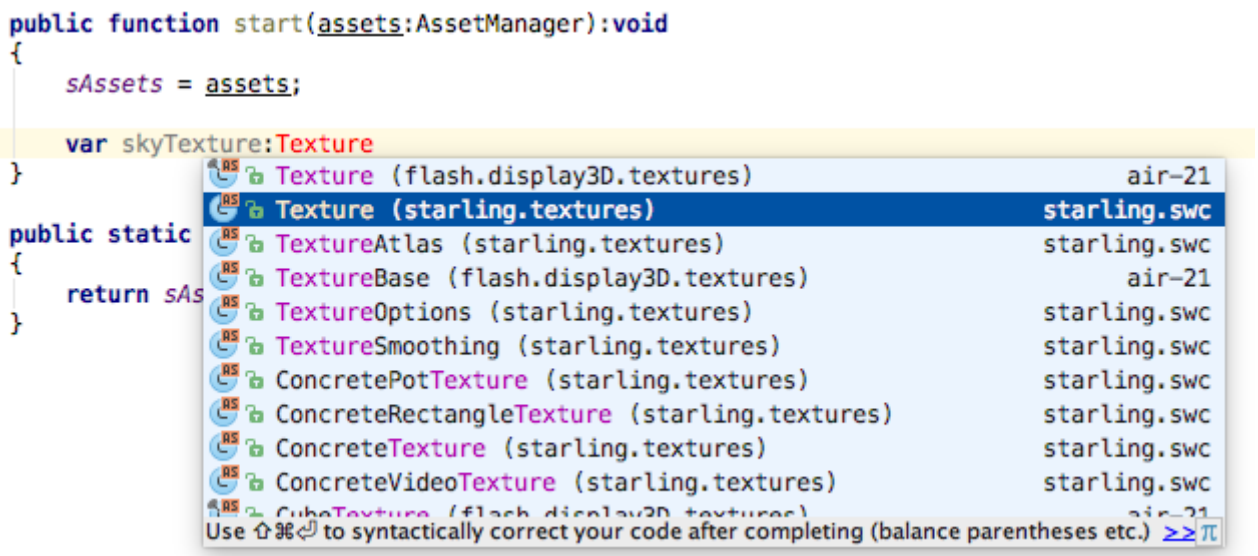


Figure 2. Choosing the correct package.

When working with Starling, you will usually need to pick a class from the `starling` package. Otherwise, the IDE will import the wrong class, and you will run into weird errors. If you accidentally picked the wrong class, it's easy to fix: just delete that `import` statement at the top of the file and try again.

When done right, your IDE will add the following *import* at the top of the file:

```
import starling.textures.Texture;
```

In the source code samples of this book, I'm normally omitting the `import` statements to save space. If you run into a compile error even though you typed the code correctly, it's always a good idea to check your imports.

Figure 3. The game's background image is in place.

Finally, we have drawn something to the screen! But how did we do that?

- First, we fetched a reference to the sky texture from the *AssetManager*. In Starling, a `Texture` stores the pixels of an image (similar to the `BitmapData` class in classic Flash).
- Then, we wrapped that texture in an `Image` instance (think `Bitmap` in classic Flash). That's one of several classes that inherit from `starling.display.DisplayObject` — the building blocks of everything you see on the screen.
- Now, it's just a matter of moving it to the right position (via the x- and y-coordinates) and adding it to Starling's display list (via `addChild`).

It might be necessary to clarify how we moved the texture to the bottom of the screen. The x- and y-coordinates control the position of an object relative to its parent. If we left both coordinates at `0`, the image would appear at the top left. That's because in Starling, the coordinate system looks like this:

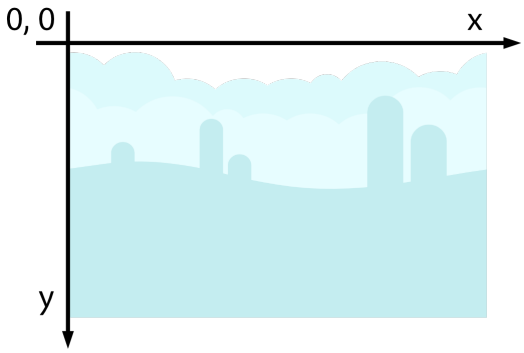


Figure 4. Starling's coordinate system has its root at the top left.

We also accessed the `stage` object and queried its height. The *Stage* describes the area to which everything is rendered; it is the very root of Starling's display list (and parent of the *Game* instance we are working with right now).

Don’t confuse this stage with the one provided by "classic" Flash. Starling manages its own, completely separate display list.

2.3. Organizing the Code

It’s nice that there is something on the screen now — but before going on, we need to ponder a little bit about how we’re going to organize all of the code we’re going to write.

Of course, we could put all of our game elements right into the *Game* class, but that’s going to get messy quickly. Instead, it’s a good idea to split the game up into logical units; each with a very limited responsibility.

After all, we’re not only going to create this landscape with the flying bird, but also a title screen, maybe a leaderboard, etc. But how should we organize all these elements?

2.3.1. Introducing: The Sprite

You have already seen this class before: our *Game* class is extending *Sprite*. This class is one of the most important building blocks within Starling. It’s a container that can group together multiple elements.

Here’s a simple example showing a typical use-case of a sprite:

```
var body:Image = ...;
var tail:Image = ...;
var beak:Image = ...;

var bird:Sprite = new Sprite();
bird.addChild(body);
bird.addChild(tail);
bird.addChild(beak);

addChild(bird);
```

The `bird` object now contains body, tail and beak, grouped together in one object. When you want to move around the bird, you just have to change the x- and y-coordinates of the `bird` instance; all child elements will move with it.

If you want the bird sprite to be reusable, you could also use the following approach:

```
public class Bird extends Sprite ①
{
    public function Bird()
    {
        var body:Image = ...;
        var tail:Image = ...;
        var beak:Image = ...;

        addChild(body); ②
        addChild(tail);
        addChild(beak);
    }
}

var bird:Bird = new Bird(); ③
addChild(bird);
```

① Create the *Bird* class so that it extends *Sprite*.

② Just like before, we're adding some child objects; here, we are doing it in the constructor.

③ Instantiate and use the bird sprite like this.

That's a very common pattern when working with Starling. One might even argue that this is your main task as a developer: to organize your display objects hierarchically by grouping them into sprites (or other containers), and to write the code that glues everything together.

As mentioned above, the *Game* class is our top-level sprite. It's going to be the *controller* that manages what's going on at the highest level. The other elements will follow the lead; one of them is the *World* class.

2.3.2. The World class

The *World* class is going to deal with the actual game mechanics. When the *Game* tells it to launch the bird, it will do just that; and when the bird crashes into an obstacle, it will notify the *Game* and will wait for instructions about how to go on.

For starters, create that new class and add the following code:

World.as

```
public class World extends Sprite
{
    private var _width:Number;
    private var _height:Number;

    public function World(width:Number, height:Number)
    {
        _width = width; ①
        _height = height;

        addBackground(); ②
    }

    private function addBackground():void
    {
        var skyTexture:Texture = Game.assets.getTexture("sky"); ③
        var sky:Image = new Image(skyTexture);
        sky.y = _height - skyTexture.height;
        addChild(sky);

        var cloud1:Image = new Image(Game.assets.getTexture("cloud-1"));
        cloud1.x = _width * 0.5;
        cloud1.y = _height * 0.1;
        addChild(cloud1);

        var cloud2:Image = new Image(Game.assets.getTexture("cloud-2"));
        cloud2.x = _width * 0.1;
        cloud2.y = _height * 0.2;
        addChild(cloud2);
    }
}
```

① I added `width` and `height` arguments to the constructor. That way, the *Game* class can pass in the stage dimensions.

② The code that creates the sky texture was moved into a separate method. While I was at it, I also added two clouds to the sky to make it look a little more interesting.

③ Thanks to the static `assets` property on the *Game* class, we can easily access our textures without passing the *AssetManager* instance around.

Relative Positioning

Right now, we are creating a Flash game with a fixed size of 320x480 pixels. However, I'd like to port the game to mobile, later.

Mobile devices have all kinds of different screen sizes, so it's a good idea to take that into account right away. I'm doing that by working with relative instead of absolute positions.

```
// bad: fixed position
_cloud1.x = 160;

// better: relative position
_cloud1.x = _width * 0.5;
```

What's left to do, of course, is to reference this new class from *Game.as*. Here's how it should look like now:

Game.as

```
public class Game extends Sprite
{
    private static var sAssets:AssetManager;

    private var _world:World; ①

    public function Game()
    { }

    public function start(assets:AssetManager):void
    {
        sAssets = assets;

        _world = new World(stage.stageWidth, stage.stageHeight); ②
        addChild(_world);
    }

    public static function get assets():AssetManager
    {
        return sAssets;
    }
}
```

① Add a new member variable.

② Instantiate the new class (passing the stage dimensions along) and add it to the stage.

Compile and run to check if everything worked out. Except for the two inconspicuous new clouds, it should look just like before.

Figure 5. Our refactoring didn't change the look of our game much.

2.4. Adding the Bird

It's a game about a bird, so let's finally add the hero to our game. Our assets contain three textures that illustrate a flapping red bird. That's all we need for a neat flapping animation.

Figure 6. The three frames of our bird movie.

Just like in classic animation movies, we will display those images in quick succession to create the illusion of movement.

2.4.1. The MovieClip class

To do that, we first create a *Vector of Textures* that references the frames of our animation.

```
var birdTextures:Vector.<Texture> = Game.assets.getTextures("bird-");
birdTextures.push(birdTextures[1]);
```

The textures we are looking for are named `bird-1`, `bird-2`, and `bird-3`. To get those textures from the *AssetManager*, the `getTextures` method comes in handy; it returns all textures with a name that starts with a given string (sorted alphabetically). The middle frame is supposed to be reused for the upward-movement of the wing, so I added another reference to that texture to the very end of the vector.

To actually display those animation frames, let me introduce you to the *MovieClip* class. It works

very similar to an *Image* (it extends that class), with the difference that it changes its texture over time.

Let's try this out in our *World* class. Make the following modifications:

```
private var _bird:MovieClip; ①

public function World(width:Number, height:Number)
{
    /* ... */

    addBackground();
    addBird(); ②
}

private function addBird():void
{
    var birdTextures:Vector.<Texture> = Game.assets.getTextures("bird-");
    birdTextures.push(birdTextures[1]);

    _bird = new MovieClip(birdTextures); ③
    _bird.pivotX = 46; ④
    _bird.pivotY = 45;
    _bird.x = _width / 3; ⑤
    _bird.y = _height / 2;

    addChild(_bird);
}
```

① Add a new member variable referencing the bird.

② Add a new helper method that creates the bird.

③ Create the actual *MovieClip*.

④ Move the bird's pivot point to the center of its body (see below).

⑤ Move the bird itself to its starting point, a little left of the center of the screen.

As you can see, the creation of the *MovieClip* is quite straight-forward; you simply pass the *Vector* of textures to its constructor.

Pivot Points

The code also introduces the concept of *pivot points*. Per default, the origin of all display objects is at the top left (just as the stage's origin is at the top left). Via the `pivotX` and `pivotY` properties, you can move that origin (the pivot point) to a different location.

Above, we are moving the pivot point to the center of the bird's body, which has two advantages:

1. Should we want to rotate the bird later, it will rotate around its center.
2. It will simplify the collision detection code we need to write a little later.

Run the project now to see the bird airborne!

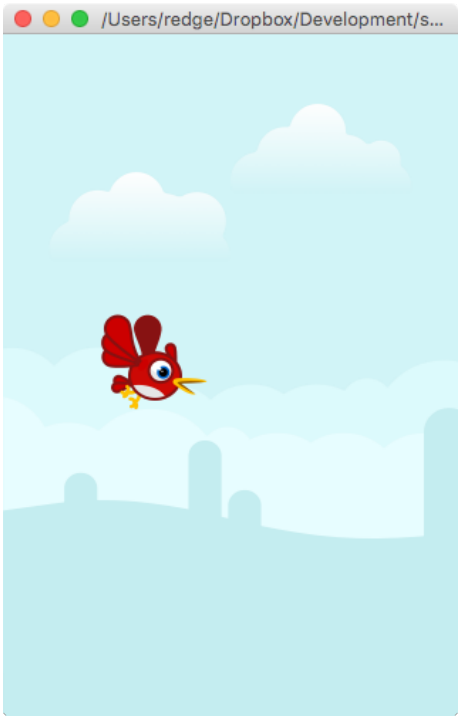


Figure 7. The bird is in the air.

Something seems to be wrong, though: there is no flapping animation, just a static image!

That's because the *MovieClip* doesn't have any information about the passage of time yet; and without that information, it will simply stick to the very first frame.

In this case, we want to have full control over the bird's animation, so we will update it manually once per frame. Let's do this by adding the following method to the *World* class:

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

The `advanceTime` method moves the playhead forward by the given time. Thus, we want this method

to be called once per frame, and the `passedTime` parameter needs to contain the time passed since the previous frame. Sounds like a job for our *Game* class, right? Open it up and add the following code:

```
public function start(assets:AssetManager):void
{
    /* ... */

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event, passedTime:Number):void
{
    _world.advanceTime(passedTime);
}
```

This code introduces a concept that hasn't been mentioned before: events. We will look into that topic in detail a little later, but that code should be rather intuitive. We listen to an `ENTER_FRAME` event, which is a standard event that's dispatched once per frame to every display object. As a result, the method `onEnterFrame` (the event handler) will be called repeatedly.

The event handler also contains an argument called `passedTime`, which is just what we need in the `advanceTime` method we just defined on *World*.

Which means that everything is in place to get that bird flapping! Compile and run the code now to see it in action.

2.5. Scrolling

Right now, the bird is flying in empty space. We will now add some grassland below to give the player some sense of the bird's height and movement. Let's start with the following additions to the *World* class:

```
private var _ground:Image; ①

public function World(width:Number, height:Number)
{
    /* ... */

    addBackground();
    addGround(); ②
    addBird();
}

private function addGround():void
{
    var tile:Texture = Game.assets.getTexture("ground");

    _ground = new Image(tile); ③
    _ground.y = _height - tile.height;
    addChild(_ground);
}
```

① Add a new member variable referencing the ground.

② Add a new helper method that creates the ground.

③ Create the ground image and move it to the bottom.

That's quite straight-forward. The result will look like this:

Figure 8. The first tile of the ground.

Oops! That ground tile is way too short for the width of the screen. It occupies just a small area at the very left. However, that texture supports tiling, i.e. when you add the same image over and over, it will fill up seamlessly.

One way to fill up the floor would be to simply create several of those images and put them all next to each other. That's what you would do in most game engines.

However, there is an even simpler way: Starling's `tileGrid`. That's a property that is defined on the `Image` class, and as its name suggests, it was built exactly for the purpose of filling an image with tiles.

It works like this: you assign a rectangle that indicates where a single tile should be placed. Starling will then fill out the rest so that the complete area of the image repeats that pattern.

With that in mind, we modify the code of the `addGround` method slightly.

```
_ground = new Image(tile);
_ground.y = _height - tile.height;
_ground.width = _width; ①
_ground.tileGrid = new Rectangle(0, 0, tile.width, tile.height); ②
```

① Make the image as wide as the stage.

② Assign a rectangle that points to the first tile. Starling will add other tiles automatically.

That's quite simple, right? Looking at the output, it seems that did the trick!

Figure 9. The same tile, repeated along the full width.

2.5.1. Animating the Ground

That's all still a rather static business, though. Instead, we want to create the impression that the camera is following a bird that's flying to the right. To do that, we will move the ground slowly to the *left*, just like an endless ribbon.

Had we built the ground from separate images, this would be quite painful.

- In each frame, move all those images a little to the left.
- If an image moves out of the visible area, put it to the very right.
- That way, we have an endless stream of tiles.

You might run into that situation one day — it's rare that the ground is built from just a single tile. However, in this situation, with our `tileGrid` in place, it's much simpler: we let Starling do the work for us.

Our task:

- In each frame, move the (virtual) `tileGrid` rectangle a little to the left.
- That's it.

Our existing `advanceTime` method will pull that off.

```
public class World extends Sprite
{
    private static const SCROLL_VELOCITY:Number = 130; ①

    /* ... */

    public function advanceTime(passedTime:Number):void
    {
        advanceGround(passedTime); ②
        _bird.advanceTime(passedTime);
    }

    private function advanceGround(passedTime:Number):void
    {
        var distance:Number = SCROLL_VELOCITY * passedTime; ③

        _ground.tileGrid.x -= distance; ④
        _ground.tileGrid = _ground.tileGrid; ⑤
    }
```

① A constant defines the speed of movement: 130 points per second.

② The new method `advanceGround` will do the work.

③ Since `passedTime` is given in seconds, a simple multiplication will yield the passed distance.

④ Move `tileGrid.x` to the left.

⑤ That's a small API oddity: for the changes to show up, `tileGrid` has to be re-assigned.

There really isn't much to it: all that we are doing is reducing the value of `tileGrid.x` a little in each frame. The result is a neat, endless animation of the ground ribbon.

You'll notice that the we are not moving the blue background in any way. That's intentional: by keeping it at a fixed position, we're creating the illusion that it is very far away.

2.6. Physics

Flappy Starling ought to be a game, but right now, the level of interactivity is rather … lacking. Thus, the next step should be to make the bird's flight follow some basic physical rules, and to let the player take control.

We have two choices when it comes to physics:

- We could do it ourselves: *Isaac Newton* came up with a couple of formulas; we can plug those into our code.
- Or we could add a physics library to our project and let it do the heavy lifting.

Which approach should we use?

Make no mistake: it's actually *really* hard to create a convincing physics simulation. Things get complicated quickly, especially when multiple objects interact with each other. The math will become very challenging, and before you know it, the calculations will eat up all available CPU time.

Thus, I'd typically recommend using a turn-key physics engine (like Nape). Let it do the work for you while you concentrate on your actual project. Games like *Angry Birds* rely on the realism only such an engine can deliver.

The physics of *Flappy Starling*, on the other hand, are extremely simple. We have one flying bird that's affected by gravity and can flap upwards on touch of the screen. That's something we can easily do ourselves.

2.6.1. What goes up, must come down

A bird that flies in the air is constantly being pulled down by gravity. What does that mean?

Gravity changes the velocity of an object: push a stone off a cliff and its velocity will rise from "zero" (the moment it loses contact) to … well, "very fast" before it hits the ground. The higher the cliff, the faster it will be on impact. Each instant it falls, it will be faster than the instant before.

So we make a mental note that we need to store our bird's current velocity somewhere, as well as the acceleration imposed on it by gravity.

The other force acting on the bird is — the player. When he or she touches the screen, the bird is catapulted upwards, i.e. the velocity is set to a fixed value.

Let's incorporate all that into our *World* class:

```
public class World extends Sprite
{
    private static const SCROLL_VELOCITY:Number = 130;
    private static const FLAP_VELOCITY:Number = -300; ①
    private static const GRAVITY:Number = 800; ②

    private var _bird:MovieClip;
    private var _birdVelocity:Number = 0.0; ③

    /* ... */
}
```

① The bird's velocity will be set to that value when the user touches the screen.

② The *gravity* constant stores the acceleration that gravity inflicts on the bird.

③ This value will store the current velocity of the bird; it will be updated in each frame.

You might ask why *birdVelocity* is a scalar, not a vector. That's because our bird actually only moves along the y-axis (i.e. vertically). There is no horizontal movement — that's just an illusion we created by moving the ground to the left!

Don't forget to assign `birdVelocity` a value (zero, in this case). *AS3* has a very peculiar default value for variables of type *Number*: `NaN` (not a number). So much for language designers not having a sense of humor!

The next step is to update `birdVelocity` each frame, and to move the bird by that velocity.

```
public function advanceTime(passedTime:Number):void
{
    /* ... */

    advancePhysics(passedTime); ①
}

private function advancePhysics(passedTime:Number):void
{
    _bird.y += _birdVelocity * passedTime; ②
    _birdVelocity += GRAVITY * passedTime; ③
}

public function flapBird():void ④
{
    _birdVelocity = FLAP_VELOCITY;
}
```

① The new method `advancePhysics` will contain our tiny physics engine.

② Move the bird along the y-axis according to its current velocity.

③ Add the `GRAVITY` to the bird's current velocity.

④ This method will be called when the player touches the screen. It instantly sets the bird's velocity to a fixed value.

That's actually all our physics code. You probably agree that adding a full-fledged physics library would have been an overkill in this situation!

There's one more step to do: allowing the player to control the bird, i.e. to call that new `flapBird` method. Please open up the *Game* class and make the following modifications:

```
public function start(assets:AssetManager):void
{
    /* ... */

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(TouchEvent.TOUCH, onTouch); ①
}

private function onTouch(event:TouchEvent):void
{
    var touch:Touch = event.getTouch(stage, TouchPhase.BEGAN); ②
    if (touch) _world.flapBird(); ③
}
```

① Another event listener: this one listens for `TOUCH` events on the stage.

② In Starling, touches pass through different phases. We are interested in the first phase only: `TouchPhase.BEGAN` means that the finger has just begun touching the screen.

③ If there is such a touch, we let the bird flap upwards.

Just like we were notified of the passing of time via the `ENTER_FRAME` event, there is an event that's dispatched when the user's finger touches the screen: the `TOUCH` event. When such a touch occurs, we let the bird flap upwards.

Touch Events vs. Mouse Events

We are working on a *Flash* project, which means that most users will probably use a mouse to flap the bird. Why aren't we listening to *mouse* events, then?

In Starling, *touch* and *mouse* input handling is unified. When the user clicks on the mouse button, this will be registered just the same as if he had touched the screen at this position. The advantage: you can use the same code to handle both mouse and touch input. This greatly simplifies porting the game to mobile platforms.

Start the game to check out if everything works. The bird will start to fall downwards immediately, and you can bring it up again by clicking on the screen repeatedly. Finally, this is starting to feel like a real game, right?

What's a little annoying, though, is that when you're not careful, the bird falls off the bottom of the screen, and it's quite an effort to bring it back up again.

So I'd like to fix that before moving on, even though I'm getting a little ahead of myself. What we're going to do is implement the first part of the *collision detection* logic, a topic we will soon come back to.

Just like the physics code, this needs to be part of the *World* class.

```
private static const BIRD_RADIUS:Number = 18; ①

public function advanceTime(passedTime:Number):void
{
    /* ... */

    checkForCollisions(); ②
}

private function checkForCollisions():void
{
    var bottom:Number = _ground.y - BIRD_RADIUS; ③

    if (_bird.y > bottom) ④
    {
        _bird.y = bottom;
        _birdVelocity = 0;
    }
}
```

① Add a new constant at the very top: the radius of the bird's body circle (in points).

② Collision detection is handled in a separate method, called every frame.

③ The bird's fall will stop a little upwards of the ground image. (Remember: we moved the bird's pivot point to the center of its body.)

④ When we hit bottom, the bird's velocity is set to zero immediately, and we make sure it doesn't move any further down.

That makes playing around with our prototype much easier! Granted: when we hit rock bottom, it should actually mean "game over", but the bird just moves on. That's the topic of the next chapter!

Figure 10. Hitting rock bottom.

2.7. Moving on

That's where this sample chapter ends. The finished book will show you all that needs to be done to make this into a real game. Stay tuned!

Chapter 3. 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.

Special Effects

Effects and filters that will make your graphics stand out.

Utilities

A number of helpers to make your life easier.

In this extract, we will look at the event system.

3.1. 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.

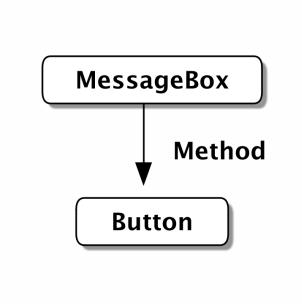
That's what Starling's event mechanism is for.

3.1.1. 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*.



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

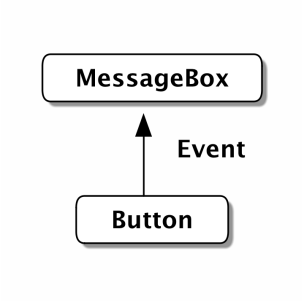
```
public class MessageBox extends DisplayObjectContainer
{
    private var _yesButton:Button;

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

① Communicate with the *Button* via a property.

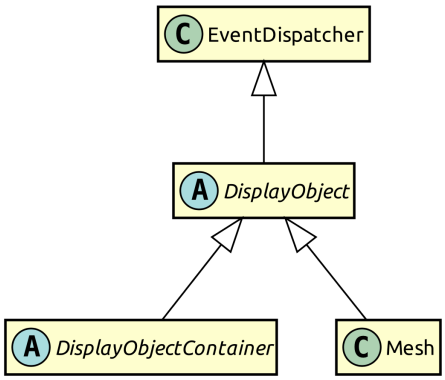
The *Button* instance, on the other hand, does not own a reference to the message box. After all, a button can be used by any component — it's totally independent of the *MessageBox* class. That's a good thing, because otherwise, you could only use buttons inside message boxes, and nowhere else. Ugh!

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.



3.1.2. 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.removeEventListener("bark", onBark); ①
einstein.removeEventListeners("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

3.1.3. 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.

② The reason for creating a custom event: we want to store some information with it. Here, that's the `reason` String.

③ Call the super class in the constructor. (We will look at the meaning of `bubbles` shortly.)

④ Make `reason` accessible via a property.

The dog can now use this custom event when barking:

```
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 ①
{
    if (event.reason == "hungry") ②
        einstein.feed();
    else
        einstein.walk();
}
```

① Note that the parameter is of type `BarkEvent`.

② 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!

3.1.4. 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.

3.1.5. 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.

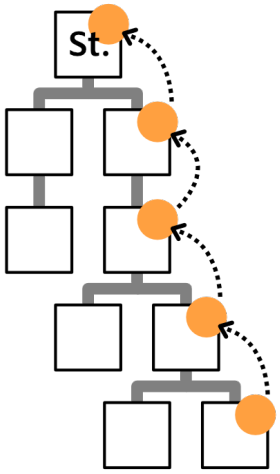


Figure 11. An event bubbles all the way up to the stage.

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.

3.1.6. 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.

```
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.

Figure 12. Simulating Multitouch with mouse and keyboard.

To react to touch events (real or simulated), you need to listen for events of the type `TouchEvent.TOUCH`.

```
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:

```
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).

```
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.

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

Chapter 4. Advanced Topics

With all the things we learned in the previous chapters, you are already well equipped to start using Starling in real projects. If you already did that: chapeau, that's the spirit! Using Starling is definitely the best way to really learn it.

However, in the course of this project, you might have run into a few things that seemed suboptimal:

- Managing the large number of assets was quite painful.
- Talking of assets: your textures quickly ate up all available memory.
- You probably encountered a *context loss* a couple of times. WTF!? [1: My editor said it's not polite to swear, but (1) I used an acronym and (2) context loss really s*cks.]
- It's very likely that you used up more memory and CPU cycles than what's necessary.
- Or you might be one of those masochists who like to write their own vertex and fragment programs, and didn't know where to start.

Oddly enough, that perfectly summarizes what this chapter is going to be about. Don't miss it!

Chapter 5. 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 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.]

Among other topics, this chapter will introduce you to the following topics:

- Multi-Resolution Development
- Device Rotation

Chapter 6. Pro Tips

Now that you know every nut, bolt and screw of the Starling API, I'd like to give you a taste of what can be achieved if it's used the right way. What follows is a number of cookbook-like recipes that may help you out in different situations. I also threw in a few tips that apply to Flash and ActionScript in general. In short, there should be something for everybody!

You will learn

- about the memory pitfalls coming with the *ByteArray* API, as well as how to use it as a *StringBuilder*.
- how best to embed an SWF file in a website.
- how to recognize the most important multi-touch gestures.
- that the *FragmentFilter* class can be used to replace `cacheAsBitmap` and `BlendMode.LAYER`, and to apply selective anti-aliasing.
- that the *MeshBatch* provides the fastest way to render a massive number of tiles and …
- … how to avoid the problems that arise in the process.
- that you can load textures asynchronously to keep your app responsive.
- how to squeeze more textures into memory by compressing them at runtime.
- several neat visual effects that utilize some of Starling's standard features.

As a preview, I will show you how to create a neat reflection effect with Starling's *DisplacementMapFilter*.

6.1. Water Reflection

In the early years of the Internet, many web developers — frustrated by the limitations of early HTML — used Flash simply to show off. Even if the website itself didn't have much content, you could be sure there was a Flash intro and some kind of animated reflection effect. Anybody with a tiny bit of self respect simply *had* to do it!

Thankfully, those days are long over. Nevertheless, a part of me still likes eye-candy like that. If used with care, an unobtrusive reflection might be turned into a nice special effect. So let's look at how this can be achieved with Starling.

The Plain Reflection

For this sample, we will set up a sprite with a bird flying in front of a colorful sky.

The basic (flat) reflection is actually achieved with a really lame trick: we simply create the same sprite twice and flip the bottom one vertically. Here is the basic setup-code, to be called e.g. from the constructor of a sprite.

```
private function setup():void
{
    var topSprite:Sprite = createSprite(); ①
    addChild(topSprite);

    var bottomSprite:Sprite = createSprite(0xddeeff); ②
    bottomSprite.scaleY = -1; ③
    bottomSprite.y = topSprite.height * 2;
    addChild(bottomSprite);
}

private function createSprite(tint:uint=0xffffff):Sprite
{
    var sprite:Sprite = new Sprite();
    var sky:Image = new Image(assets.getTexture("dusk-sky"));
    sky.color = tint;
    sprite.addChild(sky);

    var bird:MovieClip = new MovieClip(assets.getTextures("bird-")); ④
    bird.alignPivot();
    bird.x = sky.width / 2;
    bird.y = sky.height / 2;
    sprite.addChild(bird);
    juggler.add(bird); ⑤

    return sprite;
}
```

① The top sprite contains the upright objects; that's what we are directly looking at.

② The bottom sprite represents the reflection. By tinting the sky image in light blue, we can make it look more like water.

③ This code flips the bottom sprite vertically.

④ The bird animation is implemented with a *MovieClip*.

⑤ This code supposes that the class contains a custom *Juggler*. You could also use the standard `Starling.juggler`, of course.

Now what we are seeing is this:

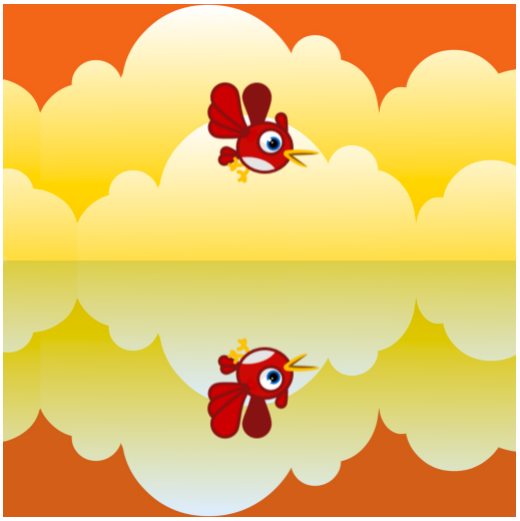


Figure 13. For the basic (plain) reflection effect, the bottom sprite is flipped.

Adding a Distortion

For the distortion effect, we are going to use Starling's *DisplacementMapFilter*. That's an extremely versatile filter that can distort the pixels of an object via a *map texture*. It's not the easiest filter to use and set up, but once you master it, you can achieve amazing things.

Step one is to create a map texture. To create such a texture, we can use the `perlinNoise` method of Flash's *BitmapData* class.

The Perlin Noise algorithm produces a pseudo-random noise texture that's perfect to simulate natural phenomena like clouds or water.

The following settings work fine:

```
var perlinData:BitmapData = new BitmapData(width, height, false);
perlinData.perlinNoise(200, 12, 2, 0, true, true, 0, true);
```

The fifth parameter of the `perlinNoise` method is called `stitch`. Note that we set it to `true` — this will become important in a minute.

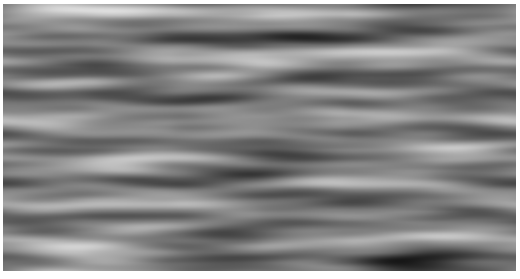


Figure 14. Perlin noise textures can be created at runtime.

The sophisticated process I used to arrive at suitable values is called "trial & error".

Now we can create a *DisplacementMapFilter* with our Perlin noise as `mapTexture`:

```
var mapTexture:Texture = Texture.fromBitmapData(perlinData);
var filter:DisplacementMapFilter = new DisplacementMapFilter(
    mapTexture, BitmapDataChannel.RED, BitmapDataChannel.RED, 15, 15);

bottomSprite.filter = filter;
```

Here's what all those parameters are:

mapTexture
: The texture with our Perlin noise.

componentX
: Displacement in x-direction is read from the `RED` channel.

componentY
: Displacement in y-direction is read from the `RED` channel, too.

scaleX
: Pixels are distorted up to 15 points horizontally.

scaleY
: Pixels are distorted up to 15 points vertically.

The *components* decide which channel of the map texture is used for the distortion. In our case, the filter looks at the values of the red channel. Since our texture is black and white, though, it doesn't really matter which one we use.

With that filter applied, the bottom sprite looks like this:

Figure 15. The bottom sprite with attached DisplacementMapFilter.

Animating the Reflection

Right now, this effect is purely static. In order to simulate a live, moving surface, we need to animate this reflection. How could we do that?

The first thing that comes to mind: we could update the Perlin texture each frame, always slightly changing its parameters. That would work — but it's very expensive; after all, we would need to create and upload a new texture in each and every frame. We clearly need a more efficient approach.

If you review the *DisplacementMapFilter* API, you'll find that it contains the properties `mapX` and

`mapY`. Those parameters configure an offset for the map texture lookup. In other words, you can move the map texture to a different position relative to the filtered object.

That's exactly what we need: by tweening `mapY` to a different value, we can slowly move the map texture downwards. In motion, the output will look just like the rising and falling waves of water.

The problem, though, is that this animation can't go on forever. We can only move the map texture while it's still overlapping the target image. If we move it any further, the map texture will end, and there's no more distortion.

That, however, can be solved with a neat little trick.

In Stage3D (and Starling), a texture can be set up to repeat endlessly — just like a wallpaper. When our map texture is configured like that, we will never run out of Perlin noise.

Texture repeat does not work with all kinds of textures, though. You can only use this feature with textures that have side lengths that are powers of two (e.g. 256×128, 512×1024, etc). That's not really a problem in our case, since the Perlin texture is created dynamically, anyway.

Let's move the generation of the map texture into a new method:

```
private function createMapTexture(minWidth:Number, minHeight:Number):Texture
{
    var width:Number  = MathUtil.getNextPowerOfTwo(minWidth); ①
    var height:Number = MathUtil.getNextPowerOfTwo(minHeight);
    var perlinData:BitmapData = new BitmapData(width, height, false);
    perlinData.perlinNoise(200, 12, 2, 0, true, true, 0, true); ②
    return Texture.fromBitmapData(perlinData, false, false, 1, "bgra", true); ③
}
```

① For both width and height, we pick the next suitable powers of two. For example, if the object is 400 pixels wide, the texture will have a width of 512 pixels.

② The Perlin noise is created just like before, with `stitch` enabled.

③ This method call leaves all parameters at their default values — except for last one: `forcePotTexture`. This makes sure the texture is created with the specific "power-of-two" texture type.

Since we enabled the `stitch` parameter when creating the Perlin texture, and since we created it as "power-of-two" texture, it can be seamlessly repeated.

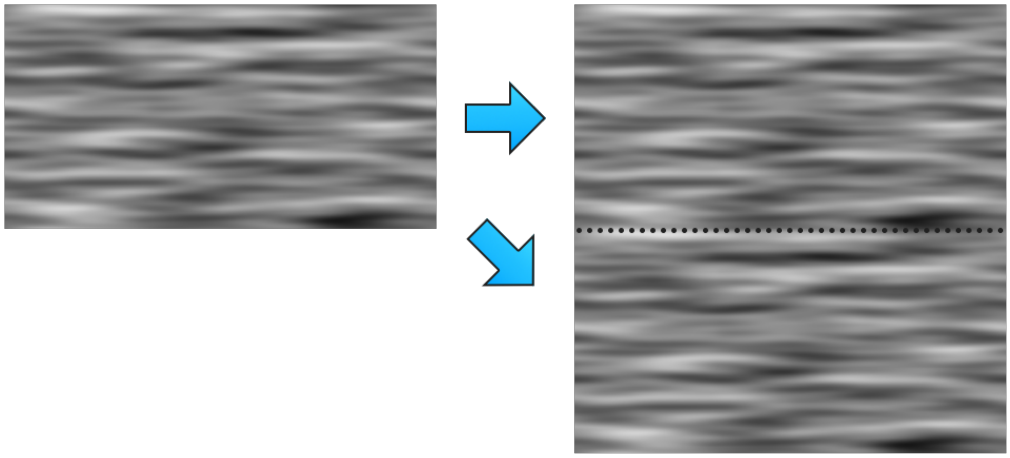


Figure 16. The Perlin texture can be seamlessly repeated.

With that method prepared, we can now instantiate the *DisplacementMapFilter*.

```
var map:Texture = createMapTexture(bottomSprite.width, bottomSprite.height);
var filter:DisplacementMapFilter = new DisplacementMapFilter(
        map, BitmapDataChannel.RED, BitmapDataChannel.RED, 15, 15);
filter.mapRepeat = true; ①
juggler.tween(filter, 5, { mapY: map.height, repeatCount: 0 }); ②
bottomSprite.filter = filter;
```

① This is crucial: we need to tell the filter that the map texture should actually repeat.

② The animation itself is a simple tween of `mapY`. To make the tween repeat endlessly, we assign a `repeatCount` of zero.

To make this a little clearer, think of the map texture as an endless ribbon that lines up one Perlin texture after the other. When you move that ribbon downwards, there is always another texture to replace the previous one. Our tween will reset after scrolling through just one texture, but we could also just raise `mapY` indefinitely; we will never run out of textures.

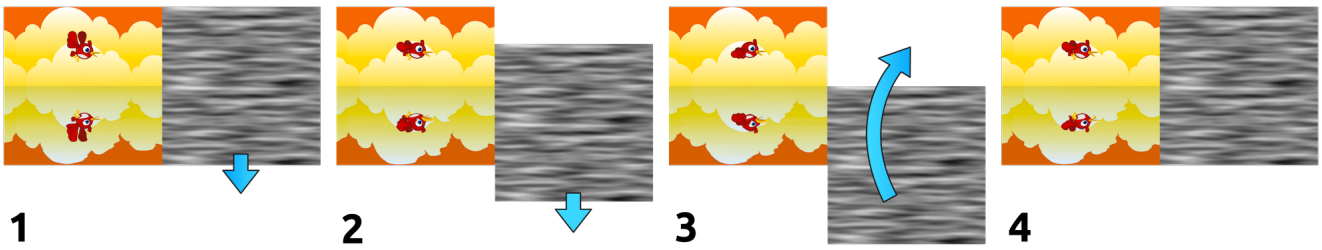


Figure 17. Endlessly animating the map texture.

If you try that out now, you'll see that the animation does in fact loop seamlessly. However, one small flaw is remaining: the edges of the object are distorted, too.

To get rid of that, we need to deactivate all padding on the filter. Per default, the *DisplacementMapFilter* sets up `padding` high enough so that nothing gets clipped away. In our case, this clipping is actually what we want, so we undo that automatism.

```
filter.padding.setTo(); ①
```

① Sets the padding to zero in all directions.

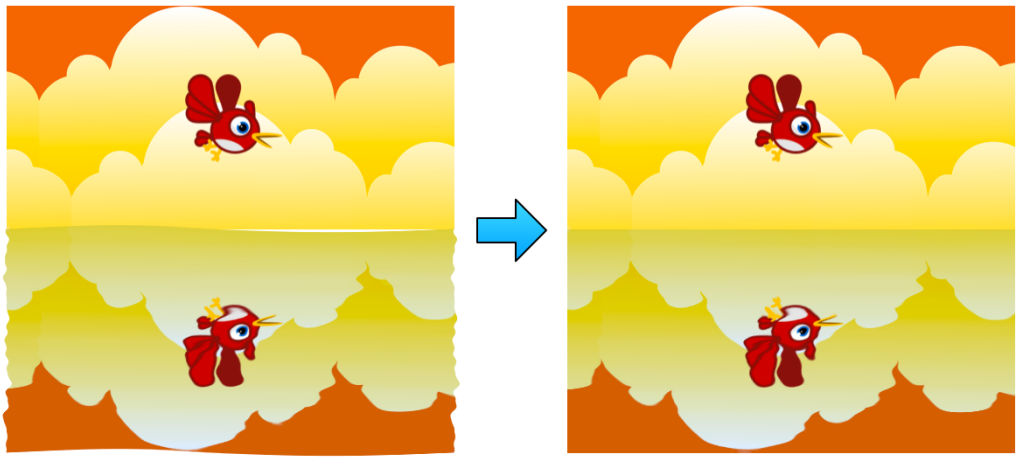


Figure 18. With padding deactivated, the bounds of the object stay intact.

I think the outcome is pretty neat, especially considering that this effect is actually rather lightweight! After all, it's just the one `mapY` property that's animated in each frame.

Chapter 7. Extensions

To this point, we looked at all kinds of features that are part of the actual Starling library. Of course, these features cannot cover *all* possible use cases, though. It would be impossible to provide everything out of the box; every game or app has different needs and requirements, after all.

For this reason, Starling does not even attempt to be a jack-of-all-trades. Its goal has always been to provide a lightweight, robust core that can be extended in lots of different ways. Especially since the introduction of Starling 2.0, I think it has come very close to that goal. Thankfully, other developers quickly embraced this offer and made good use of Starling's extensibility!

Extensions Wiki

The Starling Wiki provides a respectable list of extensions of all types. Find it here: Starling Extensions.

If you ever decide to write your own extension (thanks in advance!), please take the extra effort to add it to this list. That way, it will be easy for other developers to find it, and you can add some form of documentation or sample code right away.

An extension doesn't have to be big and complex, by the way! If it consists of just a few lines of code that solve a common problem in a smart way, that's already reason enough to share it with the community.

I even wrote some of the extensions myself, providing features that I'm considering useful, but just not universal enough to be part of the main framework. I provide the same support for them as for the rest of the Starling code.

In this chapter, we will look at some of the most useful extensions, tools and libraries:

Particle System
: Create special effects like explosions, smoke, snow, fire, etc.

Dynamic Lighting
: Use normal maps for realistic three-dimensional lighting effects.

Texture Mask
: Mask a display object based on the alpha values of a texture.

The most exciting additions are saved for last: they are hugely popular among Starling users and are extremely extensive!

Feathers
: Provides an incredible number of user interface components to spice up your games or power complex business apps.

Starling Builder
: A visual editor allowing to create in-game user interfaces in a matter of minutes.

Chapter 8. Get your own copy now!

Thanks for looking through the preview of the *Starling Handbook*!

Did you like what you see? I'd be honored to see you get the full version. You won't regret it, I promise!

Get the full version here: gamua.com

Daniel Sperl
Vöcklabruck, Austria