Building GPU Accelerated Applications

Introducing

Starling

Adobe Developer Library

Thibault Imbert

「Introducing Starling Building GPU Accelerated Applications」

本文は http://shop.oreilly.com/product/0636920024217.do または http://www.bytearray.org/?p=3371 ページから無料で入手できる「Introducing Starling Building GPU Accelerated Applications」をヒム・カンパニー 永井勝則が自主的に日本語に訳したものです。

http://www.himco.jp/
knagai@himco.jp
(2012/1)

Starling の紹介

訳注：準備

本翻訳文書では Flash CS5.5 を使っています。Starling は Flash Player 11 と AIR 3 の機能を利用するので、これを Flash CS5.5 で使用できるようにするには、いくつかの設定が必要になります。以下はその手順と参考リンクです。

（Flash Player 11 を使用する場合）

●Flash Player 11 のダウンロードとブラウザへのインストール

・http://get.adobe.com/jp/flashplayer/

●Flash CS5.5 で Flash Player 11 用の SWF をパブリッシュできるようにする機能拡張のインストール

・http://cuaoar.jp/2011/11/-flash-professional-cs5.html

・http://cuaoar.jp/2011/11/flash-professional-cs5-c.html

・http://www.fumiononaka.com/TechNotes/Flash/FN1111001.html

●Flash CS5.5 での設定

・http://www.adobe.com/jp/devnet/flash/articles/stage3d.html

（AIR 3 を使用する場合）

●Flash CS5.5 で AIR SDK を使用する方法

・http://kb2.adobe.com/jp/cps/910/cpsid_91088.html

・アプリケーション記述ファイルの<renderMode>を手動で direct に指定する必要があります。

（両方で必要になる）

●Starling のダウンロードとインストール

・http://www.starling-framework.org/

このページでは、[Download]ボタンから直接ダウンロードせず、その下にある GitHub リンクからダウンロードする方が、新しいライブラリがダウンロードできます。本ドキュメントでは GitHub リンクからのダウンロードをおすすめします。FLA ファイルでは、そのソースパスに、Starling ライブラリの src フォルダを追加します。

本翻訳文書では AIR 3 を使用します。原書ではサンプルファイルとそれに使用する関連ファイルが提供されていないので、原書の記述にのっとり Flash CS5.5 の AIR 3 環境で再現したものを作成しました。サンプルファイルは http://www.himco.jp/test/StarlingSample.zip からダウンロードできます。

ファーストフライト

Starlingとは?

Starling(ムクドリの意)は Stage3D API 上で開発された ActionScript 3 の 2D フレームワークです(デスクトップ版の Flash Player と Adobe AIR 3 で使用できます)。Starling は主にゲーム開発向けに設計されていますが、ほかの多くの場合にも活用できます。Starling によって、低レベルの Stage3D API に触れることなく、高速な GPU アクセラレートの効いたアプリケーションの記述が可能になります。

Starlingを使う理由

Flash デベロッパーのほとんどは、GPU アクセラレーション(Stage3D による)を利用するとき、低レベルの Stage3D API に踏み込まず、もっと高レベルのフレームワークを使ってアプリケーションを記述したいでしょう。Starling は Flash Player API の後に設計され、Stage3D(Molehill)の複雑性を取り除いているので、容易で直感的なプログラミングが可能になります。

Starling は ActionScript 3 の、特に 2D ゲーム開発に携わるデベロッパー向けのライブラリなので、ActionScript 3 に関する基本的な理解が必要になります。Starling は、その設計(軽量で柔軟、かつ単純)によって、UI プログラミングのような場合にも使用できます。Starling は可能な限り直感的に設計されているので、Java や.Net デベロッパーでも時間をかけずに理解することができます。

理念

直感的

Starling の学習は容易です。Starling は ActionScript のほとんどの基本概念を踏襲しつつ、低レベルの Stage API の複雑性を取り除いているので、Flash/Flex デベロッパーはすぐになじむことができます。頂点バッファや透視行列、シェーダプログラムといった概念に対するコードを記述するのではなく、DOM 表示リストやイベントモデル、MovieClip や Sprite、TextField といったよく知った API など、なじみある概念が使用できます。

軽量

Starling はいろいろな意味で軽量な鳥です。クラスの数は限定的です(コードはわずか 80k ほどです)。Flash Player 11 や AIR 3(モバイルは今後のリリースでサポートされます)のほかには外部への依存性はなく、アプリケーションの小ささとワークフローの単純性を保つことができます。

自由

Starling は自由に生きています。Simplified BSD ライセンス下で規定されているので、商用アプリケーションであっても自由に使用できます。われわれは日夜 Starling に取り組み、コミュニティが精力的にさらに向上させてくれることを願っています。

仕組み

Starling は舞台裏で、Stage3D API を使用します。これは、デスクトップでは OpenGL と DirectX の上で、モバイルデバイスでは OpenGL ES2 の上で動作する低レベルの GPU API です。みなさんはデベロッパーとして、Starling が Sparrow（http://www.sparrow-framework.org/）の ActionScript 3 への移植版であることを知っておく必要があります。Sparrow は OpenGL ES2 API に依存する、iOS のライブラリです（図１参照）。

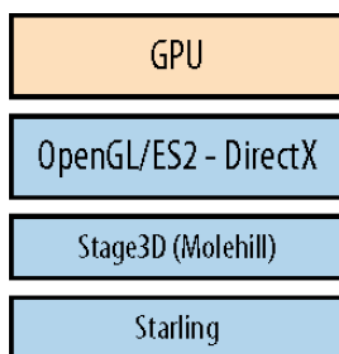


図１：Starling レイヤーは Stage3D（molehill）の上にある

Starling は、Flash デベロッパーがすでに慣れ親しんでいる多くの API を再作成します。図2は、Starling がグラフィック要素に関して公開する API を示しています。

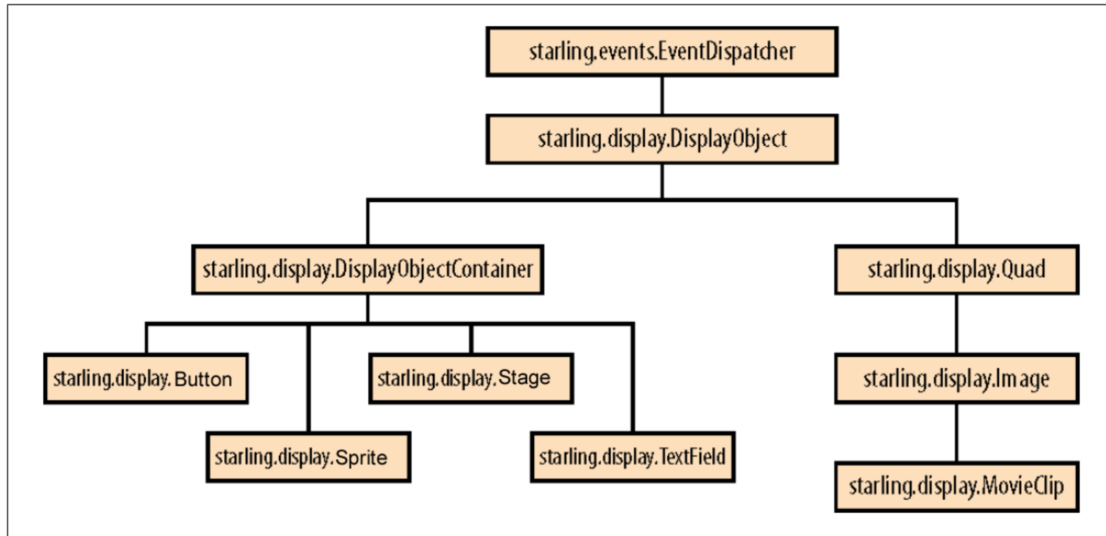


図2：表示オブジェクトの継承

多くのみなさんは、Stage3D API は 3D コンテンツに厳密に限定されていると思われるでしょう（その名前が混乱の元です）が、実際には 2D コンテンツも作成できます。

図3はこの考えを示しています。MovieClip のようなものは drawTriangles API でどのように描画されるのでしょう？

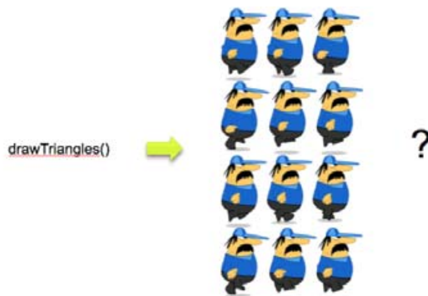


図3：drawTriangles と 2D？

実はこれはいたって単純です。GPU は三角形を描画するときに最も効率が高くなるので、drawTriangles API で三角形を2つ描画し、テクスチャをサンプリングして、それを UV マップを使って2つの三角形に適用します。するとテクスチャのついた四角形ができあがります。これがわれわれのスプライト（図3の人物の絵）です。毎フレーム、三角形のテクスチャを更新すると、それが MovieClip（図3の一連の人物の絵）になります。実にクールでしょ？

さらに、Starling を使用するときには、こういった詳細を気にかける必要はありません。われわれはただ、フレ

ームをこしらえ、それを Starling の MovieClip に与えるだけでよいのです（図4参照）。

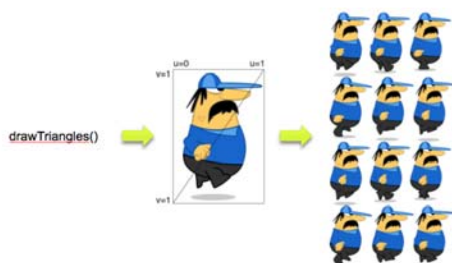


図4：drawTriangles ＋ テクスチャをつけた四角形 ＝ 2D

Starling がこの複雑性をどのように低減しているかを理解するため、テクスチャをつけた単純な四角形表示で、低レベルの Stage3D API を使わなければならないとしたら、どのようなコードになるかを見てみましょう。

```
// メインのタイムライン
import flash.display3D.Context3D;
import flash.display3D.Context3DProgramType;
import flash.display3D.Context3DRenderMode;
import flash.display3D.Context3DTriangleFace;
import flash.display3D.Context3DVertexBufferFormat;
import flash.display3D.IndexBuffer3D;
import flash.display3D.Program3D;
import flash.display3D.VertexBuffer3D;
import flash.display3D.textures.Texture;
import flash.display3D.Context3DTextureFormat;

import com.adobe.utils.AGALMiniAssembler;

// Stage3D の取得
var stage3D:Stage3D = stage.stage3Ds[0];

// Context3D は、Stage3D のインスタンスが作成、初期化する。
// 時間がかかるので非同期的に見張る
stage3D.addEventListener( Event.CONTEXT3D_CREATE, contextCreated );
// Context3D インスタンス作成を命令する
stage3D.requestContext3D( Context3DRenderMode.AUTO );
```

```
var context3D:Context3D;

// Context3D が作成されたら呼び出される
function contextCreated(event:Event):void
{
	// Context3D への参照を取得
	context3D = Stage3D(event.target).context3D;
	// 表示バッファを初期化
	context3D.configureBackBuffer(550, 400, 2, false);
	difficultWorks()
}

function difficultWorks() {
	// 頂点データの作成
	var vertices:Vector.<Number> = Vector.<Number>([
		-0.5,-0.5,0,	0,	0, // x, y, z, u, v
		-0.5,	0.5,	0,	0,	1,
		0.5,	0.5,	0,	1,	1,
		0.5,	-0.5,	0,	1, 0]);

	// 頂点バッファの作成（5つの値を持つ4つの頂点用バッファ）
	var vertexbuffer:VertexBuffer3D = context3D.createVertexBuffer(4,5);
	// 頂点バッファを GPU にアップロード（0番めから4個の頂点をアップロード）
	vertexbuffer.uploadFromVector(vertices, 0, 4);
	// インデックスバッファの作成（2つの三角形を作るので、インデックスは6個いる）
	var indexbuffer:IndexBuffer3D = context3D.createIndexBuffer(6);
	// 三角形の頂点の並び
	var indexData:Vector.<uint> = Vector.<uint>([
			0, 1, 2, // 1つめの三角形の頂点の並び
			2, 3, 0	// 2つめの三角形の頂点の並び
		]);

	// インデックスバッファを GPU にアップロード（0 番めから開始し、6個送る）
	indexbuffer.uploadFromVector(indexData, 0, 6);

	// ビットマップの作成
```

```
var bitmap:Bitmap = new Bitmap(new TextureBitmap());
// ビットマップをアップロードするテクスチャビットマップの作成
var texture:Texture = context3D.createTexture(bitmap.bitmapData.width,
                        bitmap.bitmapData.height, Context3DTextureFormat.BGRA, false);
// ビットマップのアップロード
texture.uploadFromBitmapData(bitmap.bitmapData);

// ミニアセンブラの作成
var vertexShaderAssembler : AGALMiniAssembler = new AGALMiniAssembler();
// 頂点シェーダのアセンブル（コンパイル）
vertexShaderAssembler.assemble( Context3DProgramType.VERTEX,
                                    "m44 op, va0, vc0¥n" +
                                    "mov v0, va1");

var fragmentShaderAssembler:AGALMiniAssembler = new AGALMiniAssembler();
// 断片シェーダのアセンブル
fragmentShaderAssembler.assemble( Context3DProgramType.FRAGMENT,
                                    "tex ft1, v0, fs0 <2d,linear, nomip>;¥n" +
                                    "mov oc, ft1");

// シェーダプログラムの作成
var program:Program3D = context3D.createProgram();
// 頂点シェーダと断片シェーダを GPU にアップロード
program.upload( vertexShaderAssembler.agalcode, fragmentShaderAssembler.agalcode);

// バッファをクリア
context3D.clear( 1, 1, 1, 1 );
// 頂点バッファを設定
// 頂点データの0番めの属性は3つの浮動小数点数（三角形の頂点の座標）
context3D.setVertexBufferAt(0, vertexbuffer, 0, Context3DVertexBufferFormat.FLOAT_3);
// 頂点データの1番めの属性は2つの浮動小数点数（三角形の UV 座標）
context3D.setVertexBufferAt(1, vertexbuffer, 3, Context3DVertexBufferFormat.FLOAT_2);

// テクスチャの設定
context3D.setTextureAt( 0, texture );
// シェーダプログラムの設定
```

```
    context3D.setProgram( program );

    // 3D 行列の作成
    var m:Matrix3D = new Matrix3D();
    // 頂点が Z 軸回りに回転するよう、行列に回転を適用
    m.appendRotation(getTimer()/50, Vector3D.Z_AXIS);
    // プログラム定数を設定（ここでは行列）
    context3D.setProgramConstantsFromMatrix(Context3DProgramType.VERTEX, 0, m, true);
    // 三角形を描画
    context3D.drawTriangles(indexbuffer);
    // バックレンダリングバッファを表示
    context3D.present();
}
```

訳注：

これは Flash のフレームスクリプトです。ここで使用している AGALMiniAssembler クラスは https://github.com/Barliesque/EasyAGAL からダウンロードできます。

頂点データとして作成している vertices は次の表のように考えることができます。x、y、z は3次元の座標値で、u と v は UV マップの座標値です。z はすべて 0 なので、頂点は全部2次元の xy 平面上にあります。

x	y	z	u	v
−0.5	−0.5	0	0	0
−0.5	0.5	0	0	1
0.5	0.5	0	1	1
0.5	−0.5	0	1	0

この表の値を図で表すと、図 4-1 のようになります。頂点の座標は(−0.5, −0.5)から始まり、時計回りに(−0.5, 0.5)、(0.5, 0.5)、(0.5, −0.5)と進み、UV 座標も同様に(0, 0)に始まり、時計回りに(0, 1)、(1, 1)、(1, 0)と進みます。頂点の座標と UV 座標はそれぞれ対応しています。UV 座標は正方形のテクスチャに割り当てられ、テクスチャの左下隅(0, 0)は点(−0.5、−0.5)に、テクスチャの左上隅（0, 1）は点(−0.5, 0.5)に、テクスチャの右上隅(1, 1)は点(0.5、0.5)に、テクスチャの右下隅（1, 0）は点(0.5, −0.5)に”吸着”します。これにより 3D 空間にテクスチャづけされた物体が登場することになります。

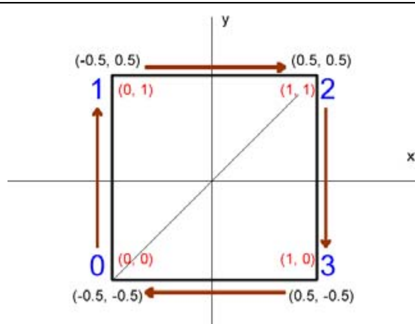


図 4-1：2つの三角形の頂点情報

ただしまだ、これらの頂点からどのような三角形を形作るかを指定する必要があります。それを指定するのが indexData です。0, 1, 2 は1つめの三角形の頂点の順番で、図 4-1 でも 0→1→2 と進んで、三角形を形成します。2, 3, 0 は2つめの三角形の頂点の順番で、図 4-1 でも 2→3→1 と進んで、もう1つの三角形を形成します。斜辺 2-0 は2つの三角形に共通する辺で、これにより2つの三角形から1つの四角形ができあがります。

```
// 三角形の頂点の並び
    var indexData:Vector.<uint> = Vector.<uint>([
            0, 1, 2, // 1つめの三角形の頂点の並び
            2, 3, 0  // 2つめの三角形の頂点の並び
        ]);
```

ここで使われている Stage3D API の詳しい説明は、http://cuaoar.jp/2011/11/stage3d.html、http://cuaoar.jp/2011/11/stage3d-2.html 、 http://cuaoar.jp/2011/11/stage3d-agal.html 、http://cuaoar.jp/2011/11/stage3d-context3d.html で読むことができます。

これを実行すると図5に示す結果を得ることができます。

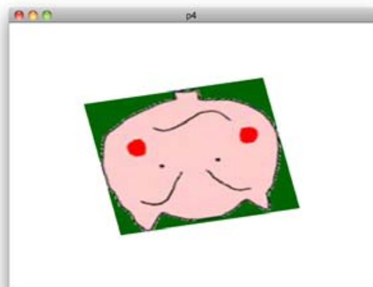


図5：テクスチャをつけた単純な四角形

これはずいぶん複雑なコードですよねぇ？　低レベルの API にアクセスしなければならないので大変です。制御できることは多くありますが、なにぶん低レベルなのです。

Starling では、次のようなコードで済みます。

```
// 埋め込んだビットマップから Texture オブジェクトを作成
var texture:Texture = Texture.fromBitmap ( new embeddedBitmap() );
// Texture から Image オブジェクトを作成
var image:Image = new Image(texture);
// プロパティを設定
image.pivotX = 50;
image.pivotY = 50;
image.x = 300;
image.y = 150;
image.rotation = Math.PI/4;
// これを表示
addChild(image);
```

ActionScript 3 デベロッパーは Flash API に慣れているので、こういった API にもすぐに習熟できます。Stage3D API の複雑性は全部、舞台裏で処理されます。

再描画領域機能を使おうとすると、Starling は予想されるように、従来の表示リストではなく、Stage3D 上にレンダリングします。図6はその振る舞いを示しています。ここでは毎フレーム、四角形を回転させていますが、再描画領域に表示されるのは、表示リストに置いた FPS カウンタだけです（CPU で実行されます）。

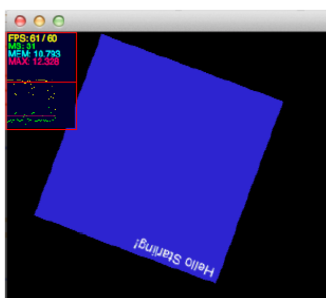


図6：コンテンツは Stage3D を通してレンダリングされる

Stage3D のアーキテクチャでは、コンテンツは GPU によってレンダリングされ合成されるということを覚えておいてください。その結果として、表示リストで使用される再描画領域機能は利用できません。

階層関係による制約

Starling を使用するデベロッパーとしてみなさんは、コンテンツを開発するときには1つの制限があることを覚えておく必要があります。前述したように、Stage3D は Flash Player 内の新しいレンダリングアーキテクチャで、GPU サーフェスは表示リストの下に置かれます。これは、表示リストの中で実行されるコンテンツはすべて、Stage3D の上に置かれるということです。表示リストで実行されるコンテンツが Stage3D レイヤーの下に置かれることは決してありません。

図7はその階層関係を示しています。

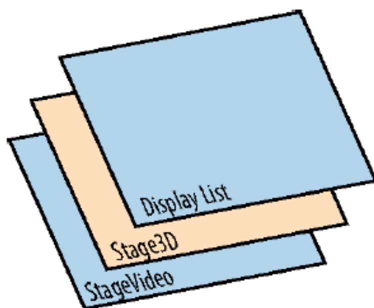


図7：StageVideo の上に Stage3D があり、その上に表示リストがある

Stage3D オブジェクトは透明にできません。もしそういった機能があれば、StageVideo を使ってビデオを再生し、そのビデオの上に Stage3D を通してレンダリングしたコンテンツを重ねることも可能になるでしょう。将来の Flash Player と AIR のリリースでは可能になるかもしれません。

スタート

Starling をダウンロードするには、次のリンクを調べてください。

- 公式の Starling Github: http://github.com/PrimaryFeather/Starling-Framework/
- 公式の Starling Web サイト: http://www.starling-framework.org

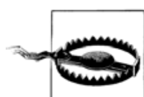
Starling は Simplified BSD ライセンスで許諾されているので、どのようなプロジェクトでも自由に使用できます。Starling フレームワークの開発者には office@starling-framework.org.から連絡を取ることができます。

ダウンロードしたら、通常の AS3 ライブラリと同じ方法で Starling ライブラリを参照します。Starling が必要とする新しい Stage3D API を使用するには、追加的なコンパイラ引数を Flex コンパイラに渡すことで、SWF

バージョン 13 をターゲットにする必要があります（-swf-version=13）。

シーンの設定

前置きはもう十分です。ここからはコードを探り、この小さな鳥に何ができるのかを見ていきましょう。Starling の設定は簡単です。まず Starling オブジェクトを作成し、それをメインクラスに追加します。

これ以降、MovieClip や Sprite といったオブジェクトを述べるときには、それは Starling API のことで、特段の説明がない限り、Flash Player ネイティブのオブジェクトを指していません。

Starling のコンストラクタは、複数の引数を期待します。以下がその引数です。

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

とは言え、通常使用されるのは初めの２つ（rootClass と stage）だけです。rootClass は starling.display.Sprite を拡張するクラスへの参照で、stage はステージです。

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import starling.core.Starling;
    import net.hires.debug.Stats

    [SWF(width = "1280",height = "752",frameRate = "60",backgroundColor = "#002143")]

    public class Startup extends Sprite
    {

        private var mStarling:Starling;
```

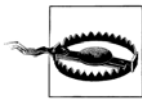
```
		public function Startup()
		{
			// fps 計測用クラス
			addChild (new Stats());

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

			// Starling インスタンスの作成
			mStarling = new Starling(Game, stage);
			//  アンチエイリアスの設定（大きいほど品質はアップするが、パフォーマンスは落ちる）
			mStarling.antiAliasing = 1;
			//  スタート！
			mStarling.start();
		}
	}
}
```

以下は、ステージに追加されるとき単純な四角形を作成する Game クラスです。

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

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

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

		private function onAdded ( e:Event ):void
```

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

いつも行っているように、Event.ADDED_TO_STAGE イベントを監視して、そのイベントハンドラでアプリケーションを初期化しています。このようにすることで、ステージに安全にアクセスすることができます。

再度述べておきますが、次の細かな点には注意してください。この Game クラスが拡張しているのは starling.display パッケージの Sprite クラスで、flash.display パッケージのそれではありません。import ステートメントをつねに調べ、ネイティブのものではなく、必ず Starling API を使用するよう気をつけることは良いプラクティスです。これにはすぐに慣れることができますが、最初のうちは戸惑いがあるかもしれません。

訳注:
FLA ファイルでは、Starling の src フォルダと Stats の src フォルダをソースパスに追加する必要があります。Stats は https://github.com/mrdoob/Hi-ReS-Stats からダウンロードできます。また AIR 3 で実行する場合には、忘れずに<renderMode>を direct に設定します。

このコードを実行すると、ステージ左上隅に四角形が現われます。これをステージ中央に持ってくるには次のコードを追加します。

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

右への 1 ビットシフト(>> 1)は 2 で割ることと同じで、この方が計算が高速で行われます。図 8 はここまでの結果を示しています。

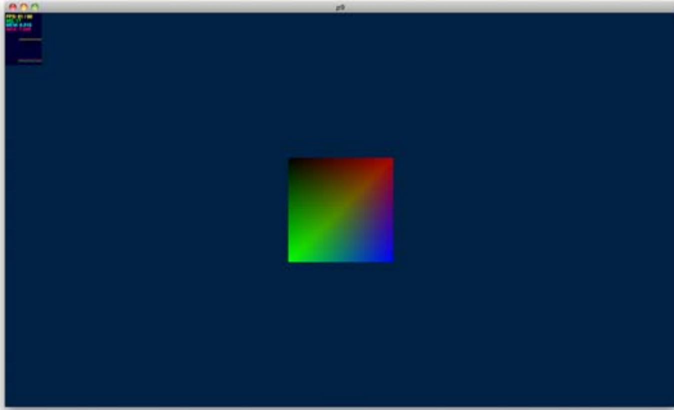


図8：初めての四角形

antiAliasing の値を使用すると、希望するアンチエイリアス処理を設定することができます。一般的には 1 の値で十分ですが、もっと大きくすることもできます。技術的には 0 から 16 までの値が指定できます。以下の値がよく使用されます。

- 0: アンチエイリアスなし
- 2: 最小限のアンチエイリアス
- 4: 高品質のアンチエイリアス
- 16: 最高品質のアンチエイリアス

2 より大きな値は、特に 2D コンテンツではめったに必要ありませんが、それはみなさんが使用ケースに応じて決めることです。図9では、1 と 4 を指定した場合の微妙な違いを示しています。

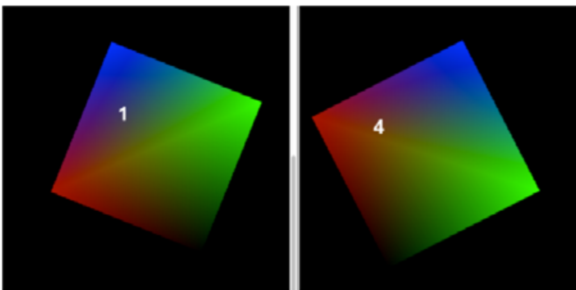


図9：アンチエイリアス処理の違い

希望する品質を調整するために 2 以上の値を試してみることもできますが、大きな値の選択は当然のことながら、パフォーマンスに相応の影響を及ぼします。

では Starling オブジェクトで使用できるそのほかの API について、ざっと見ておきましょう。

・enableErrorChecking：エラーチェックの有効化と無効化が行えます。レンダラが遭遇したエラーをアプリケーションに報告するかどうかを指定します。enableErrorChecking が true のときには、Starling によって内部的に呼び出される clear()と drawTriangles()メソッドは同期的で、エラーをスローします。enableErrorChecking が false のときには（デフォルト）、clear()と drawTriangles()メソッドは非同期的で、エラーは報告されません。エラーチェックを有効化するとレンダリングのパフォーマンスが落ちるので、その有効化はデバッグ時のみに限るべきです。

・isStarted：start()が呼び出されたかどうかを示します。

・juggler：juggler は単純なオブジェクトで、IAnimatable を実装するオブジェクトのリストを保存し、オブジェクトの時間を進めるように命令された場合に（juggler 自体の advanceTime()メソッドを呼び出すことで）、そのようにします。アニメーションが完了すると、juggler はこれを破棄します。

・start()：レンダリングとイベント処理を開始します。

・stop()：レンダリングとイベント処理を停止します。このメソッドは、ゲームがバックグラウンドに移行しリソースを保存するとき、レンダリングの一時停止に使用できます。

・dispose()：レンダリング中で GPU のメモリに現在存在するコンテンツすべてを破棄したいときに呼び出します。この API は内部で、シェーダプログラムとリスナーを破棄します。

Starling オブジェクトが作成されると、使用されるレンダラの情報を表示するデバッグトレースが自動的に出力されます。デフォルトでは、SWF がページに埋め込まれているか、スタンドアロンでテスト中のときには、Starling は以下を出力します。

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

もちろんハードウェアの詳細はみなさんの設定によって変わります。ここでは、ドライバのバージョンの詳細情報を確認することで、GPU アクセラレーションを使用していることを知らせています。デバッグ目的の場合には、Flash Player が内部で使用するソフトウェアフォールバック（ソフトウェアで処理する代替方法）を使用し、ソフトウェアでの実行でコンテンツのパフォーマンスへの影響を調べることができます。

そのためには、ソフトウェアフォールバック（ソフトウェアラスタライザ）を使用することを、次のように伝えます。

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

ソフトウェアを使用すると、出力メッセージは、ソフトウェアモードで実行中であることを伝えます。

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

ソフトウェアモードで実行するユーザーも想定する場合には、そのパフォーマンスを把握するために、コンテンツソフトウェアモードでテストする方がよいでしょう。グラフィックドライバが 1/1/2009 以前のものである場合には、コンテンツはソフトウェアで代替実行されます。ただしこれは今後の Flash Player と AIR のバージョンではもっと緩和される可能性があります。では、SWF をページに埋め込むときの Stage3D の要件を見ておきましょう。

wmode の要件

Stage3D と GPU のアクセラレーションを有効にするには、ページの埋め込みモードとして wmode=direct を使用する必要があります。値を指定しないか、"direct" 以外の値（"transparent" や "opaque"、"window"）を選ぶと、Stage3D は有効になりません。Stage3D 上で requestContext3D を呼び出すとき、Context3D オブジェクトの作成に失敗したことを伝えるランタイム例外が発生します。図 10 に示すランタイム例外ダイアログボックスが表示されます。

図 10：Context3D が使用できないときのランタイム例外

アプリケーションが間違った wmode で埋め込まれている場合、この状況の処理は重要で、この問題を説明するメッセージを表示して、適切に応答する必要があります。Starling はありがたいことにこれを自動的に処理し、図 11 の上に示す英文メッセージを表示します。このメッセージは、Starling のソースコードを修正して変更してもかまいません。見栄えをカスタマイズすることも自由です。

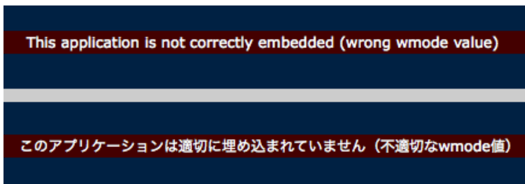


図 11：アプリケーションが適切に埋め込まれていないときの警告メッセージ

訳注：
図 11 の下は英文を日本語に変えたものです。これは starling/core/Starling.as ファイルの 309 行めにある showFatalError()のストリング引数を変更することで、日本語に変えています。

ステージ品質

Flash デベロッパーのみなさんには、ステージ品質の概念は目新しいものではないでしょう。とは言え、重要なのは、Stage3D を扱うときや Starling を使うときには、ステージ品質はパフォーマンスに影響を与えないということです。

先進的な拡張機能

すでに述べたように、GPS のアクセラレーションが利用できないとき、Stage3D はソフトウェアに移行し、Transgaming 社の SwiftShader と呼ばれるソフトウェアフォールバック（代替）エンジンを内部的に使用します。この状況でもコンテンツが確実にうまく実行されるようにするには、ソフトウェアで実行されるタイミングを検出し、ソフトウェアの動作を遅くする可能性のある要因を取り除く必要があります。

2D コンテンツの場合には、ソフトウェアフォールバックでも多くのオブジェクトを処理し、すぐれたパフォーマンスを提供できるはずです。それでもなおこれを判定したいときには、Starling の静的プロパティ context を使って、Context3D オブジェクトにアクセスできます。

```
// 実行しているのはハードウェアかソフトウェアか？
var isHW:Boolean = Starling.context.driverInfo.toLowerCase().indexOf("software") == -1;
```

コンテンツに関し、つねにソフトウェアフォールバックのことまで気にかけて設計するのは、あらゆる設定において可能な限り最良の体験が提供されるので、良いプラクティスです。デベロッパーの中には、ハードウェアアクセラレーションのオン、オフを定期的に切り替えて Stage3D ゲームを開発する人もいます。こうすることで、ソフトウェア上で実行するときのゲーム体験も確実に向上させることができます。

次はエキサイティングなトピックである、Starling の表示リストについて見ていきましょう。

表示リスト

Starling はネイティブの Flash 表示リストと同じルールにしたがっています（たとえば、オブジェクトが表示リストに追加されるまで、ステージは有効になりません）。ステージに安全にアクセスするには通常、Flash の最も重要な次のイベントに依存します。これらは Starling でも使用できます。

・ Event.ADDED: オブジェクトが親に追加された
・ Event.ADDED_TO_STAGE: オブジェクトが、ステージにつながっている親に追加され、可視的になっている
・ Event.REMOVED: オブジェクトが親から削除された
・ Event.REMOVED_FROM_STAGE: オブジェクトが、ステージにつながっている親から削除され、不可視になっている

われわれは、以降のサンプルでこれらのイベントに積極的に依存します。Flash コンテンツと同様、これらのイベントを使用すると、オブジェクトの初期化や無効化、パフォーマンスやリソースの最適化に役立てることができます。

以下は DisplayObject クラスで定義されているメソッドのリストです。

・ removeFromParent: 存在する場合、親からオブジェクトを削除する
・ getTransformationMatrixToSpace: ローカル座標システムから別の座標システムへの変換を表す行列を作成する
・ getBounds: 別の座標システムでも見えるように、オブジェクトを完全に内包する矩形を返す
・ hitTestPoint: ローカル座標のある点で、最上位で見つかったオブジェクトを返す。テストが失敗した場合には null を返す
・ globalToLocal: グローバル（ステージ）座標からローカル座標システムに、点を変換する
・ localToGlobal:ローカル座標システムからグローバル（ステージ）座標に、点を変換する

以下は DisplayObject クラスで定義されているプロパティのリストです。これらはすべて公開されており、中にはネイティブの表示リストと比べて、少し向上しているものもあります。たとえば pivotX と pivotY を使うと、DisplayObject の基準点を動的に変更することができます。

・ transformationMatrix: 親を基準にした、オブジェクトの変換行列
・ bounds: 親のローカル座標を基準にした、オブジェクトの境界

・ width: オブジェクトの幅（ポイント単位）
・ height: オブジェクトの高さ（ポイント単位）
・ root: オブジェクトを含む表示ツリーの最上位にあるオブジェクト
・ x: 親のローカル座標を基準にした、オブジェクトの x 座標
・ y: 親のローカル座標を基準にした、オブジェクトの y 座標
・ pivotX: オブジェクト自体の座標空間内での、オブジェクトの原点の x 座標（デフォルトは 0）
・ pivotY: オブジェクト自体の座標空間内での、オブジェクトの原点の y 座標（デフォルトは 0）
・ scaleX: 水平方向の拡大縮小要因。1 は拡大縮小なしを意味し、負の値はオブジェクトを裏返しにすることを意味する
・ scaleY: 垂直方向の拡大縮小要因。1 は拡大縮小なしを意味し、負の値はオブジェクトを裏返しにすることを意味する
・ rotation: ラジアンタインのオブジェクトの回転（Sparrow では角度はすべてラジアン単位で計測される）
・ alpha: オブジェクトの不透明度
・ visible: オブジェクトの可視性。不可視のオブジェクトにはタッチできない
・ touchable: このオブジェクト（とその子）がタッチイベントを受け取るかどうかを示す
・ parent: この表示オブジェクトを含む表示オブジェクトコンテナ
・ stage: 表示オブジェクトがつながっているステージか、ステージにつながっていない場合には null

ネイティブの Flash API と同様、Sprite は再軽量のコンテナです。もちろん DisplayObject のサブクラスとして前述したすべての API のほか、コンテンツをネストする機能があります。ここまでシーン以外には、コンテンツのネストは行っていませんが、Game クラスは Sprite を拡張していたことを思い出してください。Sprite は DisplayObjectContainer です。

以下は DisplayObjectContainer クラスが公開する API です。

・ addChild: コンテナに子を追加する。子は最上位に来る
・ addChildAt: コンテナに子を特定のインデックスに追加する
・ dispose: GPU バッファとそのオブジェクトに登録されたすべてのリスナーを削除する
・ removeFromParent: 子をその親から削除する
・ removeChild: 子をコンテナから削除する。そのオブジェクトが子でない場合には何も起きない
・ removeChildAt: 特定のインデックスにある子を削除する。その子の上位にある（複数の）子は下位に移る
・ removeChildren: すべての子をそのコンテナから削除する
・ getChildAt: 特定のインデックスにある子オブジェクトを返す
・ getChildByName: 特定の名前を持つ子オブジェクトを返す（非再帰的）
・ getChildIndex: コンテナ内の子のインデックスを返す

・ setChildIndex: 指定された子のインデックスを変更する
・ swapChildren: 2つの子のインデックスを交換する
・ swapChildrenAt: 2つの子のインデックスを交換する
・ contains: 特定のオブジェクトがそのコンテナの子であるかどうかを調べる（再帰的）

ステージにアクセスできるようになったら、そこではほとんどの DisplayObjectContainer API を呼び出すことができ、さらにステージにカラーを渡すこともできます。Starling はデフォルトで、SWF のデフォルトの背景色を取り入れます。これは次の SWF タグで設定しています。

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

この振る舞いはオーバーライドできます。そのためにはただ stage オブジェクトにカラーを渡すだけです。stage には表示リストに追加した DisplayObject ならどれからでもアクセスできます。

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

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

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

		private function onAdded ( e:Event ):void
		{

			// 背景色を青に設定する
			stage.color = 0x002143;

			q = new Quad(200, 200);
```

```
				q.setVertexColor(0, 0x000000);
				q.setVertexColor(1, 0xAA0000);
				q.setVertexColor(2, 0x00FF00);
				q.setVertexColor(3, 0x0000FF);

				q.x = stage.stageWidth - q.width >> 1;
				q.y = stage.stageHeight - q.height >> 1;
				addChild ( q );
			}
		}
}
```

テクスチャはまだ使っていませんが、基本的には2つの三角形をグループ化して四角形にし、その平面の各頂点に、GPU で補間される異なるカラーを持たせている、ということを理解しておいてください。

もちろん、単色の平面がほしい場合には、Quad オブジェクトの color プロパティが使用できます。

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

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

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

		private function onAdded ( e:Event ):void
		{
			q = new Quad(200, 200);
			q.color = 0x00FF00;
```

```
            q.x = stage.stageWidth - q.width >> 1;
            q.y = stage.stageHeight - q.height >> 1;
            addChild ( q );
        }
    }
}
```

図 12 はこのコードの結果を示しています。

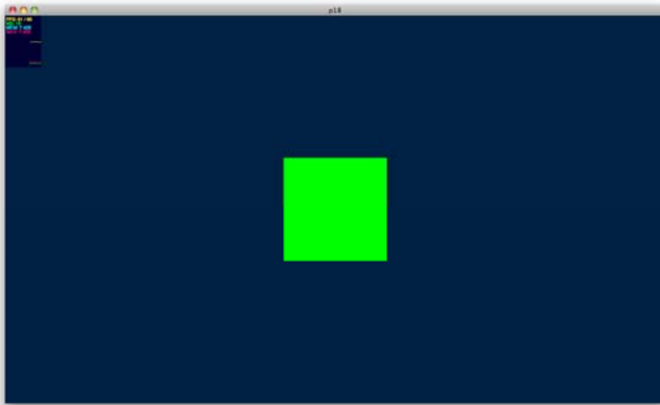


図 12：単色の緑の四角形

次は Event.ENTER_FRAME イベントを使ってみましょう。次のハンドラではランダムなカラー間での単純なイージング効果を使って、四角形のカラーを補間しています。

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

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

        private var r:Number = 0;
        private var g:Number = 0;
        private var b:Number = 0;
```

```
private var rDest:Number;
private var gDest:Number;
private var bDest:Number;

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

private function onAdded( e:Event ):void
{
      resetColors();

      q = new Quad(200,200);
      q.x = stage.stageWidth - q.width >> 1;
      q.y = stage.stageHeight - q.height >> 1;

      addChild( q );

      addEventListener(Event.ENTER_FRAME, onFrame);
}

private function onFrame(e:Event):void
{
      r -=  (r - rDest) * .01;
      g -=  (g - gDest) * .01;
      b -=  (b - bDest) * .01;

      var color:uint = r << 16 | g << 8 | b;
      q.color = color;

      // カラー値に届いたら、別のカラーに変える
      if ( Math.abs( r - rDest) < 1 && Math.abs( g - gDest) < 1 && Math.abs( b - bDest) )
      {
            resetColors();
```

```
				}
			}

			private function resetColors():void
			{
				rDest = Math.random() * 255;
				gDest = Math.random() * 255;
				bDest = Math.random() * 255;
			}
		}
}
```

この四角形の回転には rotation プロパティが使用できます。ただし、Flash Player は度で動作しますが、Starling ではラジアンを使用する点に注意してください。これは Sparrow と Starling 間の一貫性を保つためになされた選択です。度を使って回転を適用したいときにはつねに、starling.utils.deg2rad()関数を使うか、インラインで変換します。

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

それでもなお度を使いたい場合には、Starling のソースコードを自由に変更してください。またすべての DisplayObject は、実行時にその基準点が移動できる pivotX と pivotY プロパティを持っています。

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

Starling を使用する ActionScript デベロッパーには、この四角形は DisplayObject として、Sprite 内に TextField といっしょにネストでき、そのスプライトを移動させるとその要素もグループとしていっしょに移動すると聞いても、当たり前だと思われるでしょう。これはネイティブの表示リストとまったく同じです。

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

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

	private var r:Number = 0;
	private var g:Number = 0;
	private var b:Number = 0;

	private var rDest:Number;
	private var gDest:Number;
	private var bDest:Number;

	public function Game()
	{
		addEventListener(Event.ADDED_TO_STAGE, onAdded);
	}
	private function onAdded( e:Event ):void
	{
		resetColors();

		q = new Quad(200,200);
		s = new Sprite();
		var legend:TextField = new TextField(100, 20, "Hello Starling!", "Arial", 14,
0xFFFFFF);
		s.addChild(q);
		s.addChild(legend);

		s.pivotX = s.width >> 1;
		s.pivotY = s.height >> 1;

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

		addChild(s);
```

```
            s.addEventListener(Event.ENTER_FRAME, onFrame);
        }

        private function onFrame(e:Event):void
        {
            r -=  (r - rDest) * .01;
            g -=  (g - gDest) * .01;
            b -=  (b - bDest) * .01;

            var color:uint = r << 16 | g << 8 | b;
            q.color = color;

            if ( Math.abs( r - rDest) < 1 && Math.abs( g - gDest) < 1 && Math.abs( b - bDest) )
                resetColors();
            (e.currentTarget as DisplayObject).rotation += .01;
        }

        private function resetColors():void
        {
            rDest = Math.random() * 255;
            gDest = Math.random() * 255;
            bDest = Math.random() * 255;
        }
    }
}
```

ここでは、Quad と TextField を含むコンテナスプライトを、その基準点を中心に回転させています（図 13 参照）。

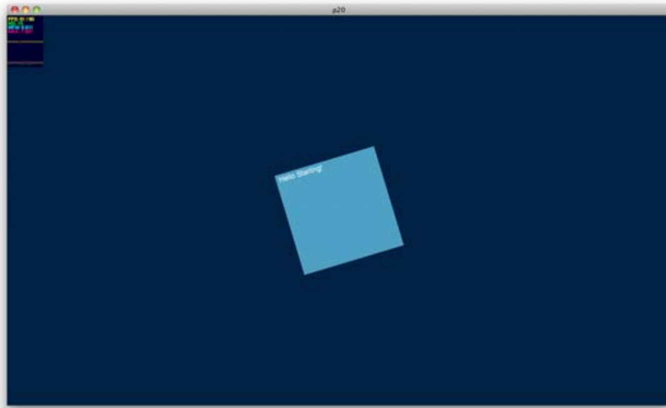


図 13：Quad と TextField を含んだ Sprite

コードがいささかごちゃごちゃしてきたので、CustomSprite を定義してそこにコードを移しましょう。これは内部的なカラーの振る舞いと子をカプセル化するクラスです。そのコードは次のようになります。

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

	public class CustomSprite extends Sprite
	{
		private var quad:Quad;
		private var legend:TextField;

		private var quadWidth:uint;
		private var quadHeight:uint;

		private var r:Number = 0;
		private var g:Number = 0;
		private var b:Number = 0;

		private var rDest:Number;
		private var gDest:Number;
		private var bDest:Number;
```

```
public function CustomSprite(width:Number, height:Number, color:uint=16777215)
{
	// カラー成分の値をリセット
	resetColors();

	// 幅と高さを設定
	quadWidth = width;
	quadHeight = height;

	// ステージ追加されたら、有効化する
	addEventListener(Event.ADDED_TO_STAGE, activate);
}

private function activate(e:Event):void
{
	// 指定されたサイズで四角形を作成
	quad = new Quad(quadWidth,quadHeight);

	// レジェンドを追加
	legend = new TextField(100,20,"Hello Starling!","Arial",14,0xFFFFFF);

	// 子を追加
	addChild(quad);
	addChild(legend);

	// 基準点を変更
	pivotX = width >> 1;
	pivotY = height >> 1;
}

private function resetColors():void
{
	// ランダムなカラー成分を選定
	rDest = Math.random() * 255;
	gDest = Math.random() * 255;
```

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

			/**
			* 内部的な振る舞いを更新
			*
			*/
			public function update():void
			{
				// 成分のイージング
				r -=  (r - rDest) * .01;
				g -=  (g - gDest) * .01;
				b -=  (b - bDest) * .01;

				// カラーを組み合わせる
				var color:uint = r << 16 | g << 8 | b;
				quad.color = color;

				// カラー値に届いたら、別のカラーを選定する
				if ( Math.abs( r - rDest) < 1 && Math.abs( g - gDest) < 1 && Math.abs( b - bDest) )
					resetColors();
				// 回転させる！
				// rotation += .01;
			}
		}
}
```

カスタムスプライトの更新は CustomSprite の update API で行います。これはオブジェクトを制御する司令塔の Game クラスのメインループから呼び出します。ほかの要素に対してもこの方法を使うことで、コンテンツ全体に及ぶ一時停止メカニズムを追加するときにも分かりやすくなります。

つづけてこの小さなテストにインタラクションを加えていきましょう。次の例では四角形にマウスを追随させる動きを追加します。これはコードを少し加えるだけで実現できます。

```
package
{
```

```
	import flash.geom.Point;

	import starling.display.Sprite;
	import starling.events.Event;
	import starling.events.Touch;
	import starling.events.TouchEvent;

	public class Game extends Sprite
	{
		private var customSprite:CustomSprite;
		private var mouseX:Number = 0;
		private var mouseY:Number = 0;

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

		private function onAdded( e:Event ):void
		{
			// カスタムスプライトを作成
			customSprite = new CustomSprite(200,200);

			// 位置はデフォルトでステージの中央
			// カスタムスプライトの基準点はセンターにあるので、半分だけ追加する
			customSprite.x = (stage.stageWidth - customSprite.width >> 1 ) + 
(customSprite.width >> 1);
			customSprite.y = (stage.stageHeight - customSprite.height >> 1) + 
(customSprite.height >> 1);

			// 表示する
			addChild(customSprite);

			// ステージ上でのマウスの動きを監視
			stage.addEventListener(TouchEvent.TOUCH, onTouch);
			stage.addEventListener(Event.ENTER_FRAME, onFrame);
```

```
		}

		private function onFrame(e:Event):void
		{
			// カスタムスプライトの位置をイージング
			customSprite.x -= ( customSprite.x - mouseX ) * .1;
			customSprite.y -= ( customSprite.y - mouseY ) * .1;

			// カスタムスプライトの更新
			customSprite.update();
		}

		private function onTouch (e:TouchEvent):void
		{
			// ステージを基準にしたマウス位置を取得
			var touch:Touch = e.getTouch(stage);
			var pos:Point = touch.getLocation(stage);
			// マウス座標を保持
			mouseX = pos.x;
			mouseY = pos.y;
		}
	}
}
```

ここでは Mouse API を使っていないことに注目してください。Starling には実際、マウスの概念はありません。これについてはこの後、すぐに触れます。

TouchEvent.TOUCH イベントの監視によって、従来の MousEvent.MOUSE_MOVE と同じように、実際にはマウスや指の動きが監視できます。TouchEvent.TOUCH イベントが発生するたびに呼び出される onTouch イベントハンドラでは、getTouch や getLocation といった TouchEvent オブジェクトのヘルパーAPI を使って、そのときのマウス位置を保持しています。onFrame イベントハンドラでは簡単なイージング式を使って、四角形を移動させています。

すでに述べているように、Starling はみなさんの GPU プログラミングを容易にするだけでなく、リソースのためにオブジェクトを破棄するときにも力を発揮します。たとえば四角形のクリックでこの四角形をシーンから削除したい場合には、コードは次のようになります。

```
package
{
	import flash.geom.Point;

	import starling.display.DisplayObject;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.events.Touch;
	import starling.events.TouchEvent;
	import starling.events.TouchPhase;

	public class Game extends Sprite
	{
		private var customSprite:CustomSprite;
		private var mouseX:Number = 0;
		private var mouseY:Number = 0;

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

		private function onAdded( e:Event ):void
		{
			// カスタムスプライトを作成
			customSprite = new CustomSprite(200,200);

			// 位置はデフォルトでステージの中央
			// カスタムスプライトの基準点はセンターにあるので、半分だけ追加する
			customSprite.x = (stage.stageWidth - customSprite.width >> 1 ) + 
(customSprite.width >> 1);
			customSprite.y = (stage.stageHeight - customSprite.height >> 1) + 
(customSprite.height >> 1);

			// 表示する
```

```
        addChild(customSprite);

        // ステージ上でのマウスの動きを監視
        stage.addEventListener(TouchEvent.TOUCH, onTouch);
        stage.addEventListener(Event.ENTER_FRAME, onFrame);
        // スプライトがタッチされたら
        customSprite.addEventListener(TouchEvent.TOUCH, onTouchedSprite);
    }

    private function onFrame(e:Event):void
    {
        // カスタムスプライトの位置をイージング
        customSprite.x -= ( customSprite.x - mouseX ) * .1;
        customSprite.y -= ( customSprite.y - mouseY ) * .1;

        // カスタムスプライトの更新
        customSprite.update();
    }

    private function onTouch (e:TouchEvent):void
    {
        // ステージを基準にしたマウス位置を取得
        var touch:Touch = e.getTouch(stage);
        var pos:Point = touch.getLocation(stage);
        // マウス座標を保持
        mouseX = pos.x;
        mouseY = pos.y;
    }

    private function onTouchedSprite(e:TouchEvent):void
    {
        // タッチポイントを取得（マルチタッチなので複数タッチも可能）
        var touch:Touch = e.getTouch(stage);
        var clicked:DisplayObject = e.currentTarget as DisplayObject;

        // クリックかリリース段階かを判定
```

```
            if ( touch.phase == TouchPhase.ENDED )
            {
                // クリックされたオブジェクトを削除
                removeChild(clicked);
            }
        }
    }
}
```

ここでは子は削除していますが、Event.ENTER_FRAME リスナーは削除していません。スプライトが依然としてリスナーを持っているかどうかは、hasEventListener API で調べることができます。

```
private function onTouchedSprite(e:TouchEvent):void
{
    // タッチポイントを取得（マルチタッチなので複数タッチも可能）
    var touch:Touch = e.getTouch(stage);
    var clicked:DisplayObject = e.currentTarget as DisplayObject;

    // クリックかリリース段階かを判定
    if ( touch.phase == TouchPhase.ENDED )
    {
        // クリックされたオブジェクトを削除
        removeChild(clicked);
        // 出力 : true
        trace ( clicked.hasEventListener(e.type) );
    }
}
```

子を安全に削除するには、removeChild API の2つめの dispose パラメータを使用します。dispose を使うと、そのオブジェクトに登録されたすべてのリスナーも自動的に削除することができます。

```
private function onTouchedSprite(e:TouchEvent):void
{
    // タッチポイントを取得（マルチタッチなので複数タッチも可能）
    var touch:Touch = e.getTouch(stage);
    var clicked:DisplayObject = e.currentTarget as DisplayObject;
```

```
        // クリックかリリース段階かを判定
        if ( touch.phase == TouchPhase.ENDED )
        {
                // 全リスナーも削除して破棄
                removeChild(clicked, true);
                // 出力 : false
                trace ( clicked.hasEventListener(e.type) );
        }
}
```

子がさらに（複数の）子を持っている場合には、そのすべての子も破棄されます。この dispose 引数はまた、removeChildren や removeChildAt といった子を削除する API でも使用できます。この破棄の振る舞いはまたそのオブジェクトの GPU バッファもクリアしますが、テクスチャは破棄しません。テクスチャを破棄するには、Texture や TextureAtras オブジェクトで dispose メソッドを呼び出します。

すべてのリスナーの削除はまた、どの DisplayObject でも dispose API を明示的に呼び出すことで行えます。

```
clicked.dispose();
```

ここでは Starling のイベントモデルを初めて使用しましたが、これはネイティブの Flash Player のそれと非常によく似ています。次は少し時間を取って、Starling で使用できる EventDispatcher API を見ていきましょう。

イベントモデル

図２に示したように、すべての Starling オブジェクトは EventDispatcher クラスのサブクラスです。ネイティブの EventDispatcher API と同様、すべての Starling オブジェクトも、リスナーの追加と削除を行う API を公開しています。

・ addEventListener: リスナーを特定のイベントに登録する
・ hasEventListener: 特定のイベントを監視するリスナーを持っているかどうかを調べる
・ removeEventListener: イベントリスナーを削除する
・ removeEventListeners: 特定のイベント、またはすべてのイベントに登録されたすべてのリスナーを削除する

removeEventListeners API は非常に便利です。特定のイベントに登録したすべてのリスナーを削除したいときはいつでも、removeEventListeners にイベントタイプを渡してこれを呼び出します。

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

オブジェクトを取り除いたり無効化したいような場合で、イベントの種類に関係なくすべてのリスナーを削除する必要があるときには、removeEventListeners API にパラメータとしてイベントタイプを渡さずに呼び出します。

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

前の例では、リスナーを破棄する引数を取る removeChild API を使用しました。removeChild は内部的に同じ API を、各子について呼び出しています。

イベントの伝播

この Starling のチュートリアルの最初から見てきているように、Starling は Stage3D の上に表示リストの概念を再作成します。イベントの伝播のパワーは Starling でも再利用できます。イベントの伝播は、登録と登録解除するリスナーの数を限定しなければならない場合に実に有用で、コードも実際きれいになります。

イベントの伝播の概念がよく分からない方は、アドビの「ActionScript 3.0 のイベント処理について」ページ（http://www.adobe.com/jp/devnet/actionscript/articles/event_handling_as3.html）が参考になります。

興味深い詳細として、Starling はイベントの伝播を処理しますが、Flash のネイティブのそれとは少し異なります。Starling がサポートするのはバブリング段階だけで、キャプチャ段階の概念はありません。次の例ではイベントの伝播の動作を見ていきます。

タッチイベント

すでに述べているように Starling は Sparrow のいとこです。その結果、Starling のタッチイベントのメカニズムはモバイルとタッチ操作に向けに作成されています。これにより、Starling を、マウス操作を使用するデスクトップアプリケーションで使用するときには、いささか混乱する可能性があります。

図２を見てまず気がつくことは、ネイティブの表示リストと比べて、Starling の階層には InteractiveObject

クラスがないことです。Starling ではすべての表示オブジェクトがデフォルトでインタラクティブなのです。言い方を変えると、DisplayObject クラスはインタラクティブな振る舞いを定義します。

前の例ではタッチイベントを使用しました。そのときには四角形をマウスでタッチしたときの反応など、非常に基本的なところからスタートしました。そこでは TouchEvent.TOUCH イベントを使いました。

```
// スプライトがタッチされたら
_customSprite.addEventListener(TouchEvent.TOUCH, onTouchedSprite);
```

これはかなり限定的なのではないか？と思われているかもしれません。しかしこれは実にパワフルで、この１つのイベントで多くの異なる状態を判定することができます。マウスや指がグラフィックオブジェクトと相互作用するときには毎回、TouchEvent.TOUCH イベントが送出されます。

詳しく見ていきましょう。次のコードでは、onTouch イベントハンドラで Touch オブジェクトで使用できる phase プロパティを出力しています。

```
private function onTouch (e:TouchEvent):void
{
	// ステージを基準にしたマウス位置を取得
	var touch:Touch = e.getTouch(stage);
	var pos:Point = touch.getLocation(stage);

	trace( touch.phase);

	// マウス座標を保持
	mouseX = pos.x;
	mouseY = pos.y;
}
```

四角形とのインタラクティブ作用を開始し、クリックすると、異なる段階が出力されるのが確認できます。以下は TouchPhase API の定数として使用できる全段階のリストです。

- began: マウスか指がインタラクティブ作用を開始した（マウスダウン状態と似ている）
- ended: マウスか指がインタラクティブ作用を停止（ネイティブのクリック状態と似ている）
- hover: マウスか指がオブジェクト上に乗っている（ネイティブのマウスオーバー状態に似ている）
- moved: マウスか指がオブジェクトを動かしている（ネイティブのマウスダウン状態+マウスム―ブ状態に似て

いる）
・ stationary: マウスか指がオブジェクトとのインタラクティブ作用を停止し、そのままそこにある

TouchEvent オブジェクトには以下の API があります。

・ ctrlKey: ctrl キーの状態（押下げられているか否か）を返すブール値
・ getTouch: 特定のターゲット上で発生し、特定の段階にある最初の Touch オブジェクトを取得する
・ getTouches: 特定のターゲット上で発生し、特定の段階にある Touch オブジェクトのセットを取得する
・ shiftKey: shift キーの状態（押下げられているか否か）を返すブール値
・ timestamp: イベントが発生した時間（アプリケーションが起動してからの秒数）
・ touches: 現在発生しているすべてのタッチ

shiftKey と ctrlKey プロパティは、キーボードのキーとの組み合わせの判定に役立ちます。相互作用はつねに存在するので、指やマウスはそれに関係する Touch オブジェクトを持っています。

Touch オブジェクトには以下の API があります。

・ clone: オブジェクトのクローンを作成する
・ getLocation: タッチの現在の位置を、表示オブジェクトのローカル座標システムに変換する
・ getPreviousLocation: タッチの前の位置を表示オブジェクトのローカル座標システムに変換する
・ globalX: タッチの画面座標での x 位置
・ globalY: タッチの画面座標での y 位置
・ id: オブジェクトの一意の ID
・ phase: タッチの現在の段階
・ previousGlobalX: タッチの画面座標での前の x 位置
・ previousGlobalY: タッチの画面座標での前の y 位置
・ tapCount: 指で短時間に行われたタップ数。たとえばダブルタップの判定に使用
・ target: タッチが発生した表示オブジェクト
・ timestamp: イベントが発生した時間（アプリケーションが起動してからの秒数）

マルチタッチのシミュレート

モバイルデバイス向けのコンテンツを開発するときには、たとえば拡大操作のような、マルチタッチによる操作を利用したい機会が多くあります。デスクトップでのオーサリング時、実際のデバイスでテストできない場合でも、Starling にはマルチタッチをシミュレートするすばらしいメカニズムが組み込まれています。

これを有効にするには、Starling オブジェクトの simulateMultiTouch プロパティを使用します。

```
package
{
	import flash.display.Sprite;
	import flash.display.StageAlign;
	import flash.display.StageScaleMode;
	import starling.core.Starling;
	import net.hires.debug.Stats;

	[SWF(width = "1280",height = "752",frameRate = "60",backgroundColor = "#002143")]

	public class Startup extends Sprite
	{

		private var mStarling:Starling;

		public function Startup()
		{
			// fps 計測用クラス
			addChild (new Stats());

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

			// Starling インスタンスの作成
			mStarling = new Starling(Game, stage);
			// マルチタッチをシミュレート
			mStarling.simulateMultitouch = true;

			// アンチエイリアスの設定（大きいほど品質はアップするが、パフォーマンスは落ちる）
			mStarling.antiAliasing = 4;
			// スタート！
			mStarling.start();
		}
	}
```

```
}
```

有効にしたら、ctrl キーを押すと自動的に2つのドットが現われ、複数のタッチポイントをシミュレートします。図 14 はこの考えを示しています。

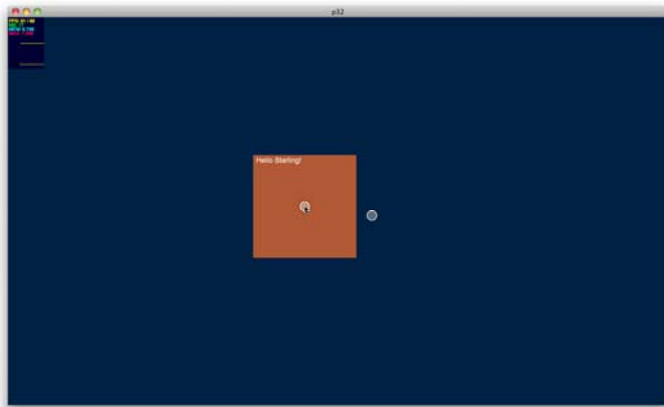


図 14：マルチタッチシミュレーション

次のコードでは、２本指のように、マルチタッチポイントを使って四角形を拡大縮小します。ここではタッチポイントを取得し、それらの間の距離を計算しています。

```
package
{
	import flash.geom.Point;

	import starling.display.Sprite;
	import starling.events.Event;
	import starling.events.Touch;
	import starling.events.TouchEvent;
	import starling.events.TouchPhase;

	public class Game extends Sprite
	{
		private var customSprite:CustomSprite;

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

```
		}

		private function onAdded( e:Event ):void
		{
			// カスタムスプライトを作成
			customSprite = new CustomSprite(200,200);

			// 位置はデフォルトでステージの中央
			// カスタムスプライトの基準点はセンターにあるので、半分だけ追加する
			customSprite.x = (stage.stageWidth - customSprite.width >> 1 ) +
(customSprite.width >> 1);
			customSprite.y = (stage.stageHeight - customSprite.height >> 1) +
(customSprite.height >> 1);

			// 表示する
			addChild(customSprite);

			stage.addEventListener(Event.ENTER_FRAME, onFrame);
			// スプライトがタッチされたら
			customSprite.addEventListener(TouchEvent.TOUCH, onTouchedSprite);
		}

		private function onFrame(e:Event):void
		{
			// カスタムスプライトの更新
			customSprite.update();
		}

		private function onTouchedSprite(e:TouchEvent):void
		{

			// タッチポイントを取得
			var touches:Vector.<Touch> = e.touches;
			// 2本指なら
			if ( touches.length == 2 )
			{
```

```
                    var finger1:Touch = touches[0];
                    var finger2:Touch = touches[1];

                    var distance:int;
                    var dx:int;
                    var dy:int;

                    // 両方の指が動いているなら（ドラッグ中なら）
                    if ( finger1.phase == TouchPhase.MOVED && finger2.phase ==
TouchPhase.MOVED )
                    {
                        // 各軸で距離を計算
                        dx = Math.abs ( finger1.globalX - finger2.globalX );
                        dy = Math.abs ( finger1.globalY - finger2.globalY );

                        // ２点間の距離を計算
                        distance = Math.sqrt(dx*dx+dy*dy);
                        trace ( distance );
                    }
                }
            }
        }
}
```

次はテクスチャです！

テクスチャ

テクスチャオブジェクトは Image オブジェクトに与えるために作成します。この関係はネイティブ API の Bitmap と BitmapData のようなものです。Texture オブジェクトを作成するときはつねに、Texture API を使用する必要があります。これには以下の API があります。

- base: テクスチャのベースになる Stage3D テクスチャオブジェクト
- dispose: 元になっているテクスチャデータを破棄する
- empty: サイズ（幅と高さ）から Texture オブジェクトを返す
- frame: テクスチャのフレーム（クラス定義を参照）

訳注:クラス定義での frame の説明。
テクスチャの frame プロパティを使用すると、テクスチャをイメージの範囲内に表示し、テクスチャの周囲の透明部分を残しておくことができます。フレーム矩形は、イメージではなく、テクスチャの座標システムで指定します。

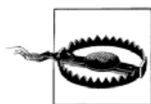
・ fromBitmap: Bitmap オブジェクトから Texture オブジェクトを返す。この Bitmap オブジェクトは埋め込みでも動的なロードによるものでも可能。
・ fromBitmapData: BitmapData オブジェクトから Texture オブジェクトを返す
・ fromAtfData: ATF(Adobe Texture Format)で圧縮されたテクスチャを使用可能にする。圧縮されたテクスチャを使用すると、特にモバイルデバイスのような制約のある環境で多くのメモリが節約できる
・ fromTexture: テクスチャを使用可能にし、新しいテクスチャを返す
・ height: テクスチャの、ピクセル単位の高さ
・ mipmapping: テクスチャがミップマップを含んでいるかどうかを示す
・ premultipliedAlpha: アルファ値が RGB 値に乗算済みであるかどうかを示す
・ repeat: テクスチャを壁紙のように繰り返すか、最も外側のピクセルをさらに引き伸ばすかどうかを示す
・ width: テクスチャの幅

テクスチャには以下のイメージ形式が使用できます。

・PNG: テクスチャには通常アルファチャンネルが必要なので、最もよく使用されます。
・JPEG: 従来の JPEG 形式も使用できます。ただし、GPU ではイメージの圧縮は解除されるので、JPEG を使用したからと言ってメモリ使用量を抑えたことにはならず、テクスチャで透明度が使用できなくなることは覚えておいてください。
・ATF(Adobe Texture Format): 圧縮に関する最良のファイル形式。ATF ファイルは一義的に、非可逆のテクスチャデータを保持するためのファイルコンテナです。ATF はその非可逆圧縮を、JPEG-XR1 圧縮とブロックベース圧縮という2つの一般的な技術を使って実現します。JPEG-XR 圧縮はストレージ容量とネットワークの帯域幅を節約する、優位な方法を提供し、ブロックベース圧縮は、RGBA テクスチャと比べ 1:8 の比率で、クライアントでのテクスチャのメモリ使用量を抑える方法を提供します。ATF は DXT12、ETC13、PVRTC4 という3つのブロックベース圧縮をサポートします。

ここからはテクスチャの概念を掘り下げ、GPU のイメージの本質的な概念であるミップマッピングを探っていきましょう。ミップマッピングは重要な概念ですが、簡単に理解できます。テクスチャのスケールダウン(縮小)版をミップマップと言います。テクスチャを GPU で扱うときには、コンテンツのサイズによってはイメージを縮小する必要のある場合があります。これはたとえば、カメラがコンテンツに向かって移動したり、コンテンツがカメラから離れる場合などです。その場合にはコンテンツは小さくなるので、その結果テクスチャも縮小する必要があ

ります。


テクスチャのサイズは 2 のべき乗である必要があります（1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048）が、正方形である必要はありません。このルールを尊重しない場合には、Starling がそのイメージのサイズに最も近い 2 のべき乗を自動的に見つけて、そのサイズのテクスチャを作成します。これは結果的にメモリの浪費につながります。テクスチャが使用するメモリを確実に最適化するには、一般にスプライトシートと呼ばれるテクスチャアトラスの使用が推奨されます。このトピックについては後で見ていきます。

GPU で確実に最高の品質にするには、イメージのミップマップレベルをすべて用意する必要があります。これは、イメージの全バージョンを、元のサイズ（2 のべき乗である必要があります）から 1 のサイズまで準備するということです。Starling を使用しない場合には、BitmapData.draw と半分に縮小する変換行列を使った単純な工夫で手作業で生成する必要があります。

ミップマップは 2D コンテンツでは、アップロードするのが良いプラクティスです。これによりコンテンツの実行が速くなり、要素を縮小するときその見栄えがよくなります（ギザギザが減ります）。

ありがたいことに、Starling では前述したようにミップマップを自動生成してくれます。以下は Starling が内部的に使用するミップレベル生成コードです。

```
if (generateMipmaps)
{
    var currentWidth:int = data.width >> 1;
    var currentHeight:int = data.height >> 1;
    var level:int = 1;
    var canvas:BitmapData = new BitmapData(currentWidth, currentHeight, true, 0);
    var transform:Matrix = new Matrix(.5, 0, 0, .5);

    while (currentWidth >= 1 || currentHeight >= 1)
    {
        canvas.fillRect(new Rectangle(0, 0, currentWidth, currentHeight), 0);
        canvas.draw(data, transform, null, null, null, true);
        texture.uploadFromBitmapData(canvas, level++);
```

```
            transform.scale(0.5, 0.5);
            currentWidth = currentWidth >> 1;
            currentHeight = currentHeight >> 1;
        }

        canvas.dispose();
}
```

ATF 形式（Adobe Texture Format）を使用するときには、これを心配する必要はありません。ATF ファイル形式にすでにミップマップレベルが含まれており、前もって生成されているので実行時に生成する必要はありません。これは、ミップマップレベルの生成に使う時間が節約できるという理由で重要です。多くのテクスチャを使用する場合には、貴重な時間が節約でき、コンテンツの初期化も速くなります。

Texture オブジェクトには frame プロパティがあります。これを使用すると、テクスチャを Image オブジェクトに割り当てるとき、その位置を決めることができます。イメージの周りに何らかの枠が欲しい場合には、イメージよりも小さなテクスチャを使い、テクスチャをイメージの中央に配置します。

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

Image オブジェクトが出てきたので、次はこれを見ていきましょう。

イメージ

Starling の starling.display.Image オブジェクトは、ネイティブの flash.display.Bitmap オブジェクトに相当します。

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

イメージを表示するには、Image オブジェクトを作成し、それに Texture オブジェクトを渡します。

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

        import starling.display.Image;
```

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

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

		private const NUM_SAUSAGES:uint = 400;

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

		public function Game()
		{
			addEventListener(Event.ADDED_TO_STAGE, onAdded);
		}
		private function onAdded(e:Event):void
		{
			// 埋め込んだイメージから Bitmap オブジェクトを作成
			var sausageBitmap:Bitmap = new Sausage();

			// Image オブジェクトに渡す Texture オブジェクトを作成
			var texture:Texture = Texture.fromBitmap(sausageBitmap);

			for (var i:int = 0; i < NUM_SAUSAGES; i++)
			{
				// テクスチャを使って Image オブジェクトを作成
				var image:Image = new Image(texture);

				// ランダムなアルファ、位置、回転を設定
				image.alpha = Math.random();

				// ランダムな初期位置を決める
```

```
                image.x = Math.random() * stage.stageWidth;
                image.y = Math.random() * stage.stageHeight;
                image.rotation = deg2rad(Math.random() * 360);

                // 表示
                addChild(image);

                // 後で使えるように参照を保持しておく
                sausagesVector[i] = image;
            }
        }
    }
}
```

Texture オブジェクトの生成には、Texture クラスの静的な fromBitmap API を使っています。このコードを実行すると、図 15 に示すような結果が得られます。

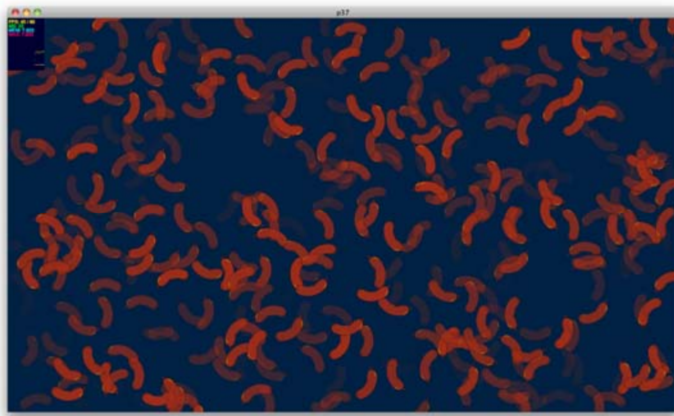


図 15：ランダムに配置したソーセージ

訳注：原書ではサンプルファイルが提供されていないので、ここで使用している sausage.png は原書の P43 から撮ったものを PNG ファイルとして使用しています。サイズは 32 x 64 ピクセルで、背景は透明です。このファイルは FLA ファイルのあるフォルダと同階層の media フォルダ内の texture フォルダ内に置きます。

ここではビットマップを埋め込んでいますが、これは動的にロードすることもできます。そのためには Loader オブジェクトを使ってテクスチャをロードし、ロードしたビットマップを取得して、fromBitmap API を使って Starling 用テクスチャを生成します。

```
// ローダーを作成
var loader:Loader = new Loader();

// テクスチャをロード
loader.load ( new URLRequest ("texture.png") );

// テクスチャがロードされたら
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );

function onComplete ( e : Event ):void
{
	// ロードしたビットマップを取得
	var loadedBitmap:Bitmap = e.currentTarget.loader.content as Bitmap;

	// ロードしたビットマップからテクスチャを作成
	var texture:Texture = Texture.fromBitmap ( loadedBitmap )
}
```

Texture オブジェクトを Bitmap からではなく、BitmapData から生成するときには、別の API を使用します。これについては後で見ていきます。

ここでは画面上のすべてのスプライトで同じテクスチャを再利用している点に注目してください。次のコードでは、各繰り返しにおいて、初めにループの外で作っておいたただ1つだけのテクスチャを再利用しています。

```
// Image オブジェクトに渡す Texture オブジェクトを作成
	var texture:Texture = Texture.fromBitmap(sausageBitmap);

	for (var i:int = 0; i < NUM_SAUSAGES; i++)
	{
		// テクスチャを使って Image オブジェクトを作成
		var image:Image = new Image(texture);
```

次のように、各繰り返しにおいてテクスチャを再作成するのは悪いプラクティスです。

```
for (var i:int = 0; i < NUM_SAUSAGES; i++)
```

```
{
	// ソーセージごとに新しいテクスチャを作成して、Image オブジェクトを作成
	var image:Image = new Image(Texture.fromBitmap(new Sausage()));
```

これは、テクスチャ用の同じビットマップの複数のコピーが割り振られるので、メモリに悪い影響を及ぼすだけでなく、コンテンツの実行にも全体的に影響します。なぜなら GPU には同じイメージの複数のコピーがアップロードされるからで、これは非効率的です。大事なことが抜けていましたが、ループの実行にも影響します。ループの中で fromBitmap を呼び出すと、そのたびにミップマップを生成することになります。

では先に進みましょう。次の CustomImage クラスを作成します。

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

	public class CustomImage extends Image
	{
		public var destX:Number = 0;
		public var destY:Number = 0;

		public function CustomImage(texture:Texture)
		{
			super(texture);
		}
	}
}
```

この CustomImage クラスを次のように使用します。

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

	import starling.display.Image;
	import starling.display.Sprite;
```

```
	import starling.events.Event;
	import starling.textures.Texture;
	import starling.utils.deg2rad;

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

		private const NUM_SAUSAGES:uint = 400;

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

		public function Game()
		{
			addEventListener(Event.ADDED_TO_STAGE, onAdded);
		}
		private function onAdded(e:Event):void
		{
			// 埋め込んだイメージから Bitmap オブジェクトを作成
			var sausageBitmap:Bitmap = new Sausage();

			// Image オブジェクトに渡す Texture オブジェクトを作成
			var texture:Texture = Texture.fromBitmap(sausageBitmap);

			for (var i:int = 0; i < NUM_SAUSAGES; i++)
			{
				// テクスチャを使って Image オブジェクトを作成
				var image:Image = new Image(texture);

				// ランダムなアルファ、位置、回転を設定
				image.alpha = Math.random();

				// ランダムな初期位置を決める
				image.x = Math.random() * stage.stageWidth;
```

```
                    image.y = Math.random() * stage.stageHeight;
                    image.rotation = deg2rad(Math.random() * 360);

                    // 表示
                    addChild(image);

                    // 後で使えるように参照を保持しておく
                    sausagesVector[i] = image;
                }
            }
        }
}
```

すでに見たように、Starling はイベントの伝播を処理します。これにより、移動するすべてのイメージからのタッチイベントを捕捉するコードがきれいになります。そのためにはただ、TouchEvent.TOUCH イベントをステージで監視するだけです。

```
// ステージ上のマウスの動きを監視
stage.addEventListener(TouchEvent.TOUCH, onClick);
```

target と currentTarget プロパティを調べると、そのイベントを送出したオブジェクト（currentTarget）が Stage で、イベントを初期化したオブジェクト（target）が CustomImage インスタンスであることが確認できます。

```
private function onClick(e:TouchEvent):void
{
    // タッチポイントを取得（マルチタッチなので複数も可）
    var touches:Vector.<Touch> = e.getTouches(this);
    var clicked:DisplayObject = e.currentTarget as DisplayObject;

    // タッチが１本指かマウスなら
    if ( touches.length == 1 )
    {
        // タッチポイントを取得
        var touch:Touch = touches[0];
```

```
            // クリック/リリース段階かを判定
            if ( touch.phase == TouchPhase.ENDED )
            {
                  // 出力 : [object Stage] [object CustomImage]
                  trace ( e.currentTarget, e.target );
            }
      }
}
```

この方法を使用すると、リスナーをイメージごとに登録する必要がなくなります。イベントはただイメージのコンテナ（今の場合はステージ）で監視し、バブリング段階でイベントを捕捉するだけです。TouchEvent オブジェクトの bubble プロパティを使うと、イベントがバブリング（遡上）していることが分かります。

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

イベント伝播のメカニズムは以上です。次は Image オブジェクトとその機能を見ていきましょう。予想されるように、Image オブジェクトでは DisplayObject から継承する全 API が公開されていますが、それに加えイメージの滑らかさを処理する smoothing プロパティもあります。使用できるのは次の値で、これらは TextureSmoothing クラスの定数として定義されています。

- BILINEAR: 拡大縮小時、テクスチャにバイリニアフィルタを適用する（デフォルト）
- NONE: 拡大縮小時滑らかさは適用されず、ピクセルは大きな矩形として拡大される（最近傍補間、ニアレストネイバー）
- TRILINEAR: 拡大縮小時、テクスチャにトリリニアフィルタを適用する

次のように使用します。

```
// 拡大縮小時フィルタリングを無効にする
image.smoothing = TextureSmoothing.NONE;
```

図 16 はバイリニアフィルタ（TextureSmoothing.BILINEAR）を使って拡大したイメージです。

図 16 : TextureSmoothing.BILINEAR

図 17 はトリリニアフィルタ（TextureSmoothing.TRILINEAR）を使って拡大したイメージです。

図 17 : TextureSmoothing.TRILINEAR

図 18 はフィルタを使用せずに拡大したイメージです（TextureSmoothing.NONE）。

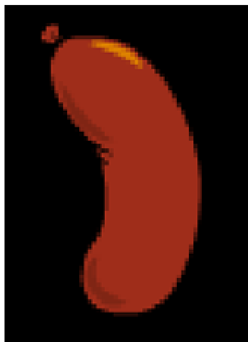


図 18 : TextureSmoothing.NONE

このイメージについては、フィルタを適用したいときが良く見えます。

Image オブジェクトでは、カラー値が指定できる color プロパティが公開されています。各ピクセルに関し、最終的なカラーは、テクスチャのカラーと指定したカラーとの乗算の結果になります。これによりイメージの色合いを簡単に変えることができ、別のテクスチャを使用しなくてもイメージのさまざまなバリエーションが作成できます。

図 19 はこの考えを示しています。

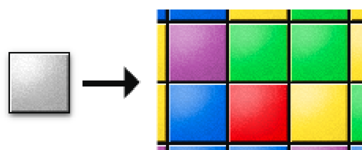


図 19：左のカラーによって、テクスチャの色合いを変える（図は http://www.sparrow-framework.org/からの引用）

では Starling で動的なカスタム形状を使用したい場合にはどうすればよいのでしょう？ 問題ありません、次節に進んでください。

衝突判定

ほとんどのゲームでは、物理エンジン（Box2D のような、これについては後で述べます）に依存しない場合、単純な当たり判定を処理することになります。円のような形状を使うときには、２点間の距離が半径よりも小さいかどうかを調べるだけで事足ります。そのほかの場合でも、境界ボックス間を調べる単純なテストで済みますが、ピクセルパーフェクト（ピクセル同士）の衝突はどうすればよいのでしょう？

図 20 はその典型的な例です。ここでは透明な領域を含んだ2つのオブジェクト同士の衝突を判定する必要があります。

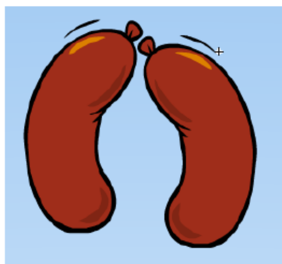


図 20：ピクセルパーフェクトの衝突

この場合、曲線があるので、衝突はピクセルレベルで判定したいわけです。もちろん、ピクセルレベルの当た

り判定を実行するには、透明度を持ったイメージを使用します。

ActionScript コードの実行によるピクセルの判定テストは実際には、コストが高くつく可能性があります。ありがたいことに、Starling でネイティブの BitmapData API に依存してテクスチャを作成している場合には、その BitmapData オブジェクトの hitTest API が使用できます。

hitTest API は実際には単純なのですが、初めは少し戸惑うかもしれません。以下はそのシグネチャです。

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

secondObject パラメータは Point か Rectangle、BitmapData オブジェクトで、これにより多くの場合でこの API の使い勝手が非常によくなっています。次の例では2つの BitmapData オブジェクトを使っていますが、Image オブジェクトは BitmapData を使ったテクスチャで作成できるので、これは Starling ではよくあるケースです。

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

ここではまず、衝突先の２つめの（衝突される）オブジェクトの位置であるポイント（new Point(sausageImage2.x, sausageImage2.y)）、次にピクセルが不透明だと見なすアルファのしきい値（255、ほとんどの場合で 255 か 0xFF を使用します）を渡します。そして現在のイメージ（衝突する方）の位置であるポイント（new Point(sausageImage1.x, sausageImage1.y)）とそのアルファのしきい値を渡します。

この当たり判定テストは各フレームで１回呼び出すことになるので、ガベージコレクタのロードは最小限に抑えるべきです。前のコードでは、当たり判定を実行するときに Point オブジェクトを作成していましたが、もっとよい方法は、Point オブジェクトの x と y プロパティを更新する方法です。これにより毎フレーム、それらを割り振る必要がなくなります。

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

```
        point2.x = sausageImage2.x;
        point2.y = sausageImage2.y;

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

次は、Starling での ActionScript ネイティブの描画 API の使用方法を見ていきましょう。

描画 API

Starling では、ネイティブの flash.display.Graphics オブジェクトのような描画 API は公開されていませんが、ネイティブの描画 API を使って BitmapData オブジェクト内に描画し、それをテクスチャとして使用することで、描画機能を簡単にシミュレートできます。

Starling で円を使ってそれを表示するコードは、たとえば次のように記述できます。

```
// ベクターシェイプの作成（Graphics）
var s:flash.display.Sprite = new flash.display.Sprite();

// カラーの選定
var color:uint = Math.random() * 0xFFFFFF;

// カラーの塗りの設定
s.graphics.beginFill(color,ballAlpha);

// 半径
var radius:uint = 20;

// 指定された半径で円を描画する
s.graphics.drawCircle(radius,radius,radius);
s.graphics.endFill();

// BitmapData の作成
```

```
var buffer:BitmapData = new BitmapData(radius * 2, radius * 2, true, color);

// BitmapData 上でシェイプを描画
buffer.draw(s);

// BitmapData からテクスチャを作成
var texture:Texture = Texture.fromBitmapData(buffer);

// テクスチャから Image を作成
var image:Image = new Image(texture);

// 表示
addChild(image);
```

アイデア自体は単純です。ネイティブの Graphics API を使って線やストローク、塗りを CPU で描画し、それからビットマップデータ上でラスタライズして、それをテクスチャとして GPU にアップロードします。

図21は Starling で円を描画した例です。

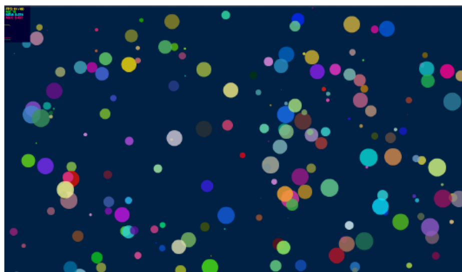


図21：動的に作成したカスタムシェイプ

Starling のフラットスプライトはご存じでしょうか？　次はこの実にパワフルな機能を使ってパフォーマンスをアップさせる方法を見ていきましょう。

フラットスプライト

Starling には、パフォーマンスの大きな向上が見込める、フラットスプライトと言う非常にパワフルな機能があります。

デフォルトでの表示リストの動作をよりよく理解するため、図 22 を見てください。これは多くの異なるオブジェ

クトをネストしたアプリケーションの典型的な表示ツリーを示しています。

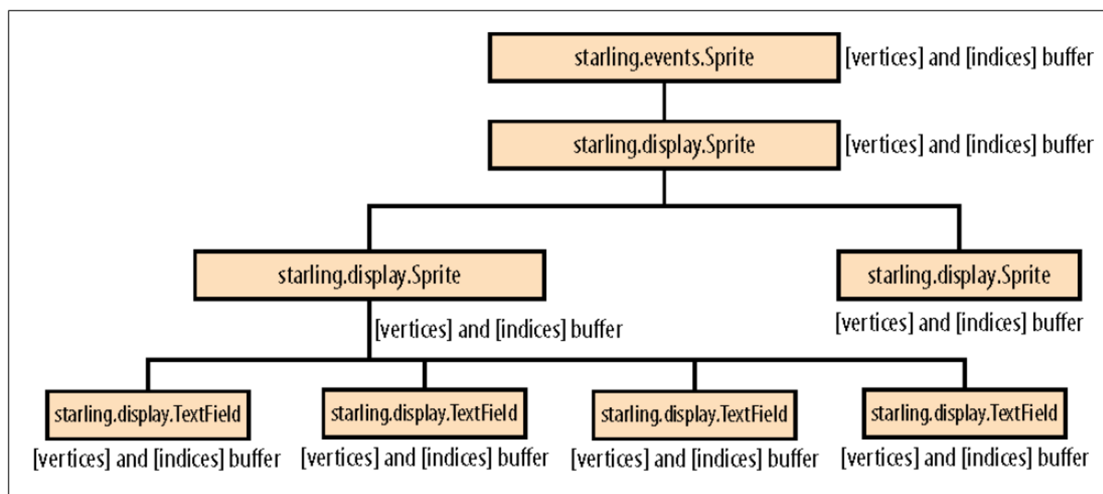


図 22：子はそれ自身の頂点バッファとインデックスバッファを持っている

図 22 で示すように、Starling は、自分自身の頂点バッファとインデックスバッファを持つ子それぞれを処理し、すべての子を振る舞いを個別に処理しなければなりません。そのためには多量の計算が必要で、計算がパフォーマンスに影響する場合もあります。

Starling はこれを、すべての子のジオメトリ（形状）を単一の大きな頂点バッファに集め、その全体（コンテ他とその複数の子）を１回の描画呼び出しで（子が同じテクスチャを共有している場合）、単純なテクスチャのように描画することで、処理します。図 23 を見てください。

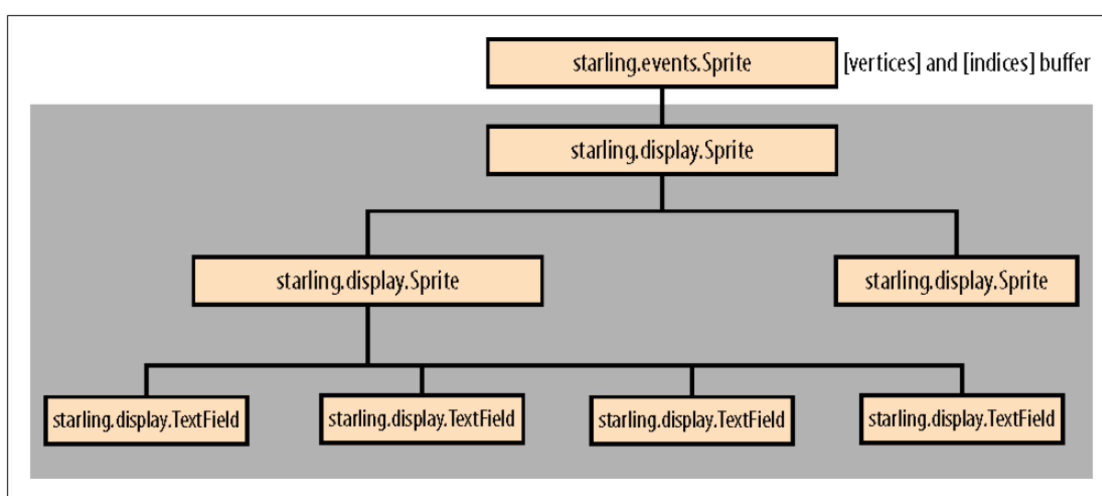


図 23：flatten（平準化）されると、子は１回の描画呼び出しで描画される（１つのインデックス/頂点バッファ）

これは、ネイティブの表示リストでサポートされている cacheAsBitmap（ビットマップキャッシュ）機能のような方法だと考えることができます。ただし例外があり（これがキーなのですが）、ツリー内の子を変更するとき、描画されるサーフェスは自動的にはレンダリングされません。変更を表示するには、明示的に flatten API を呼び出す必要があります。

以下は使用できる API のリストです。

・flatten：コンテンツをできるだけ速くレンダリングしたいとき flatten を呼び出します。これが呼び出されると、Starling は表示ツリーの描画に必要なすべてのジオメトリを集め、すべてのデータを単一のバッファにグループ化して、まるで単純なテクスチャを描画するかのように、1回の描画呼び出しでコンテンツを描画します。もちろんこのパワーにも制限があり、呼び出し後に（複数の）子に行われたすべての変更は、それを反映するために再度 flatten を呼び出すまで、反映されません。
・unflatten：flatten の振る舞いを無効化します。
・isFlattened：そのスプライトが現在 flatten されているかどうかを示します。

ではこれを試してみましょう。次のコードでは Sprite 内に複数のイメージを追加し、毎フレーム、コンテナを回転させます。

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

	import starling.display.Image;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.textures.Texture;

	import starling.utils.deg2rad;
	public class Game extends Sprite
	{
		private var container:Sprite;

		private static const NUM_PIGS:uint = 400;

		[Embed(source = "../media/textures/pig-parachute.png")]
```

```
private static const PigParachute:Class;

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

private function onAdded(e:Event):void
{
	// コンテナの作成
	container = new Sprite();

	// 基準点の変更
	container.pivotX = stage.stageWidth >> 1;
	container.pivotY = stage.stageHeight >> 1;

	container.x = stage.stageWidth >> 1;
	container.y = stage.stageHeight >> 1;

	// 埋め込んだイメージから Bitmap オブジェクトを作成
	var pigTexture:Bitmap = new PigParachute();

	// Image オブジェクトに与える Texture オブジェクトを作成
	var texture:Texture = Texture.fromBitmap(pigTexture);

	// ブタの配置
	for (var i:uint = 0; i< NUM_PIGS; i++)
	{
		// 新しいブタの作成
		var pig:Image = new Image(texture);
		// ランダムな位置
		pig.x = Math.random() * stage.stageWidth;
		pig.y = Math.random() * stage.stageHeight;
		pig.rotation = deg2rad(Math.random() * 360);
		// ブタをコンテナにネスト
		container.addChild( pig );
```

```
            }

            container.pivotX = stage.stageWidth >> 1;
            container.pivotY = stage.stageHeight >> 1;

            // ブタを表示
            addChild( container );

            // 毎フレーム呼び出す
            stage.addEventListener(Event.ENTER_FRAME, onFrame);
        }
        private function onFrame(e:Event):void
        {
            // コンテナを回転
            container.rotation +=   .1;
        }
    }
}
```

このテストでは、アニメーションは非常に滑らかで、毎秒 60 フレームで実行されます。しかしこれは最適化して、描画呼び出しの回数を最小限に減らすことができます。では次のように flatten API を呼び出してみましょう。

```
// 子をフリーズさせる
container.flatten();
```

これによりすべての子は単一の描画呼び出しによって描画されます。デスクトップ上ではフレームレートに実行的な違いは現れないかもしれませんが、CPU の使用量については確実に差があります。何回かテストを行うと、10 倍もしくはそれ以上、CPU の使用量が減っていることが分かります。

子が同じテクスチャを共有していない場合には、Starling は描画呼び出しを分けます。このような場合には、flatten の振る舞いのメリットは小さくなります。一般的には、ゲームはレイヤーでデザインするようにします。そして各レイヤーの要素はすべて、同一のスプライトシートに置くようにします。

この機能はまたモバイルでも大いに役立つでしょう。そしてもちろん、この機能のもう1つの価値は動的であるということなので、unflattenを呼び出した後オブジェクトを再配置し、再度flattenを呼び出すこともできます。そのときにはテクスチャを動的なスプライトからコンパイルすることができます。子はいつでも修正でき、その変更は、flatten を再度明示的に呼び出すことで、画面上に反映されます。

ただし覚えておく必要があるのは、flatten の振る舞いは静的なコンテンツ(Sprite)でのみ動作するということです。Starling では現在、MovieClip 向けの最適化は提供されていません。こういった機能は今後のリリースで実現するかもしれません。

MovieClip

ここまで Starling の Sprite API の使用方法を見てきましたが、では MovieClip を使ったアニメーションについてはどうなのでしょう？ Flash のデベロッパーはみんな、ムービークリップの概念に習熟しており、ここ何年か、ほとんどの AS3 デベロッパーは、レンダリングをビットマップの使用によって高速化しつつ MovieClip ごとに独立したフレームレートを獲得しようと、BitmapData をベースにした MovieClip API を再作成してきました。

図 24 はわれわれの MovieClip の各フレームを示すスプライトシートアトラス(図版)です。

図 24：ランニングボーイのスプライトシートアトラス

これは、各フレームを GPU でサンプリングし、シーンに表示しようというアイデアです。毎フレーム、テクスチャを更新することで、ムービークリップの概念を再現することができます。ではこのアニメーションはどうやって作成されるのでしょう？ Flash はこのためのみなさんの最良の友人で、アニメーションの各フレームをイメージシーケンスに書き出すことができます。それらのイメージを TexturePacker (http://www.texturepacker.com)などのツールに読み込みます。TexturePacker はイメージを単一のテクスチャに結合させます。これを Starling の MovieClip オブジェクトに使用します。Flash の次のバージョン(CS6)では、Starling のスプライトシートが生成できるようになると言われています。

図 25 はスプライトアトラスのフレームを示しています。

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

図 25：スプライトアトラスのフレーム

アトラスをミップマッピングする場合には、ミップマッピングされる時立ち落とされる部分が出ないように、各フレーム間に必ず2ピクセルの空きを設けます。これは、GPU がより小さなフレームをサンプリングするとき、別のフレームのピクセルもいっしょにサンプリングしないようにする、ということです。

ここで、テクスチャのサイズにはいくつか制限があったことを思い出してください。Stage3D（Molehill）はモバイルも念頭に置いて設計されています。その結果、Stage3D には 2 のべき乗のテクスチャといった OpenGL ES2 の制限が課されます。TexturePacker はこれを順守するときの助けになります。また AutoSize（図 26 参照）という、テクスチャの最大の幅と高さを尊重しつつ、テクスチャにとって最良の幅と高さを決めるすぐれた機能も備えています（Starling には 2048 x 2048 の制限があります）。

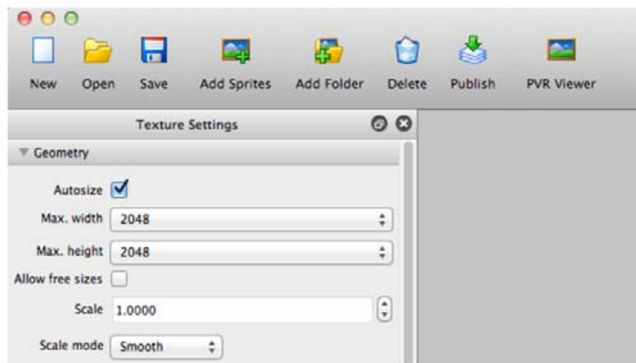


図 26：TexturePacker の AutoSize 機能

前に述べたように、Starling は必ず 2 のべき乗のサイズのテクスチャを使用するように自動的に設定します。みなさんが 2 のべき乗のテクスチャを提供しない場合には、Starling がみなさんに代わってそれを処理し、イメージで使用できる、次の 2 のべき乗のサイズを計算してトリミングします。

Starling に、フレームがどの位置にあるかを知らせるには、TextureAtlas API に XML ファイルを渡す必要があります。TextureAtlas API はその XML からテクスチャの Vector を生成します。前の TexturePacker は

このファイルを、スプライトシートの生成時に自動的に生成します。使用する形式には XML や JSON などが選択できます。

Starling はネイティブで XML をサポートします。以下は XML ファイルの例です。

```
<?xml version="1.0" encoding="UTF-8"?>
<TextureAtlas imagePath="running-sheet.png">
    <!-- Created with TexturePacker -->
    <!-- http://texturepacker.com -->
    <!-- $TexturePacker:SmartUpdate:2b3f5fa2588769393bcea9b632749826$ -->
    <SubTexture name="running0001" x="0" y="0" width="304" height="284"/>
    <SubTexture name="running0002" x="304" y="0" width="304" height="284"/>
    <SubTexture name="running0003" x="608" y="0" width="304" height="284"/>
    <SubTexture name="running0004" x="0" y="284" width="304" height="284"/>
    <SubTexture name="running0005" x="304" y="284" width="304" height="284"/>
    <SubTexture name="running0006" x="608" y="284" width="304" height="284"/>
    <SubTexture name="running0007" x="0" y="568" width="304" height="284"/>
    <SubTexture name="running0008" x="304" y="568" width="304" height="284"/>
    <SubTexture name="running0009" x="608" y="568" width="304" height="284"/>
    <SubTexture name="running0010" x="0" y="852" width="304" height="284"/>
    <SubTexture name="running0011" x="304" y="852" width="304" height="284"/>
    <SubTexture name="running0012" x="608" y="852" width="304" height="284"/>
    <SubTexture name="running0013" x="0" y="1136" width="304" height="284"/>
    <SubTexture name="running0014" x="304" y="1136" width="304" height="284"/>
    <SubTexture name="running0015" x="608" y="1136" width="304" height="284"/>
</TextureAtlas>
```

この美しさは、フレーム全体を制御し、したがってフレームレートを制御している点にあります。これにより独立したフレームレートを使う複数のムービークリップを持つことが可能になります。Starling でもこれと同じテクニックが GPU に対して利用されます。以下は Starling の MovieClip コンストラクタの定義です。

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

次のコードでは、ムービークリップのフレームを含むテクスチャを作成しています。

```
[Embed(source = "../media/textures/running-sheet.png")]
```

```
private const SpriteSheet:Class;

var bitmap:Bitmap = new SpriteSheet();

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

次いでスプライトシートの各フレームの位置を記述した XML ファイルを取得します。

```
[Embed(source="../media/textures/running-sheet.xml", mimeType="application/octetstream")]
public const SpriteSheetXML:Class;

var xml:XML = XML(new spriteSheetXML());

var sTextureAtlas:TextureAtlas = new TextureAtlas(texture, xml);
```

するとランニングボーイに関係するフレームを取得できます。

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

ここで渡している引数"running_"は、スプライトシートからはこれに関連するフレームだけを得たい、ということを意味しています。定義さえしていれば、"jump"や"fire"といったほかのシーケンスも要求できます。

以下はここまでの全コードです。

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

	import starling.core.Starling;
	import starling.display.MovieClip;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.textures.Texture;
	import starling.textures.TextureAtlas;

	public class Game extends Sprite
```

```
	{
		private var mMovie:MovieClip;

		[Embed(source = "../media/textures/running-sheet.xml",mimeType = "application/octet-stream")]
		public static const SpriteSheetXML:Class;

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

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

		private function onAdded(e:Event):void
		{
			// 埋め込みビットマップを作成（スプライトシートファイル）
			var bitmap:Bitmap = new SpriteSheet();

			// ビットマップからテクスチャを作成
			var texture:Texture = Texture.fromBitmap(bitmap);

			// スプライトシートのフレームの詳細を表す XML ファイルを作成
			var xml:XML = XML(new SpriteSheetXML());

			// テクスチャアトラスを作成（スプライトシートと XML の記述をリンクさせる）
			var sTextureAtlas:TextureAtlas = new TextureAtlas(texture,xml);

			// ランニングボーイのフレームを取得
			var frames:Vector.<Texture >  = sTextureAtlas.getTextures("running_");

			// 40fps で再生する MovieClip を作成
			mMovie = new MovieClip(frames,40);

			// MovieClip をセンターに置く
```

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

                // 表示
                addChild( mMovie );
            }
        }
}
```

訳注:
以下はわたしがこの例を試した方法です。

1） Flash IDE のメインのタイムラインでアニメーションを作成します。下図ははちまきをした金魚すくいの男が客寄せのため手を振るアニメーションです。

2） FLA ファイルでプレビューし、アニメーションの SWF を作成します。

3） TexturePacker を開き、右の[Sprites]ペインに SWF ファイルをドラッグします。すると真ん中のペインにスプライトシートが表示されます。TexturePacker（.tps）ファイルを保存します。

4） 左の[Texture Settings]ペインの[Output]の[Data Format]で[Sparrow]を選択します。その下の[Data file]で XML ファイルの、[Texture file]で PNG ファイルの出力先を指定します。

5） 上部にあるメニューバーで[Publish]ボタンをクリックします。

6） 作成された XML ファイル（wavehandman.xml）と PNG ファイル（wavehandman.png）を、media/textures フォルダにコピーします。

7） wavehandman.xml をテキストエディタで開くと、<SubTexture name> 属性が”wavehandman.swf/0000”から”wavehandman.swf/0013”までとなっていました。そこで、sTextureAtlas.getTextures()の引数には”wavahandman”を使うことにしました。getTextures()は

Starling のリファレンスによると、「特定のストリングで始まる全テクスチャを、アルファベット順にソートして返す」とあります。

図 27 はその実行結果です。なお TexturePacker をフリーで使用すると、書き出されるスプライトシートの一部が赤くなり、そのままではアニメーションには使用できません。

図 27：レンダリング結果

アセットを再利用するための簡単なキャッシュテクニックがあり、これについては後で見ていきます。このテクニックを使用すると、アプリケーションの存続中、インスタンス化するオブジェクトの数を抑えることができます。

ムービークリップをフリップ（左右反転）させるには、ネイティブの Flash API と同様に、scaleX プロパティを使い、幅を加えて前と同じ位置に戻します。

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

mMovie.scaleX = -1;

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

図 28 はこの結果を示しています。

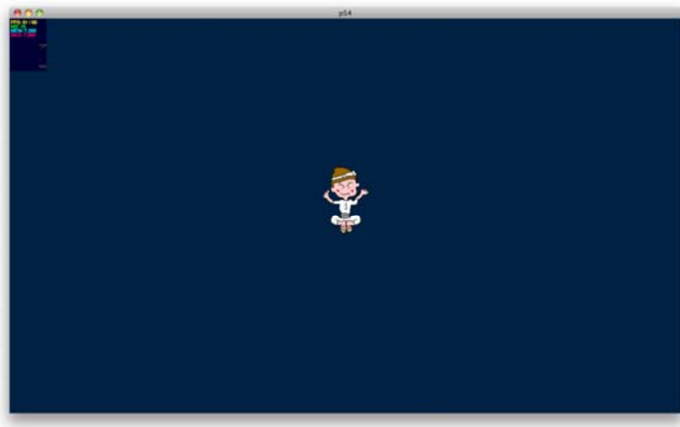


図 28：ムービークリップの左右反転

前に述べているように、アセットは通常、そのすべてが単一のテクスチャに置かれ、これは良いプラクティスとされています。この理由が分かりますか？

まず、ゲームのアセットすべてが１つのテクスチャファイルに保持されていると、便利です。テクスチャファイルを１つだけ使うことで、GPU にアップロードする回数が最小限に抑えられます。このアップロードは GPU に負担をかけ、特にモバイル環境でそうなることをよく覚えておいてください。つまり GPU にはアップロード回数が少ないほど好ましいのです。最後、あるテクスチャから別のテクスチャへの切り替えもまたコストがかかります。したがって、アセットのサンプリングには、あるテクスチャから別のテクスチャにたびたび切り替えるよりも、単一のテクスチャを使う方が好ましいのです。

テクスチャアトラス

ここまでスプライトアトラスの概念を見てきました。次はテクスチャアトラスの概念を紹介しましょう。テクスチャアトラスでは、アセットはすべて単一のテクスチャに含まれます。

図 29 では、別のフレームのシーケンス（右の男）を追加しています。

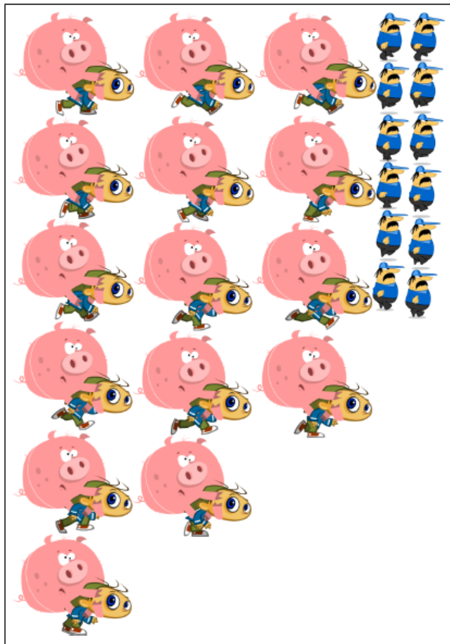


図 29：すべてのアセットを含むテクスチャアトラス

XML 記述ファイルには、新たにブッチャー（肉屋）のテクスチャが含まれます。

```
<?xml version="1.0" encoding="UTF-8"?>
<TextureAtlas imagePath="running-sheet.png">
	<!-- Created with TexturePacker -->
	<!-- http://texturepacker.com -->
	<!-- $TexturePacker:SmartUpdate:5aa8dfdc90d616e76e06b3079d2c5e80$ -->
	<SubTexture name="french-butcher_01" x="930" y="486" width="77" height="122"
frameX="-106" frameY="-33" frameWidth="275" frameHeight="200"/>
	<SubTexture name="french-butcher_02" x="845" y="588" width="77" height="118"
frameX="-106" frameY="-37" frameWidth="275" frameHeight="200"/>
…
```

これらのフレームの参照は、TextureAtlas の getTextures API で行います。

```
// ランニングボーイからフレームを取得
var frames:Vector.<Texture> = sTextureAtlas.getTextures("running_");

// ブッチャーからフレームを取得
var framesButcher:Vector.<Texture> = sTextureAtlas.getTextures("french-butcher_");
```

```
// 40fps で再生される MovieClip を作成
mMovie = new MovieClip(frames, 40);

// 25fps で再生される MovieClip を作成
mMovieButcher = new MovieClip(framesButcher, 25);

// これらを配置
mMovie.x = stage.stageWidth - mMovie.width >> 1;
mMovie.y = stage.stageHeight - mMovie.height >> 1;
mMovieButcher.x = mMovie.x + mMovie.width + 10;
mMovieButcher.y = mMovie.y;

// これらを表示
addChild ( mMovie );
addChild ( mMovieButcher );
```

ここでは少年の右横にブッチャーを配置しています（図 30 参照）。

これで画面上に2つのムービークリップが作成できましたが、まだ再生していません。これらを再生するには Juggler オブジェクトを使う必要があります。

デフォルトの juggler は、コンテンツを処理する Starling オブジェクトの juggler プロパティとして使用できます。少年とブッチャーをアニメートするには、次のコードを追加します。

```
// これらをアニメート
Starling.juggler.add ( mMovie );
Starling.juggler.add ( mMovieButcher );
```

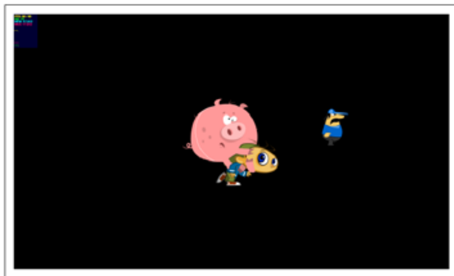


図 30：MovieClip は両方とも、同じスプライトシートからサンプリングしている（テクスチャアトラス）

ムービークリップを add すると、アニメートするようになります。もちろん再生は好きなときに一時停止や停止が行えます。

```
// 再生を一時停止、または停止する
mMovie.pause();
mMovie.stop();
```

一時停止と停止の違いは微妙です。pause API は、現在のフレーム位置のまま再生を一時停止します。一方 stop API は再生ヘッドを最初のフレームに再配置します。

では MovieClip オブジェクトで使用できる API 全体を見ておきましょう。Flash デベロッパーにおなじみのものに加え、実行時に特定のフレームレートに設定したり、フレームの置き換えやフレームが追加できる機能など、非常に有用なものもあります。

・ currentFrame: 現在のフレーム
・ fps: デフォルトの fps（1 秒間のフレーム数）。デュレーション（継続時間）を指定せずにフレームを追加するとき使用されます。
・ isPlaying: ムービーが現在再生中かどうかを示します。.
・ loop: ムービーがループしているかどうかを示します。
・ numFrames: クリップのフレーム数
・ totalTime: 全フレームの再生時間の合計
・ addFrame: 指定されたデュレーションでフレームを追加します。
・ addFrameAt: 指定されたインデックスにフレームを挿入します。
・ addFrame: デフォルトのデュレーションでフレームを追加します。
・ getFrameDuration: 特定のインデックスにあるフレームのデュレーション（秒単位）を返します。
・ getFrameSound: 特定のインデックスにあるフレームのサウンドを返します。
・ getFrameTexture: 特定のインデックスにあるフレームのテクスチャを返します。
・ pause: 再生を一時停止します。
・ play: 再生を開始します。クリップが必ず Juggler に追加されているようにします。
・ removeFrameAt: 指定されたインデックスにあるフレームを削除します。
・ setFrameDuration: 特定のフレームのデュレーションを秒単位で設定します。
・ setFrameSound: 特定のフレームがアクティブであるときに再生されるサウンドを設定します。
・ setFrameTexture: 特定のフレームのテクスチャを設定します。

以下ではこれらすべての例は取り上げず、非常に有用ものを見ていきます。たとえば実行時にフレームを追加したり、フレームを削除できる addFrameAt や removeFrameAt、特定のフレームにデュレーションを設定する API や、isPlaying や loop といったヘルパーAPI ももちろん見ていきます。

次のコードは、フレーム 5 を 2 秒のデュレーションに設定します。

```
// フレーム 5 は 2 秒の長さになる
mMovie.setFrameDuration(5, 2);
```

また指定したフレームにサウンドを動的に追加することもできます。

```
// フレーム 5 を 2 秒の長さにし、そこに到達したときサウンドを再生する
mMovie.setFrameDuration(5, 2);
mMovie.setFrameSound(5, new StepSound() as Sound);
```

これらの API によって、動的にロードしたり埋め込んだアセットから、ムービークリップ全体を実行時に組み立てることができます。これは実にパワフルな機能です。

こういった API（addFrameAt や removeFrameAt など）が必要になる一般的なケースは、1つの MovieClip 内に、アニメーションの複数の状態をグループ化したいときです。ネイティブの MovieClip API を使用すると、親の MovieClip の各フレームに MovieClip を置いて、あるフレームから別のフレームに切り替えるときに状態を再生することができます。Starling の MovieClip はコンテナでないので、希望する状態を再生するには、動的にフレームを変更する必要があります。

では最後に、ゲームでコントロールするときのように、ランニングボーイをキーボードで操作できるようにしましょう。

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

		import starling.animation.Juggler;
		import starling.core.Starling;
		import starling.display.MovieClip;
		import starling.display.Sprite;
```

```
			import starling.events.Event;
			import starling.events.KeyboardEvent;
			import starling.textures.Texture;
			import starling.textures.TextureAtlas;

			public class Game extends Sprite
			{
					private var mMovie:MovieClip;
					private var j:Juggler;

					[Embed(source="../media/textures/running-sheet.xml",
mimeType="application/octet-stream")]
public static const SpriteSheetXML:Class;

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

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

					private function onAdded (e:Event):void
					{
							var bitmap:Bitmap = new SpriteSheet();

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

							var xml:XML = XML(new SpriteSheetXML());

							var sTextureAtlas:TextureAtlas = new TextureAtlas(texture, xml);

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

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

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

				addChild ( mMovie );
				Starling.juggler.add ( mMovie );

				// キーダウン時
				stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
			}

			private function onKeyDown(e:KeyboardEvent):void
			{
				// 状態に応じて再配置
				if ( mMovie.scaleX == -1 )
					mMovie.x -= mMovie.width;

				// 右キーか左キーなら
				var state:int;
				if ( e.keyCode == Keyboard.RIGHT )
					state = 1;
				else if ( e.keyCode == Keyboard.LEFT )
					state = -1;

				// 左右反転
				mMovie.scaleX = state;

				// 状態に応じて再配置
				if ( mMovie.scaleX == -1 )
					mMovie.x = mMovie.x + mMovie.width;
			}
		}
}
```

アニメーションの終了（最終フレームへの到達）を監視する必要がある場合には、Event.COMPLETE イベントを監視します。

```
// アニメーションの終了の監視
mMovie.addEventListener(Event.MOVIE_COMPLETED, onAnimationComplete);
```

今の例では Juggler オブジェクトを使っています。次はこのオブジェクトの内部的な動作と、Juggler オブジェクトでできることを見ていきましょう。

Juggler（手品師の意）

Juggler API を使うと、IAnimatable インターフェイスを実装するオブジェクトがアニメートできます。MovieClip オブジェクトはこれを実装していますが、Starling 用のアニメーションするオブジェクトを自分で定義することもできます。そのために必要なのは IAnimatable インターフェイスを実装し、advanceTime メソッドをオーバーライドするだけです。パーティクル拡張はこの方法で動作します。これについては本チュートリアルの最後で見ていきます。

次のコードでは、MovieClip でアニメーションが行われる方法を見ることができ、メインのロジックはここにあります。毎フレーム、テクスチャが切り替えられます。ネイティブの API で言うと、この仕組みは Bitmap オブジェクトで使用される bitmapData を毎フレーム変更しているようなものです。

```
// IAnimatable
public function advanceTime(passedTime:Number):void
{
	if (mLoop && mCurrentTime == mTotalTime) mCurrentTime = 0.0;
	if (!mPlaying || passedTime == 0.0 || mCurrentTime == mTotalTime) return;
	var i:int = 0;
	var durationSum:Number = 0.0;
	var previousTime:Number = mCurrentTime;
	var restTime:Number = mTotalTime - mCurrentTime;
	var carryOverTime:Number = passedTime > restTime ? passedTime - restTime : 0.0;
	mCurrentTime = Math.min(mTotalTime, mCurrentTime + passedTime);

	for each (var duration:Number in mDurations)
	{
		if (durationSum + duration >= mCurrentTime)
		{
			if (mCurrentFrame != i)
			{
```

```
                mCurrentFrame = i;
                updateCurrentFrame();
                playCurrentSound();
            }
            break;
        }

        ++i;
        durationSum += duration;
    }

    if (previousTime < mTotalTime && mCurrentTime == mTotalTime &&
        hasEventListener(Event.MOVIE_COMPLETED))
    {
        dispatchEvent(new Event(Event.MOVIE_COMPLETED));
    }

    advanceTime(carryOverTime);
}
```

以下は Juggler API で使用できる API のリストです。

・ add: オブジェクトを juggler に追加します。
・ advanceTime: Juggler のメインループを手動で処理する場合に呼び出すよう意図された API。(すべてのオブジェクトを特定の時間(秒)だけ進めます)
・ delayCall: 特定のメソッドの実行を遅らせます。そのメソッドを呼び出すプロキシオブジェクトを返します。実行は指定された時間が経過するまで遅延します。
・ elapsedTime: juggler の存続時間の合計。
・ isComplete: Juggler の状態。
・ purge: すべてをオブジェクトを1度に削除します。
・ remove: オブジェクトを juggler から削除します。
・ removeTweens: 特定のターゲットを持つ Tween 型のすべてのオブジェクトを削除します。

juggler にはもう1つ、呼び出しを遅らせることができるという面白い機能があります。次のコードでは juggler を使って、子が親から削除されるタイミングを遅らせています。

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

あるケースでは、Juggler がもう1つ必要になるかもしれません。たとえばゲームのメインコンテンツが一時停止している間でもアニメーションを行いたいような場合です。一時停止しているゲームの上に重なっているメニューのような要素をアニメートするには、Juggler を使った方がよいでしょう。そのために必要なことはただインスタンスを作成し、その advanceTime API を呼び出すだけです。

そのためには、ゲームで使用するメインブロック（メニューや背景、プレイフィールドなど）ごとに Juggler を使用することで、ゲームを構成する必要があります。ユーザーが一時停止を押すとき、コンテンツ全体を一時停止したくはないのですが、Starling オブジェクトから stop API を呼び出すと、すべての描画呼び出しとすべてのフレームイベントが止まるので、コンテンツ全体が実際に一時停止することになります。

そうではなく、ゲームのメインブロックごとに別個の juggler で処理するようにします。ゲームを一時停止する必要があるときには、特定の（複数の）juggler だけを一時停止させます。こうすることで、一時停止または再開させる必要のあるゲーム部分が制御できます。

次のコードは、戦闘シーンを含んだカスタムの BattleScene の定義です。

```
package
{
    import starling.animation.Juggler;
    import starling.display.Sprite;

    public class BattleScene extends Sprite
    {
        private var juggler:Juggler;

        public function BattleScene()
        {
            juggler = new Juggler();
        }

        // この API は外部から呼び出す。
        // この呼び出しを止めると、このスプライト（BattleScene）の
        // この Juggler が再生するコンテンツが一時停止する
        public function advanceTime ( time:Number ):void
```

```
		{
			juggler.advanceTime( time );
		}

		public override function dispose():void
		{
			juggler.purge();
			super.dispose();
		}
	}
}
```

外部からは、EnterFrameEvent を使って advanceTime API を呼び出し、経過時間（paasedTime：直近のフレームから経過した秒数）を渡します。

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

ゲームが一時停止するときには、この advanceTime API の呼び出しを止めます。これにより戦闘シーンのコンテンツが止まります。もちろんメニューは上に重ねてアニメートする必要があるので、これを処理するため、次のようにします。

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

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

Boolean の paused を切り替えるだけで、魔法が起こります。

次は Starling のインタラクションにおいてもう1つの重要な部分である Button API を見ていきましょう。

Button

Starling はネイティブでボタンの概念をサポートしています。以下は Button コンストラクタのシグネチャです。

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

Button クラスはデフォルトで、ラベルをサポートする内部的な TextField を作成します。テキストはボタンの真ん中に置かれます。次のコードでは、スキンとして使用する埋め込みビットマップから、簡単なボタンを作成しています。

```
package
{

	import flash.display.Bitmap;

	import starling.display.Button;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.textures.Texture;

	public class Game extends Sprite
	{

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

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

		private function onAdded(e:Event):void
		{
			// 埋め込んだイメージから Bitmap オブジェクトを作成
			var buttonSkin:Bitmap = new ButtonTexture();
```

```
            // Button オブジェクトに与える Texture オブジェクトを作成
            var texture:Texture = Texture.fromBitmap(buttonSkin);

            // このスキンをアップ状態として使用するボタンを作成
            var myButton:Button = new Button(texture,"Play");

            // メニュー(複数のボタン)用コンテナを作成
            var menuContainer:Sprite = new Sprite();

            // ボタンをコンテナに追加
            menuContainer.addChild(myButton);

            // メニューをセンターに
            menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
            menuContainer.y = stage.stageHeight - menuContainer.height >> 1;

            // ボタンを表示
            addChild(menuContainer);
        }
    }
}
```

ここでは、スキン(変更可能な外見)として Button オブジェクトに与える必要のあるテクスチャの作成に fromBitmap API を使っています。

```
// Button オブジェクトに与える Texture オブジェクトを作成
var texture:Texture = Texture.fromBitmap(buttonSkin);
```

つづいて Vector に入れたデータから簡単なメニューを作成しましょう。次のコードでは単純なループでこれを行っています。

```
package
{

	import flash.display.Bitmap;

	import starling.display.Button;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.textures.Texture;

	public class Game extends Sprite
	{

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

		// セクション
		private	var	_sections:Vector.<String	>	=	Vector.<String	>
(["Play","Options","Rules","Sign in"]);

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

		private function onAdded(e:Event):void
		{
			// 埋め込んだイメージから Bitmap オブジェクトを作成
			var buttonSkin:Bitmap = new ButtonTexture();

			// Button オブジェクトに与える Texture オブジェクトを作成
			var texture:Texture = Texture.fromBitmap(buttonSkin);

			// メニュー(複数のボタン)用コンテナを作成
			var menuContainer:Sprite = new Sprite();
```

```
            var numSections:uint = _sections.length;

            for (var i:uint = 0; i< 4; i++)
            {
                // このスキンをアップ状態として使用するボタンを作成
                var myButton:Button = new Button(texture,_sections[i]);

                // ラベルを太字に
                myButton.fontBold = true;

                // ボタンの位置取り
                myButton.y = myButton.height * i;

                // ボタンをコンテナに追加
                menuContainer.addChild(myButton);
            }

            // メニューをセンターに
            menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
            menuContainer.y = stage.stageHeight - menuContainer.height >> 1;

            // ボタンを表示
            addChild(menuContainer);
        }
    }
}
```

このコードをテストすると、図 31 に示す結果が得られます。

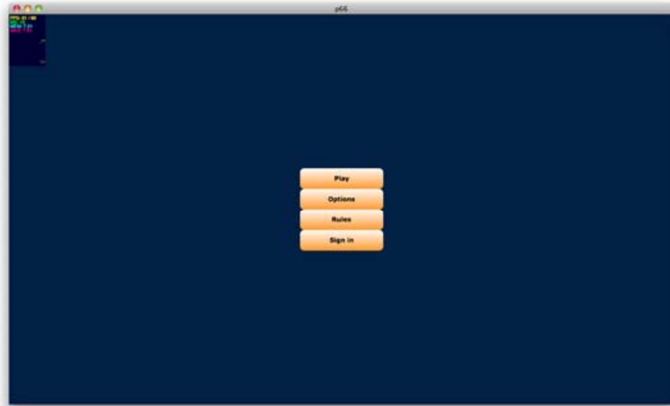


図 31：ボタンで作成した簡単なメニュー。マウスボタンを押し下げるとボタンは縮小し、上げるとボタンは元のサイズに戻る

しかし待ってください。ここではすべてのボタンのスキンにスプライトシートを使っていません。1つのスキンを埋め込み、Texture オブジェクトの fromBitmap API を通して GPU にアップロードしただけです。全部のボタンに単一のスキンを使用する場合にはこれで十分です。しかしベストプラクティスとしては、前に2つのムービークリップ（少年とブッチャー）で行ったように、すべてのスキンを単一のテクスチャアトラスに定義するようにします。

Button クラスには以下のプロパティがあります。

・ alphaWhenDisabled: ボタンが無効のとき使用されるアルファ値。
・ downState: ボタンがダウン状態（クリック時）にあるときに使用されるテクスチャ。
・ enabled: ボタンを機能させせることができるかどうか。
・ fontBold: ラベルのフォントでボールドスタイルを使うかどうか。
・ fontColor: フォントのカラー。
・ fontName: ボタンのラベルで使用するフォント。システムのフォントや登録したビットマップフォントが使用できます。
・ fontSize: ボタンのラベルで使用するフォントのサイズ。
・ scaleWhenDown: ボタンがタッチされたときの拡大縮小要因。ダウン状態のテクスチャが使用されているときには、ボタンは拡大縮小しません。
・ text: ボタンのラベルに使用されるテキスト。
・ textBounds: ボタンのラベルの位置。
・ upState: ボタンがタッチされていないとき使用されるテクスチャ。

Button オブジェクトは、ネイティブの Flash API と対照的に DisplayObjectContainer のサブクラスで、これ

は、ボタンのスキンに関する状態のプロパティの制約を受けないということを意味します。ボタンは、ほかのコンテナと同じように、好きなように装飾することができます。

Button オブジェクトはまた、クリック状態の処理で、Button 特有の Event.TRIGGERED イベントを送出します。

```
// Event.TRIGGERED イベントを監視
myButton.addEventListener(Event.TRIGGERED, onTriggered);

private function onTriggered(e:Event):void
{
	trace ("クリックされた!");
}
```

Event.TRIGGERED イベントはバブル（遡上）します。イベントの伝播を利用したい場合には、このイベントに依存し、そのコンテナのレベルでイベントが捕捉できます。

```
package
{

	import flash.display.Bitmap;

	import starling.display.Button;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.textures.Texture;

	public class Game extends Sprite
	{

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

		// セクション
		private var _sections:Vector.<String> = Vector.<String>
(["Play","Options","Rules","Sign in"]);
```

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

private function onAdded(e:Event):void
{
	// 埋め込んだイメージから Bitmap オブジェクトを作成
	var buttonSkin:Bitmap = new ButtonTexture();

	// Button オブジェクトに与える Texture オブジェクトを作成
	var texture:Texture = Texture.fromBitmap(buttonSkin);

	// メニュー(複数のボタン)用コンテナを作成
	var menuContainer:Sprite = new Sprite();

	var numSections:uint = _sections.length;

	for (var i:uint = 0; i< 4; i++)
	{
		// このスキンをアップ状態として使用するボタンを作成
		var myButton:Button = new Button(texture,_sections[i]);

		// ラベルを太字に
		myButton.fontBold = true;

		// ボタンの位置取り
		myButton.y = myButton.height * i;

		// ボタンをコンテナに追加
		menuContainer.addChild(myButton);
	}

	// Event.TRIGGERED イベントの捕捉
	menuContainer.addEventListener(Event.TRIGGERED, onTriggered);
```

```
                // メニューをセンターに
                menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
                menuContainer.y = stage.stageHeight - menuContainer.height >> 1;

                // ボタンを表示
                addChild(menuContainer);
            }

            private function onTriggered(e:Event):void
            {
                // 出力 : [object Sprite] [object Button]
                trace( e.currentTarget, e.target );
                // 出力 : triggered!
                trace("triggered!");
            }
        }
}
```

次は、アプリケーションのインターフェイスに、スクロールするテクスチャのような背景を加えてみましょう。

```
package
{

        import flash.display.Bitmap;

        import starling.display.Button;
        import starling.display.Image;
        import starling.display.Sprite;
        import starling.events.Event;
        import starling.textures.Texture;

        public class Game extends Sprite
        {

            [Embed(source = "../media/textures/sausage_normal.png")]
```

```
		private static const ButtonTexture:Class;

		[Embed(source = "../media/textures/background.jpg")]
		private static const BackgroundImage:Class;

		private var backgroundContainer:Sprite;

		private var background1:Image;
		private var background2:Image;

		// セクション
		private var _sections:Vector.<String> = Vector.<String>(["Play","Options","Rules","Sign in"]);

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

		private function onAdded(e:Event):void
		{
			// 埋め込んだイメージから Bitmap オブジェクトを作成
			var buttonSkin:Bitmap = new ButtonTexture();

			// Button オブジェクトに与える Texture オブジェクトを作成
			var texture:Texture = Texture.fromBitmap(buttonSkin);

			// 埋め込んだイメージから Bitmap オブジェクトを作成
			var background:Bitmap = new BackgroundImage();

			// Image オブジェクトに与える Texture オブジェクトを作成
			var textureBackground:Texture = Texture.fromBitmap(background);

			// 背景テクスチャ用コンテナ
			backgroundContainer = new Sprite();
```

```
// 背景用イメージの作成
background1 = new Image(textureBackground);
background2 = new Image(textureBackground);

// 2つめの背景の位置を調整（-3 は訳者による”ずる”）
background2.x = background1.width -3;

// これらをネスト
backgroundContainer.addChild(background1);
backgroundContainer.addChild(background2);

// 背景を表示
addChild(backgroundContainer);

// メニュー(複数のボタン)用コンテナを作成
var menuContainer:Sprite = new Sprite();

var numSections:uint = _sections.length;

for (var i:uint = 0; i< 4; i++)
{
	// このスキンをアップ状態として使用するボタンを作成
	var myButton:Button = new Button(texture,_sections[i]);

	// ラベルを太字に
	myButton.fontBold = true;

	// ボタンの位置取り
	myButton.y = myButton.height * i;

	// ボタンをコンテナに追加
	menuContainer.addChild(myButton);
}

// Event.TRIGGERED イベントの捕捉;
menuContainer.addEventListener(Event.TRIGGERED, onTriggered);
```

コードを実行すると、前面にメニューがあり、背景がスクロールします（図 32 参照）。

```
            // 毎フレーム
            stage.addEventListener(Event.ENTER_FRAME, onFrame);

            // メニューをセンターに;
            menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
            menuContainer.y = stage.stageHeight - menuContainer.height >> 1;

            // ボタンを表示
            addChild(menuContainer);
        }

        private function onTriggered(e:Event):void
        {
            // 出力 : [object Sprite] [object Button]
            trace( e.currentTarget, e.target );
            // 出力 : triggered!
            trace("triggered!");
        }

        private function onFrame(e:Event):void
        {
            // 背景コンテナをスクロール（右に移動）
            backgroundContainer.x -=  10;
            // リセット
            if (backgroundContainer.x <=  -  background1.width)
            {
                backgroundContainer.x = 0;
            }
        }
    }
}
```

コードを実行すると、前面にメニューがあり、背景がスクロールします（図 32 参照）。

図 32:メニューとスクロールする背景

背景イメージがスクロールするときの移動感を強調するため、イメージには Photoshop を使って少しモーションブラー—を適用しています。

> 訳注:
> 原書ではサンプルファイルが提供されていないので、図 32 は、原書から撮ったイメージを元に作成した素材を使った結果です。原書の図と比べ、この図の背景は少し横長で、ソーセージは寸胴に見えます。またソーセージの文字は少し上にずれ、黒色です。

TextField

starling.text.TextField API は前に四角形をいじったときに使っています。ここでは少し時間をとって、Starling のテキストを見ていきましょう。みなさんは GPU がどうやってフォントをレンダリングするのか不思議に思われるかもしれませんが、ここにはトリックがあります。Starling は舞台裏で、CPU 乗にネイティブの TextField オブジェクトを作成し、それをオフスクリーンバッファ(あらかじめ用意されたメモリ上のビットマップ)として使用してフォントをレンダリングします。ラスタライズされたテクスチャが GPU にアップロードされると、画面にテキストとして表示されることになります。

TextField API は、使用する starling.text.TextField ごとに、ネイティブの flash.text.TextField インスタンスを作成するわけではありません。インスタンスはキャッシュされ、すべてのテキストのレンダリングに再利用されます。

次のコードでは、TextField オブジェクトを作成し、Verdana システムフォントを使ってテキストを表示しています。

```
package
{
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.text.TextField;

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

		private function onAdded(e:Event):void
		{
			// TextField オブジェクトの作成
			var legend:TextField = new TextField(300,300,"Here is some text, using an system font!","Verdana",38,0xFFFFFF);

			// テキストをステージセンターに
			legend.x = stage.stageWidth - legend.width >> 1;
			legend.y = stage.stageHeight - legend.height >> 1;

			// 表示
			addChild(legend);
		}
	}
}
```

図 33 はこのコードの実行結果です。

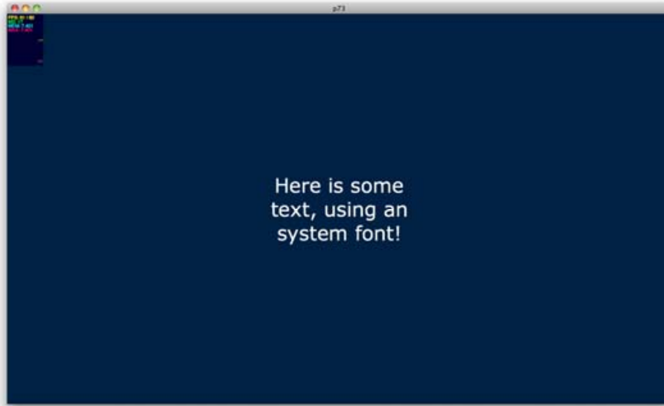


図 33：単純なテキスト

繰り返しになりますが、この画面で見ているのは実際のテキストフィールドではなく、ビットマップテクスチャに描画され GPU にアップロードされた、テキストフィールドのスナップショットです。

TextField API には以下のプロパティがあります。

- alpha: テキストのアルファ。
- autoScale: テキストを、テキストブロックのサイズに収まるように自動的に拡大縮小させます。
- bold: ラベルのフォントがボールドスタイルを使用するかどうか。
- border: テキストフィールドの周囲に境界線を表示します。視覚的なデバッグ作業に便利です。
- bounds: 親のローカル座標を基準にしたこのテキストフィールド範囲。
- color: テキストのカラー。
- fontName: フォントの名前。
- fontSize: フォントのサイズ。
- hAlign: テキストの水平方向の整列（center か left か right）。
- italic: テキストを斜体にするかどうか。
- kerning: ビットマップフォントでカーニング情報を有効にするかどうか（可能な場所で）。デフォルトは true。
- text: 表示するテキスト。
- textBounds: テキストフィールド内の実際の文字の範囲。
- underline: テキストに下線をつけるかどうか。
- vAlign: テキストの垂直方向の整列（bottom か center か top）。

TextField のすばらしい機能の1つは autoScale プロパティです。これについてはこの後見ていきます。しかしその前に、いくつかのプロパティを使ってみましょう。次のコードではテキストの周囲に境界線をつけ、テキストを太字にしてカラーを変えています。

```
// TextField オブジェクトを作成
var legend:TextField = new TextField(300, 300, "Here is some text, using an embedded font!", "Verdana", 38, 0xFFFFFF);

// カラーを変更し、太字にして、境界線を有効にする
legend.color = 0x990000;
legend.bold = true;
legend.border = true;
```

図 34 はこの結果を示しています。

図 34：色付けをした簡単なテキスト

TextField オブジェクトはネイティブでは HTML テキストを処理しません。しかし Starling の今後のバージョンではこういった機能も導入されるかもしれません。追加してほしい機能がある場合はぜひ Starling フォーラムで声を上げてください。Starling はオープンソースなので、こういった機能はみなさんが自分で実装して、それを機能拡張としてコミュニティに提供することもできます。

前の例では最も一般的なシステムフォントを使用しました。これはこれで実に有用ですが、今日の多くのプロジェクトでは埋め込みフォントを使用する必要があります。たとえばゲームの場合で言うと、ブランドを強く印象づける経験を与えたい（タイトルを強く印象づけたい）ので、埋め込みフォントが必要になります。

埋め込みフォント

Starling は、ネイティブの TextField API のように embedFonts プロパティを公開していませんが、心配いりません。実際には埋め込みフォントの使用は簡単です。まず予想されているように、必要なことはフォントの埋め込みか、実行時のフォントのロードです。次いでそのフォント名を TextField コンストラクタに渡します。

次のコードでは、フォントを埋め込みそれをインスタンス化して、その fontName プロパティを TextField コンストラクタに渡しています。

```
package
{
	import flash.text.Font;

	import starling.display.Sprite;
	import starling.events.Event;
	import starling.text.TextField;

	public class Game extends Sprite
	{
		[Embed(source = '/../media/fonts/Abduction.ttf',embedAsCFF = 'false',fontName = 'Abduction')]
		public static var Abduction:Class;

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

		private function onAdded(e:Event):void
		{
			// フォントの作成
			var font:Font = new Abduction();

			// TextField オブジェクトの作成
			var legend:TextField = new TextField(300,300,"Here is some text, using an embedded font!",font.fontName,38,0xFFFFFF);

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

			addChild(legend);
```

```
            }
        }
}
```

TextField オブジェクトは自動的に埋め込みフォントをその名前から見つけ、それを使用します。図 35 は埋め込んだ Abduction フォントを使ったテキストを表示しています（Abduction フォントは http://www.dailyfreefonts.com/fonts/info/5699-Abduction-III.html からダウンロードできます）。

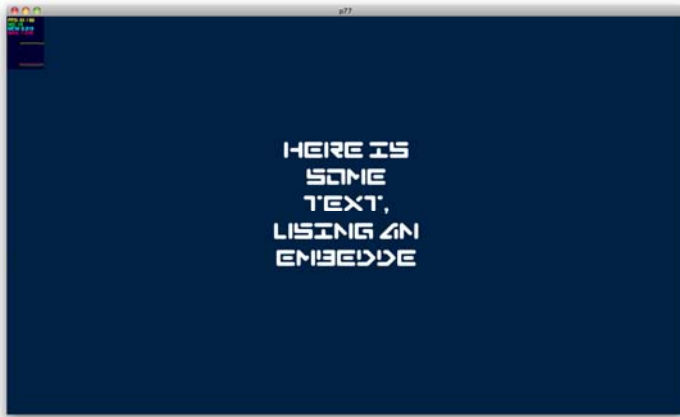


図 35：埋め込みフォントを使った簡単なテキスト

実に簡単でしょ？ みなさんのコンテンツでは、テキストを入力する方法、たとえばユーザー名やメールアドレス、そのほかの情報といったものを入力する方法を提供する場合があるでしょう。想像されている通り、GPU でのテキストの編集はトリッキーです。以下に述べるトリックは、ほとんどのプラットフォームで（モバイルでさえも）で使用できます。この場合のアイデアは単純で、ただいつも使用してきたネイティブの表示リストを使用するだけです。

表示リストは Stage3D シーンの上にあるので、テキスト入力フィールドは GPU コンテンツの上に置くことができます。そのためには、ネイティブオーバーレイという Starling の非常に便利な機能を利用します。

wmode 値が不正確に使用されているとき、Starling がエラーメッセージを表示することは前に見ました。Starling は内部的にこのネイティブオーバーレイ機能に依存しています。Starling は Stage3D の上で動作するので、このネイティブオーバーレイ機能を使用すると、Starling オブジェクトから表示リストにアクセスし、Flash で常に使用してきたネイティブの表示リストにオブジェクトを追加して、Stage3D コンテンツの上に、ビデオやテキスト入力といったネイティブ要素を重ねることができます。

Starling は、間違った wmode 値が使用されていると判定すると、内部の showFatalError 関数を呼び出し、Stage3D の上に警告メッセージを表示します。この場合、レンダリングする GPU サーフェスはないので、

Starling は Stage3D の上にある表示リストに依存しているのは明らかです。

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

次のコードでは、テキスト入力フィールドを作成し、それをStarlingコンテンツの上の表示リストに追加しています。

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

これは、たとえばゲームのスタート時さまざなま情報を入力するときなど、テキスト入力が必要な場合に極めて役立ちます。またネイティブのステージ（flash.display.Stage）には、Starling オブジェクトの nativeStage からいつでもアクセスできます。

```
// flash.display.Stage からネイティブのフレームレートにアクセスする
trace ( Starling.current.nativeStage.frameRate );
```

次は Starling で扱うフォントの中で最高のパフォーマンスが得られるビットマップフォントです。

ビットマップフォント

テクスチャは舞台裏で作成されるので、リソースと GC ロードの観点から、最高のパフォーマンスと最小のコストを考えると、TexiField API にはビットマップフォントを使用すべきです。これは、フォントのグリフ（文字のイメージ）をスプライトシートに写し、このテクスチャをサンプリングして必要なグリフをレンダリングする、という考えです。

図 36 は、Britannic Bold フォントのグリフのスプライトシートを作成している GlyphDesigner ツールです（商用、http://glyphdesigner.71squared.com/）。

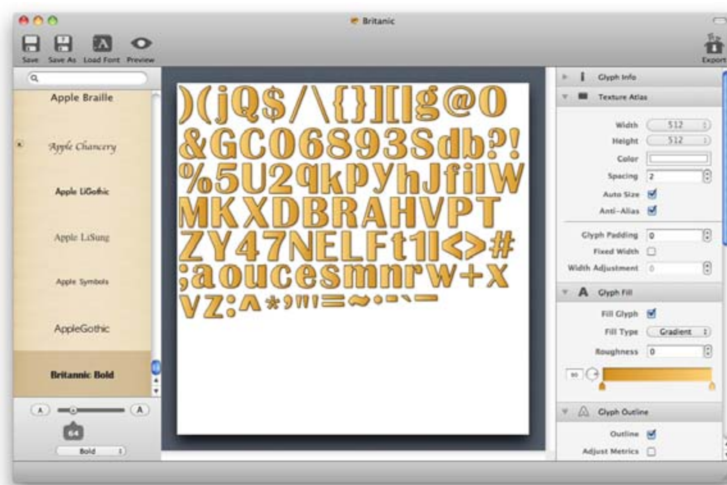


図 36：Mac OS 用の GlyphDesigner

Windows には Bitmap Font Generator（http://www.angelcode.com/products/bmfont/）という同種のフリーツールがあります（図 37 参照）。

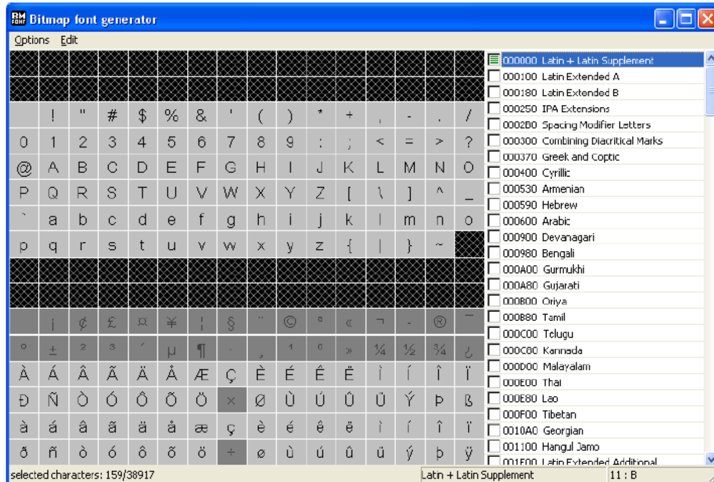


図 37：Windows の Bitmap Font Generator

ただし真面目な話、使うなら Mac OS ツールの方です。

もちろん、実行時にこのグリフのスプライトシートを生成する方法でもかまいません。各グリフを含む TextField を使い、スプライトシートにグリフをレイアウトし、BitmapData API を使ってそれを描画するという方法です。この方法と以下に述べる埋め込む方法のどちらを選ぶかは、プラットフォーム（デスクトップかモバイル）によって変わるでしょう。実行時に行う方法はサイズの理由からコストが高くつくでしょうし、埋め込む方法はコンテンツの起動が速くなるでしょう。決めるのはみなさんです。

書き出しが終わると、テクスチャがイメージとして保存され、イメージ上での各グリフの位置の詳細を記述した次のようなファイルが.fnt ファイル（XML、つまりテキスト）として作成されます。

```
<font>
    <info face="BritannicBold" size="64" bold="0" italic="0" chasrset="" unicode="0" stretchH="100" smooth="1" aa="1" padding="0,0,0,0" spacing="2,2"/>
    <common lineHeight="64" base="53" scaleW="512" scaleH="512" pages="1" packed="0"/>
    <pages>
        <page id="0" file="britannic-bold.png"/>
    </pages>
    <chars count="95">
        <char id="41" x="2" y="2" width="24" height="60" xoffset="1" yoffset="8" xadvance="23" page="0" chnl="0" letter=")"/>
        <char id="40" x="28" y="2" width="24" height="60" xoffset="1" yoffset="8"
```

```
xadvance="23" page="0" chnl="0" letter="("/>
		<char id="106" x="54" y="2" width="22" height="60" xoffset="-3" yoffset="8"
xadvance="18" page="0" chnl="0" letter="j"/>
...
	</chars>
</font>
```

訳注：
書き出しは GlyphDesigner 画面右上にある[Export]ボタンで行います。[Export type]では[.fnt (Sparrow XML)]を選びます。

ビットマップテクスチャ（britannic-bold.png）とフォント記述ファイル（britannic-bold.fnt）が書き出せたら、これまでと同じように埋め込みます。

```
[Embed(source = "../media/fonts/britannic-bold.png")]
private static const BitmapChars:Class;

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

これらを使用するには、次の静的 TextField API を使用します。

- registerBitmapFont: ビットマップフォントを登録します。
- unregisterBitmapFont: ビットマップフォントの登録を解除。

フォントテクスチャとその記述ファイルを BitmapFont オブジェクトに渡し、registerBitmapFont API を通してこれを登録します。登録したら、TextField オブジェクトを作成するときにフォント名を渡します。

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

	import starling.display.Sprite;
	import starling.events.Event;
	import starling.text.BitmapFont;
	import starling.text.TextField;
```

```
	import starling.textures.Texture;
	import starling.utils.Color;

	public class Game extends Sprite
	{
		[Embed(source = "../media/fonts/britannic-bold.png")]
		private static const BitmapChars:Class;

		[Embed(source = "../media/fonts/britannic-bold.fnt",mimeType = "application/octet-stream")]

		private static const BritannicXML:Class;

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

		private function onAdded(e:Event):void
		{
			// 埋め込みビットマップ（スプライトシートファイル）を作成
			var bitmap:Bitmap = new BitmapChars();

			// ビットマップからテクスチャを作成
			var texture:Texture = Texture.fromBitmap(bitmap);

			// スプライトシート上でのグリフの位置を記述した XML ファイルを作成
			var xml:XML = XML(new BritannicXML());

			// ビットマップフォントを登録し、TextField で使用できるようにする
			TextField.registerBitmapFont(new BitmapFont(texture, xml));

			// TextField オブジェクトを作成
			var bmpFontTF:TextField = new TextField(400,400,"Here is some text, using an embedded font!","BritannicBold",10);
```

```
            // 拡大縮小せず元のフォントサイズを使用
            bmpFontTF.fontSize = BitmapFont.NATIVE_SIZE;

            // テクスチャはそのまま、色合いを変えずに使用
            bmpFontTF.color = Color.WHITE;

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

            addChild(bmpFontTF);
        }
    }
}
```

図 38 はこの結果を示しています。

図 38：ビットマップフォントを使ってレンダリングしたテキスト

次はストリングを少し長くし、結果がどうなるかを見てみましょう。

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

案の定、テキストボックスの境界ボックスはストリング全体を表示するには小さすぎ、テキストは図 39 に示すように切り取られ、適切に表示されません。

図 39：大きなビットマップテキストは切り取られる

ありがたいことに、Starling では TextField の autoScale プロパティが公開されています。

```
// テキストをボックスにうまく納める
bmpFontTF.autoScale = true;
```

このプロパティは、ゲームでストリングをローカライズする必要があり（日本語版に加え英語版も作成するなど）、テキストを特定のボックスに確実に納めたい場合に非常に便利です。ほとんどの場合デザイン的な理由から、レイアウトをまったく変えずストリングがその長さに関係なくうまく揃うように、テキストは少し拡大縮小することになるでしょう。

図 40 は autoScale プロパティを有効にした結果です。ストリングが少し変更されています。

図 40：境界ボックスにフィットするように、少し縮小された大きなテキスト

前のメニューとスクロールする背景のサンプルは、ビットマップフォントで次のように改良することができます。

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

	import starling.display.Button;
	import starling.display.Image;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.text.BitmapFont;
	import starling.text.TextField;
	import starling.textures.Texture;

	public class Game extends Sprite
	{
		[Embed(source = "../media/textures/sausage_normal.png")]
		private static const ButtonTexture:Class;

		[Embed(source = "../media/textures/background.jpg")]
		private static const BackgroundImage:Class;

		[Embed(source = "../media/fonts/hobo-std.png")]
		private static const BitmapChars:Class;

		[Embed(source          =          "../media/fonts/hobo-std.fnt",mimeType          =
"application/octet-stream")]
		private static const Hobo:Class;

		private static const FONT_NAME:String = "HoboStd";

		private var backgroundContainer:Sprite;

		private var background1:Image;
		private var background2:Image;
```

```
// セクション
private var sections:Vector.<String> = Vector.<String>
(["Play","Options","Rules","Sign in"]);

public function Game()
{
    addEventListener(Event.ADDED_TO_STAGE, onAdded);
}
private function onAdded(e:Event):void
{
    // 埋め込みビットマップ（スプライトシートファイル）を作成
    var bitmap:Bitmap = new BitmapChars();

    // ビットマップからテクスチャを作成
    var texture:Texture = Texture.fromBitmap(bitmap);

    // スプライトシート上でのグリフの位置を記述した XML ファイルを作成
    var xml:XML = XML(new Hobo());

    // ビットマップフォントを登録し、TextField で使用できるようにする
    TextField.registerBitmapFont(new BitmapFont(texture, xml));

    // 埋め込んだイメージから Bitmap オブジェクトを作成
    var buttonSkin:Bitmap = new ButtonTexture();

    // Button オブジェクトに与える Texture オブジェクトを作成
    var textureSkin:Texture = Texture.fromBitmap(buttonSkin);

    // 埋め込んだイメージから Bitmap オブジェクトを作成
    var background:Bitmap = new BackgroundImage();

    // Image オブジェクトに与える Texture オブジェクトを作成
    var textureBackground:Texture = Texture.fromBitmap(background);

    // 背景テクスチャ用コンテナ
    backgroundContainer = new Sprite();
```

```
// 背景用のイメージを作成
background1 = new Image(textureBackground);
background2 = new Image(textureBackground);

// 2つめのイメージの位置取り
background2.x = background1.width - 3;

// これらをネスト
backgroundContainer.addChild(background1);
backgroundContainer.addChild(background2);

// 背景を表示
addChild(backgroundContainer);

// メニュー（複数のボタン）用コンテナを作成
var menuContainer:Sprite = new Sprite();

var numSections:uint = sections.length;

for (var i:uint = 0; i< numSections; i++)
{
    // このスキンをアップ状態とするボタンを作成
    var myButton:Button = new Button(textureSkin,sections[i]);

    // フォント名
    myButton.fontName = FONT_NAME;
    myButton.fontColor = 0xFFFFFF;

    // テキストの位置取り
    myButton.textBounds = new Rectangle(10,38,160,30);

    // フォントサイズ
    myButton.fontSize = 26;

    // ボタンの位置取り
```

```
			myButton.y = (myButton.height-10) * i;

			// ボタンをコンテナに追加
			menuContainer.addChild(myButton);
		}

		// Event.TRIGGERED イベントの捕捉
		menuContainer.addEventListener(Event.TRIGGERED, onTriggered);

		// 毎フレーム
		stage.addEventListener(Event.ENTER_FRAME, onFrame);

		// メニューをセンターに
		menuContainer.x = stage.stageWidth - menuContainer.width >> 1;
		menuContainer.y = stage.stageHeight - menuContainer.height >> 1;

		// ボタンを表示
		addChild(menuContainer);
	}

	private function onTriggered(e:Event):void
	{
		// 出力 : [object Sprite] [object Button]
		trace( e.currentTarget, e.target );
		// 出力 : triggered!
		trace("triggered!");
	}

	private function onFrame(e:Event):void
	{
		// 背景を右にスクロール
		backgroundContainer.x -=  10;
		// リセット
		if (backgroundContainer.x <=  -  background1.width)
		{
			backgroundContainer.x = 0;
```

```
                    }
                }
        }
}
```

図 41 はこの結果を示しています。

図 41：カスタムフォントをメニューで使用した

訳注：
ここで使用している HoboStd フォントは http://fonts.zaraf.ro/font2281/hobostd.otf.htm からダウンロードできます。

次はレンダーテクスチャと呼ばれる、Starling のもう1つのすばらしい機能を見ていきます。

RenderTexture

starling.textures.RenderTexture を使用すると、非破壊的な（前の描画を保持する）描画が作成できます。Flash デベロッパーのみなさんは BitmapData API を思い出してください。この機能は、テクスチャ内でそれまでの描画を保ちつつ連続して描画する必要のある、お絵かきツールのようなアプリケーションの作成に非常に役立ちます。

次のコードでは、BitmapData.draw の機能を、Starling を使って GPU 上で再現しています。

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

```
import flash.geom.Point;

import starling.display.Image;
import starling.display.Sprite;
import starling.events.Event;
import starling.events.Touch;
import starling.events.TouchEvent;
import starling.events.TouchPhase;
import starling.textures.RenderTexture;
import starling.textures.Texture;

public class Game extends Sprite
{
    private var mRenderTexture:RenderTexture;
    private var mBrush:Image;

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

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

    private function onAdded(e:Event):void
    {
        // 埋め込んだイメージから Bitmap オブジェクトを作成
        var brush:Bitmap = new Egg();

        // Image オブジェクトに与える Texture オブジェクトを作成
        var texture:Texture = Texture.fromBitmap(brush);

        // テクスチャに描画するテクスチャを作成
        mBrush = new Image(texture);

        // 基準点をセンターに設定する
```

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

        // 必要二応じて拡大縮小
        //mBrush.scaleX = mBrush.scaleY = 0.5;

        // 描画するキャンバスを作成
        mRenderTexture = new RenderTexture(stage.stageWidth,stage.stageHeight);

        // それを Image オブジェクトにカプセル化する
        var canvas:Image = new Image(mRenderTexture);

        // 表示
        addChild(canvas);

        // ステージ上のマウスの相互作用を監視
        stage.addEventListener(TouchEvent.TOUCH, onTouch);
    }

    private function onTouch(event:TouchEvent):void
    {
        // タッチポイント全体を取得（タッチ画面での複数の指によるタッチに備えて）
        var touches:Vector.<Touch >    = event.getTouches(this);

        for each (var touch:Touch in touches)
        {
            // HOVER かクリック状態ならスキップする
            if (touch.phase == TouchPhase.HOVER || touch.phase == TouchPhase.ENDED)
                continue;

            // マウスか各指の位置を取得
            var location:Point = touch.getLocation(this);

            // 描画するブラシの位置
            mBrush.x = location.x;
            mBrush.y = location.y;
```

```
                // キャンバスに描画
                mRenderTexture.draw(mBrush);
            }
        }
    }
}
```

マウスを押し下げたまま移動させると、図 42 に示すような結果が得られます。

図 42：非破壊的な描画

トゥイーン

Starling はそれ自体にトゥイーンエンジンを備え、図 43 に示す、通常使用されるほとんどのイージング式をサポートします。

次のコードは、テキストフィールドの x と y プロパティを跳ね返り効果でトゥイーンします。

```
// Tween オブジェクトを作成
var t:Tween = new Tween(bmpFontTF, 4, Transitions.EASE_IN_OUT_BOUNCE);

// アニメーションの設定
t.moveTo(bmpFontTF.x+300, bmpFontTF.y);

// トゥイーンを Juggler に追加
```

Starling.juggler.add(t);

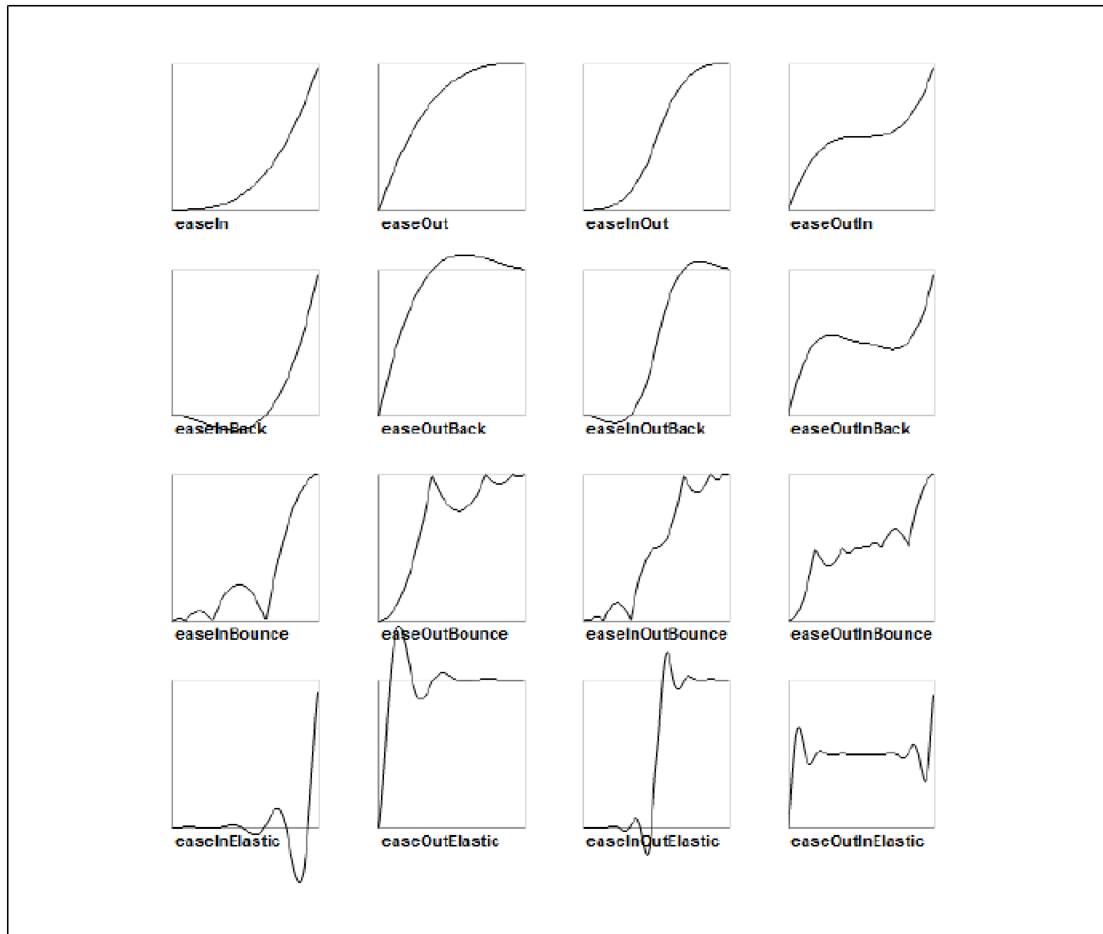


図 43：Starling で使用できるイージング式（sparrow-framework.org からの引用）

以下は Tween オブジェクトで使用できる API のリストです。

・ animate: オブジェクトのプロパティをターゲット値までアニメートします。このメソッドは、1つのトゥイーンで複数回呼び出すことができます。
・ currentTime: トゥイーンが作成されてからたった時間。
・ delay: トゥイーンが開始されるまでの遅延。
・ fadeTo: オブジェクトのアルファプロパティをターゲット値までアニメートします。このメソッドは、1つのトゥイーンで複数回呼び出すことができます。
・ isComplete: トゥイーンが完了したかどうか。
・ moveTo: オブジェクトの x と y プロパティを同時にアニメートします。
・ onComplete: トゥイーンが完了したとき呼び出されるコールバックへの参照。
・ onCompleteArgs: トゥイーンが完了したとき呼び出されるコールバックに渡すパラメータ

・ onStart: トゥイーンがスタートしたとき呼び出されるコールバックへの参照。
・ onStartArgs: トゥイーンがスタートしたとき呼び出されるコールバックに渡すパラメータ。
・ onUpdate: トゥイーンの進行中に呼び出されるコールバックへの参照。
・ onUpdateArgs: トゥイーンの進行中に呼び出されるコールバックに渡すパラメータ。
・ roundToInt: 数値を整数にキャストするかどうか。
・ scaleTo: オブジェクトの scaleX と scaleY プロパティを同時にアニメートします。
・ target: アニメートされるターゲットのオブジェクト。
・ totalTime: トゥイーンにかかる合計時間（秒単位）。
・ transition: アニメーションに使用するトランジションの方法。

次のコードでは、トゥイーンの完了を監視し、onCompleteArgs API を通して破棄引数を渡すことで、アニメーションするオブジェクトを破棄しています。

```
// Tween オブジェクトの作成
var t:Tween = new Tween(bmpFontTF, 4, Transitions.EASE_IN_OUT_BOUNCE);

// アニメーションの設定
t.moveTo(bmpFontTF.x+300, bmpFontTF.y);
t.animate("alpha", 0);

// Juggler に追加
Starling.juggler.add(t);

// 完了時、ステージからテキストフィールドを削除する
t.onComplete = bmpFontTF.removeFromParent;

// removeFromParent 呼び出しに破棄引数を渡す
t.onCompleteArgs = [true];
```

次のコードでは、onStart、onUpdate、onComplete イベントにそれぞれコールバックを割り当て、トゥイーンの進行状況を監視しています。

```
package
{
	import flash.text.Font;
```

```
	import starling.animation.Transitions;
	import starling.animation.Tween;
	import starling.core.Starling;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.text.TextField;

	public class Game extends Sprite
	{
		[Embed(source = '/../media/fonts/Abduction.ttf',embedAsCFF = 'false',fontName = 'Abduction')]

		public static var Abduction:Class;

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

		private function onAdded(e:Event):void
		{
			// フォントを作成
			var font:Font = new Abduction();

			// TextField オブジェクトの作成
			var legend:TextField = new TextField(300,300,"Here is some text, using an embedded font!",font.fontName,38,0xFFFFFF);

			// テキストをステージセンターに
			legend.x = stage.stageWidth - legend.width >> 1;
			legend.y = stage.stageHeight - legend.height >> 1;

			// Tween オブジェクトの作成
			var t:Tween = new Tween(legend,4,Transitions.EASE_IN_OUT_BOUNCE);

			// アニメーションの設定
```

```
            t.moveTo(legend.x+300, legend.y);

            // Juggler に追加
            Starling.juggler.add(t);

            // スタートの監視
            t.onStart = onStart;
            // 進行状況の監視
            t.onUpdate = onProgress;
            // 完了の監視
            t.onComplete = onComplete;

            // 表示
            addChild(legend);
        }

        private function onStart():void
        {
            trace("tweening start");
        }

        private function onProgress():void
        {
            trace("tweening in progress");
        }

        private function onComplete():void
        {
            trace("tweening complete");
        }
    }
}
```

TweenLite やそのほかの有名な既存のトゥイーンライブラリも Starling で動作するはずです。中には微調整が必要なものもあるかもしれませんが、それでも Starling に組み込むのはそう難しくはないはずです。次はアセットをもっと組織化した方法で処理する方法を見ていきます。埋め込みが必要なとき、これまではコード

内で単純な Embed タグを使ってきましたが、大きなプロジェクトでは、アセット全部を中心的な場所に置くようになります。次の節では、アセットをグループ化しそれらを再利用するための簡単なテクニックを見ていきます。

アセット管理

ここまでアセットは非常に簡単な方法で使用してきました。アセットの使用方法を最適化するには、Assets オブジェクトの機能を中心的な場所として使って、そこからリソースを取得することが推奨されます。Assets オブジェクトは使用するアセットのプーリング（貯留）を担い、これにより、廃棄し再度インスタンス化して GC の作動を促すのではなく、確実に再利用できるようになります。

次のコードでは、Assets オブジェクトに、埋め込んだテクスチャを取得する getTexture API を定義しています。

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

ここではアセットの保持に Dictionary を使っています。これにより次に必要になったときに、再作成するのではなく、プールから取得することができます。

テクスチャは通常、Embed タグを使って埋め込みます。

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

Starling では必ずしも埋め込みテクスチャを使わなければならないわけではなく、テクスチャは Loader オブジ

ェクトを使って動的にロードすることもできます。

```
// Loader の作成
var loader:Loader = new Loader();

// Event.COMPLETE イベントの監視
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );

// イメージをロード
loader.load ( new URLRequest ("texture.png" ) );

function onComplete ( e:Event ):void
{
	// Bitmap を作成
	var bitmap:Bitmap = e.currentTarget.data;

	// Bitmap からテクスチャを作成
	var texture:Texture = Texture.fromBitmap(bitmap);

	// Image オブジェクトを作成
	var image:Image = new Image (texture);

	// イメージを表示
	addChild (image);
}
```

動的なテクスチャは BitmapData オブジェクトから生成することもできます。これまでは作成済みのビットマップを使ってきましたが、テクスチャは動的に作成することもできます。

そのためには、Texture クラスの静的な fromBitmapData API を使用します。

```
// 動的なビットマップの作成
var dynamicBitmap:BitmapData = new BitmapData (512, 512);
// ライブラリからカスタムのベクターシェイプを使用するか、それも実行時に描画しそれを使う
dynamicBitmap.draw ( myCustomShape );
```

```
// BitmapData からテクスチャを作成
var texture:Texture = Texture.fromBitmapData(dynamicBitmap);

// Image オブジェクトを作成
var image:Image = new Image (texture);

// イメージを表示
addChild (image);
```

Starling アプリケーションはモバイルデバイスやデスクトップブラウザで実行される可能性があるので、さまざまな画面サイズで表示されることになります。Starling では画面のリサイズ（サイズ変更）を簡単に処理することができます。次はこれについて見ていきます。

画面のリサイズの処理

Flash デベロッパーは通常、画面のリサイズを、Event.RESIZE という簡単なイベントに依存して処理します。このイベントを使用すると、ページサイズが変わるたびに通知を受け取ることができます。その後 stage.stageWidth と stage.stageHeight プロパティを使用すると、ターゲットにする画面サイズに関係なく、コンテンツを適切にレイアウトすることができます。

これを処理するため、Starling の starling.display.Stage も同種の ResizeEvent.RESIZE イベントを送出します。これを使用するとリサイズを動的に処理することができます。

次のコードは最初のサンプルの更新版で、四角形をつねにステージセンターにレイアウトします。

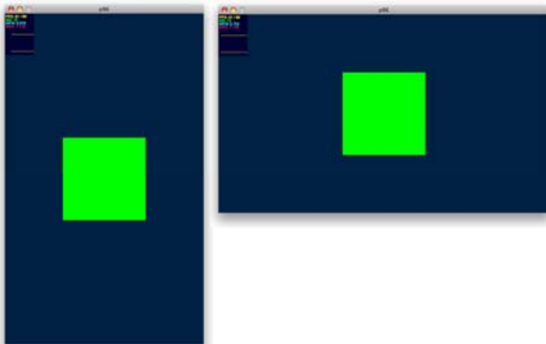
```
package
{
	import flash.geom.Rectangle;


	import starling.core.Starling;
	import starling.display.Quad;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.events.ResizeEvent;


	public class Game extends Sprite
```

```
{
	private var q:Quad;
	private var rect:Rectangle = new Rectangle(0,0,0,0);

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

	private function onAdded( e:Event ):void
	{
		// イベントの監視
		stage.addEventListener(ResizeEvent.RESIZE, onResize);

		q = new Quad(200,200);
		q.color = 0x00FF00;
		q.x = stage.stageWidth - q.width >> 1;
		q.y = stage.stageHeight - q.height >> 1;
		addChild( q );
	}

	private function onResize(event:ResizeEvent):void
	{
		// 変数 rect のサイズを設定
		rect.width = event.width,rect.height = event.height;

		// ビューポートをリサイズ
		Starling.current.viewPort = rect;

		// 新しいステージサイズを割り当て
		stage.stageWidth = event.width;
		stage.stageHeight = event.height;

		// 四角形の位置を変更
		q.x = stage.stageWidth - q.width >> 1;
		q.y = stage.stageHeight - q.height >> 1;
```

```
            }
        }
}
```

SWF がリサイズされると毎回、ResizeEvent.RESIZE が送出されます。新しいサイズは ResizeEvent オブジェクトを通して送信されるので、ここでは手動でその値を stageWidth と stageHeight プロパティに書き込んでいます。コンテンツの位置は修正します。ステージサイズをベースにしたアプリケーションのコンテンツはどれも、正確にレイアウトできます。

訳注：
下図は AIR for Android での実行結果です。ステージサイズはデフォルトの 480 x 800 です。アプリケーション記述ファイルでは<renderMode>を direct に設定します（AIR 3 の機能）。adl のメニューで[Device]→[Rotate Left]を選択すると、画面サイズが横長に変わります（図の右）。緑の四角形はつねにステージセンターにあることが分かります。



Starling と Robotlegs

みなさんの中にはすでに、あの卓越した Robotlegs フレームワーク（http://www.robotlegs.org/）を使って、クリーンで効率的なコードの構築に役立てている方のいらっしゃるかもしれません。Omar Gonzalez は Starling で Robotlegs の使用をサポートするプラグインを作成しました。詳細は https://github.com/s9tpepper/robotlegs-starling-plugin を参照してください。

Starling と Box2D

Starling の見事な点は、既存のフレームワークを容易に組み入れることができる表示リスト API にあります。たとえば、ゲームに物理的な動きを追加する Box2D フレームワークを使ってみたくはありませんか？

Box2D の情報の取得と、次の例で使用するライブラリのダウンロードは、http://box2dflash.sourceforge.net/から行えます。

次のコードでは、重力のある状態でボックスが地面に落ちます。もちろんすべては GPU を通してレンダリングされます。

```
package
{
	import Box2D.Collision.Shapes.b2CircleShape;
	import Box2D.Collision.Shapes.b2PolygonShape;
	import Box2D.Common.Math.b2Vec2;
	import Box2D.Dynamics.b2Body;
	import Box2D.Dynamics.b2BodyDef;
	import Box2D.Dynamics.b2FixtureDef;
	import Box2D.Dynamics.b2World;

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

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

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

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

```
private function onAdded(e:Event):void
{
	// 重力ベクトルの定義
	var gravity:b2Vec2 = new b2Vec2(0.0,10.0);

	// 剛体を眠らせる(アクティブでない剛体をシミュレートせず、パーフォーマンスを上げる)
	var doSleep:Boolean = true;

	// ワールドオブジェクトを作成
	m_world = new b2World(gravity,doSleep);

	// 剛体の作成に使用する変数
	var body:b2Body;
	var boxShape:b2PolygonShape;
	var circleShape:b2CircleShape;

	// 地面剛体の追加
	bodyDef = new b2BodyDef();
	//bodyDef.position.Set(15, 19);
	bodyDef.position.Set(10, 28);
	//bodyDef.angle = 0.1;
	boxShape = new b2PolygonShape();
	boxShape.SetAsBox(30, 3);
	var fixtureDef:b2FixtureDef = new b2FixtureDef();
	fixtureDef.shape = boxShape;
	fixtureDef.friction = 0.3;
	// 動かない剛体の密度は 0
	fixtureDef.density = 0;

	// スプライトを剛体の userData に追加
	var box:Quad = new Quad(2000,200,0xCCCCCC);
	box.pivotX = box.width / 2.0;
	box.pivotY = box.height / 2.0;

	bodyDef.userData = box;
```

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

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

		var quad:Quad;

		// オブジェクトを追加
		for (var i:int = 1; i < 100; i++)
		{
			bodyDef = new b2BodyDef();
			bodyDef.type = b2Body.b2_dynamicBody;
			bodyDef.position.x = Math.random() * 15 + 5;
			bodyDef.position.y = Math.random() * 10;
			var rX:Number = Math.random() + 0.5;
			var rY:Number = Math.random() + 0.5;

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

			// 四角形の作成
			quad = new Quad(100,100,Math.random() * 0xFFFFFF);
			quad.pivotX = quad.width / 2.0;
			quad.pivotY = quad.height / 2.0;

			// これはキーの行。userDataとして starling.display.Quad を渡す
			bodyDef.userData = quad;
			bodyDef.userData.width = rX * 2 * 30;
			bodyDef.userData.height = rY * 2 * 30;
```

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

                    // 各四角形を表示
                    addChild(bodyDef.userData);
                }

                // 毎フレーム
                addEventListener(Event.ENTER_FRAME, Update);
            }

            public function Update(e:Event):void
            {
                // ワールドを実行
                m_world.Step(m_timeStep, m_velocityIterations, m_positionIterations);
                m_world.ClearForces();

                // 剛体のリストを走査して、スプライトの位置、回転を更新
                for (var bb:b2Body = m_world.GetBodyList(); bb; bb = bb.GetNext())
                {
                    // これもキーの行。starling.display.DisplayObject  を見つけたら、フィジックスを適
用する
                    if (bb.GetUserData() is DisplayObject)
                    {
                        // ネイティブの DisplayObject でなく、Starling の DisplayObject としてキャスト
                        var sprite:DisplayObject = bb.GetUserData() as DisplayObject;
                        sprite.x = bb.GetPosition().x * 30;
                        sprite.y = bb.GetPosition().y * 30;
                        sprite.rotation = bb.GetAngle();
                    }
                }
                bodyDef.position.Set(10, 28);
            }
        }
}
```

このコードをテストすると、図 44 に示すような結果が得られます。

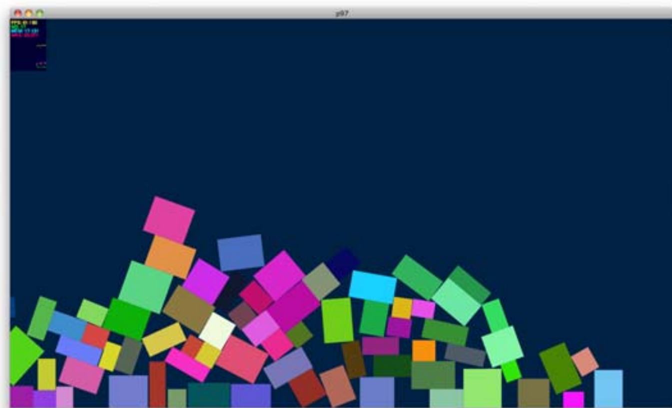


図 44：Starling で Box2D を使用した

訳注：
図 44 は Flash CS5.5 の adl によるプレビューではなく、AIR のデスクトップアプリケーションとして実行した結果です。当然ながらこの方が FPS はアップします。

このコンテンツは Stage3D をサポートするモバイル AIR の次バージョンでも動作します（図 45 参照）。

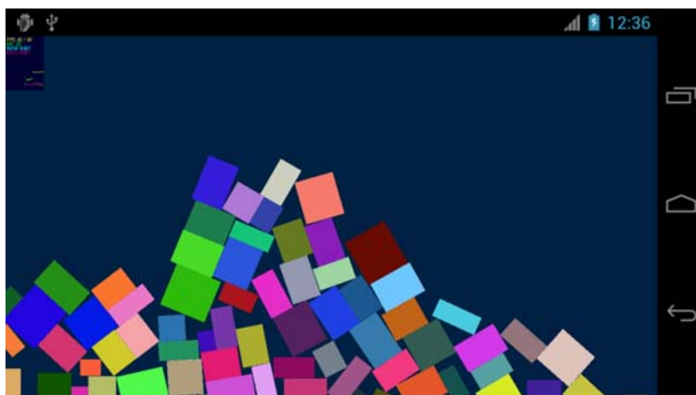


図 45：Galaxy Nexus での実行結果

もちろんこれらの四角形はテクスチャで置き換えることもできます。そうするとさらに表現豊かなコンテンツが作成できます。人々を驚かせるような 2D GPU コンテンツの構築はみなさんの肩にかかっているのです。

Starling のプロファイリング

次はパフォーマンスのプロファイリング（調べる）の話です。すべてのコンテンツで最初から、そして必ず必要になるのが FPS のベンチマーク（評価テスト）です。通常デベロッパーはデバッグ作業の間つねにデバッグ情報を表示して、パフォーマンスが落ちる場所やよくなる領域を監視します。

ここまで本チュートリアルでは、mr.doob の有名な Stats クラスを使ってきました（https://github.com/mrdoob/Hi-ReS-Stats）。

Stats クラスはこのまま使いつづけることもできますが、もちろん物事には例外があり、ここではモバイルがそれに当たります。モバイル AIR の今後のバージョンでは、Stage3D が使用できるようになります。Android などのプラットフォームのモバイルデバイスで、表示リストを Stage3D サーフェスと同時に使用して実行すると、大きなパフォーマンスヒット（余分にかかる負荷）の危険が高まります。参考までに言うと、GPU の中には、表示リストを使用すると、因数 2 だけパフォーマンスを落とすものがあり、したがってコンパイルされたフレームレート 60 を使用すると、表示リストに 30fps、Stage3D コンテンツに 30fp 当てられるということになります。その結果、デバッグという単一の目的であっても、すべてを Stage3D 上で実行することが重要になります。

ビットマップフォントの概念を学んだみなさんは、これを使ってフレームレートを Stage3D 上で表示する FPS のクラスを開発することができます。

図 46 はグリフのスプライトシートを書きだそうとしているところです。スプライトシートをできるだけ軽量にするため、ここでは FPS カウンタに使用するグリフだけを選択している点に注目してください（FPS0123456789.-を選択）。

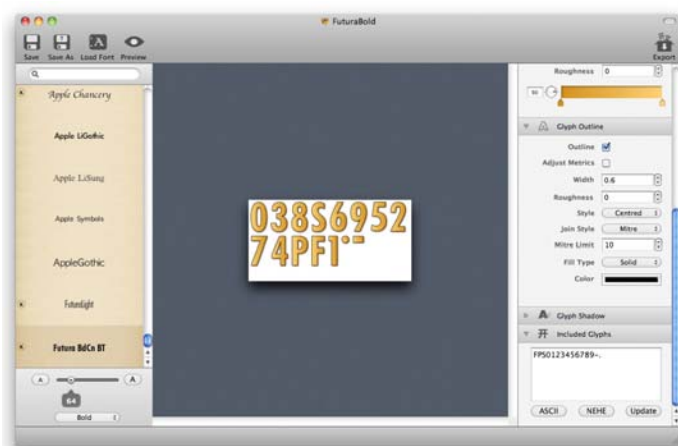


図 46：FPS カウンタ用のテキストテクスチャ

訳注：
原書では Futura フォントが使用されていますが、ここでは Futura BdCn BT を使っています。

スプライトシートとグリフの記述ファイルが書き出せたら、TextField オブジェクトと BitmapFont を使って FPS を描画します。

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

	import starling.display.Sprite;
	import starling.events.Event;
	import starling.text.BitmapFont;
	import starling.text.TextField;
	import starling.textures.Texture;
	import starling.utils.Color;

	public class FPS extends Sprite
	{
		private var container:Sprite = new Sprite	;
		private var bmpFontTF:TextField;

		private var frameCount:uint = 0;
		private var totalTime:Number = 0;

		private static var last:uint = getTimer();
		private static var ticks:uint = 0;
		private static var text:String = "--.- FPS";

		[Embed(source = "../media/fonts/futura-fps.png")]
		private static const BitmapChars:Class;

		[Embed(source = "../media/fonts/futura-fps.fnt",mimeType = "application/octet-stream")]
		private static const BritannicXML:Class;

		public function FPS()
```

```
		{
			addEventListener(Event.ADDED_TO_STAGE,onAdded);
		}

		private function onAdded(e:Event):void
		{
			// 埋め込みビットマップを作成（スプライトシートファイル）
			var bitmap:Bitmap = new BitmapChars;

			// ビットマップからテクスチャを作成
			var texture:Texture = Texture.fromBitmap(bitmap);

			// スプライトシート上のグリフの位置を記述した XML ファイルを作成
			var xml:XML = XML(new BritannicXML    );

			// TextField で使えるようにビットマップフォントを登録
			TextField.registerBitmapFont(new BitmapFont(texture,xml));

			// オブジェクトを作成
			// bmpFontTF = new TextField(70,70,"... FPS","Futura-Medium",12);
			bmpFontTF = new TextField(70,70,"... FPS","FuturaBT-BoldCondensed",12);

			// 枠線
			bmpFontTF.border = true;

			// テクスチャはそのまま使用（色合いを変更しない）
			bmpFontTF.color = Color.WHITE;

			// 表示
			addChild(bmpFontTF);

			// 毎フレーム
			stage.addEventListener(Event.ENTER_FRAME,onFrame);
		}

		public function onFrame(e:Event):void
```

```
        {
            ticks++;
            var now:uint = getTimer();
            var delta:uint = now - last;
            if ((delta >= 1000))
            {
                var fps:Number = ticks / delta * 1000;
                text = fps.toFixed(1) + " FPS";
                ticks = 0;
                last = now;
            }
            bmpFontTF.text = text;
        }
    }
}
```

使い方は簡単で、ただ次のコードを Starling ワールド内のどこかに記述するだけです。

```
// FPS カウンタを表示
addChild ( new FPS() );
```

図 47 は動作中の FPS カウンタです。

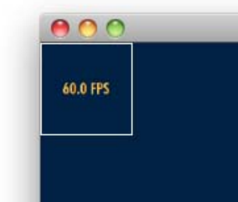


図 47：GPU を通してレンダリングされる FPS カウンタ

前の物理エンジンのデモを更新して FPS カウンタを組み入れることができます（図 48 参照）。

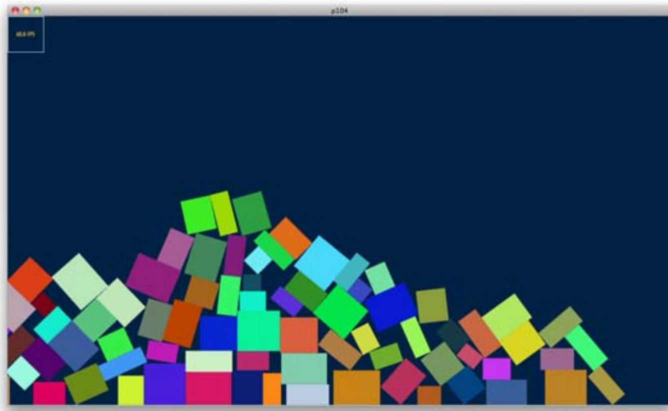


図 48：アプリケーションに FPS カウンタを組み込んだ

これでモバイルでもコンテンツのパフォーマンスのプロファイリングが行える方法を手に入れることができました。前に述べたように、この解決方法によって表示リストを完全に取り除いて、メニューの UI アニメーションだけでなく、フレームレートのようなデバッグ情報も表示できるようになります。

mr.doob の Stats クラスは新たに Starling に移植され、http://forum.starling-framework.org/topic/starling-port-of-mrdoobs-stats-class で見つけることができます。

パーティクル

美しいエフェクトの話となると、わたしは断然パーティクルが好きです。信じるか信じないかはみなさんの自由ですが、パーティクルはその美しい見栄えと裏腹に、実はさほど複雑ではありません。専門的にいうと、パーティクルはテクスチャをつけた多くの四角形が、特有のブレンドモードを使って動き回り、美しい複合体を成したものです。

Starling でパーティクルを設計するには、ParticleDesigner という名前のツールが非常に便利です（GlyphDesigner と同じ 71 squared 社の製品、http://particledesigner.71squared.com/）。図 49 は ParticleDesigner のスナップショットです。右のエミュレーションモードでは、作成中のパーティクルがプレビューできます。

メインウィンドウにはたくさんのパラメータが並んでいます。希望するパーティクルを作成すべくこれらのパラメータをいじり始めると、あっという間に時間がたってしまいます。実はパラメータを自動的に生成する[Randomize]ボタンがあり、さまざまなパーティクルのエフェクトを作成することができます。

図 49：Mac OS の ParticleDesigner

図 50 は ParticleDesigner の[Save As]ダイアログです。ここではパーティクルをレンダリングするための ParticleEmitter ファイル（.pex）とテクスチャを書き出すことができます（[Embed texture]チェックボックスはオフにします）。この2つのファイルは次で取り上げる ParticleDesignerPS オブジェクトに与えるために使用するファイルです。

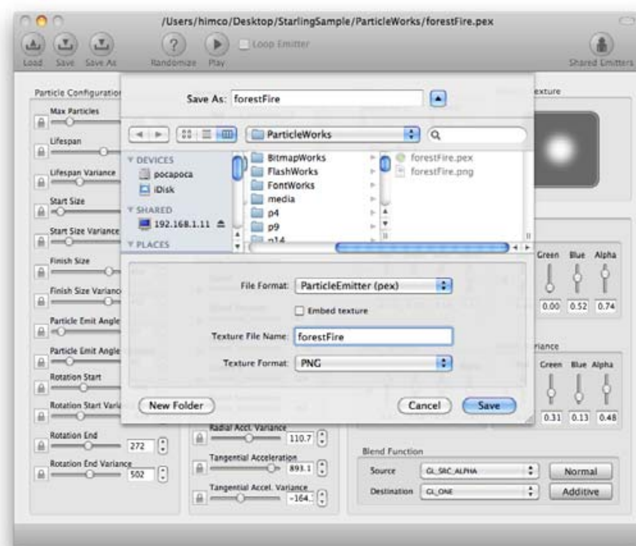


図 50：.pex ファイルと.png ファイルを保存

訳注：
ParticleDesigner は非常に安価なアプリケーションですが、Web ベースで同様のファイルが作成できる Particle Editor（http://onebyonedesign.com/flash/particleeditor/）という選択肢もあります。

図 51 は ParticleDesigner と Starling の実行から作成したパーティクルの例です。

図 51：Starling で実行したカスタムのパーティクル

きれいですよね？　しかしこれを見ようとしても、Starling にはパーティクル機能はついていません。その通り、パーティクルエンジンは実際には Starling の機能拡張で、https://github.com/PrimaryFeather/Starling-Extension-Particle-System からダウンロードできます。

図 52 は美しいパーティクルのもう1つの例です。

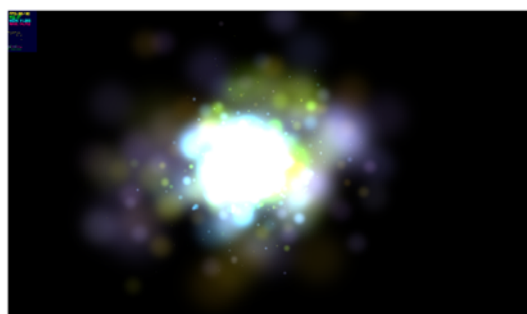


図 52：Starling で実行したカスタムのパーティクル

Starling のパーティクルは、Starling でアニメーションするほかのオブジェクトと同じであることをよく覚えておいてください。つまりパーティクルのアニメーションを表示するには、パーティクルオブジェクトを Juggler に追加する必要があるのです。

```
// XML コンフィグファイルのロード
var psConfig:XML = XML(new StarConfig());

// パーティクルテクスチャの作成
var psTexture:Texture = Texture.fromBitmap(new StarParticle());
```

```
// テクスチャと XML 記述ファイルからパーティクルシステムを作成
mParticleSystem = new ParticleDesignerPS(psConfig, psTexture);

// パーティクルの開始位置の設定
mParticleSystem.emitterX = 800;
mParticleSystem.emitterY = 240;

// パーティクルの開始
mParticleSystem.start();

// 表示
addChild(mParticleSystem);

// アニメートする
Starling.juggler.add(mParticleSystem);
```

パーティクルは好きな位置に置くことができます。ParticleDesignerPS オブジェクトは実際には DisplayObject なので、期待されるすべての機能が使用できます。もちろんいつも通り、終了時には juggler からパーティクルを削除し、ParticleDesignerPS オブジェクトで dispose API を呼び出すことでパーティクルを破棄する必要があります。

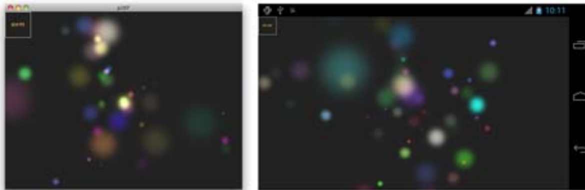
訳注：
以下は Starling-Extension-Particle-System のデモを参考にわたしが作成したクラスです。図 50 で作成したパーティクルを再生します。

```
package
{

	import starling.core.Starling;
	import starling.display.Sprite;
	import starling.events.Event;
	import starling.extensions.ParticleDesignerPS;
	import starling.textures.Texture;

	public class Game extends Sprite
	{
```

```
		// ParticleDesigner の XML ファイルの埋め込み
		[Embed(source = "../media/textures/forestFire.pex",mimeType = "application/octet-stream")]
		private static const ForestFireConfig:Class;

		// ParticleDesigner の PNG ファイルの埋め込み
		[Embed(source = "../media/textures/forestFire.png")]
		private static const ForestFireParticle:Class;

		// 変数
		private var mParticleSystem:ParticleDesignerPS;

		public function Game()
		{
			// GPU 版 FPS カウンタの設置
			addChild(new FPS());

			// ParticleDesigner の XML と PNG ファイルを作成
			var psConfig:XML = XML(new ForestFireConfig());
			var psTexture:Texture = Texture.fromBitmap(new ForestFireParticle());

			// パーティクルシステムの作成と設定、開始
			mParticleSystem = new ParticleDesignerPS(psConfig, psTexture);
			mParticleSystem.emitterX = 320;
			mParticleSystem.emitterY = 240;
			mParticleSystem.start();
			addChild(mParticleSystem);

			addEventListener(Event.ADDED_TO_STAGE, onAddedToStage);
			addEventListener(Event.REMOVED_FROM_STAGE, onRemovedFromStage);
		}

		private function onAddedToStage(event:Event):void
		{
			// アニメーション開始
			Starling.juggler.add(mParticleSystem);
```

```
            }

            private function onRemovedFromStage(event:Event):void
            {
                // 後始末
                Starling.juggler.remove(mParticleSystem);
                mParticleSystem.dispose();
            }
        }
}
```

下図左は Flash CS5.5 の adl での、右は Galaxy Nexus での実行結果です。



なお Starling のバージョンが古いと、ParticleSystem.as の 182 行で、オーバーライドに対応していません、という実行時エラーが発生します。この場合には、本ドキュメント冒頭で述べているように、Starling ライブラリを GitHub ページからダウンロードして使用します。

Lee Brimelow（http://www.leebrimelow.com）は、パーティクルを使って宇宙船の噴射をシミュレートするサンプルを作成しています（図 53 参照）。ビデオチュートリアルとサンプルコードは http://www.gotoandlearn.com/play.php?id=151 から得られます。

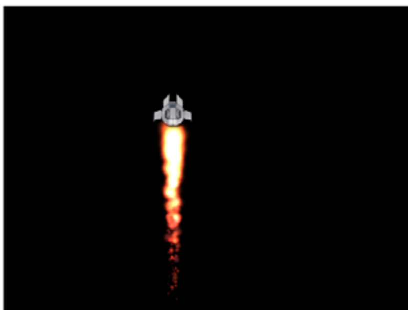


図 53：パーティクルエフェクトを宇宙船に組み込んだ

さらに図 54 に示すように、宇宙船から発射されるロケットにも小さなパーティクルを追加できます。

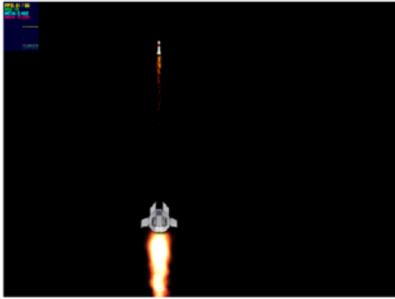


図 54：宇宙船からロケットを発射

ロケットがステージの外に出たら、juggler とシーンから削除します。

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

実はここでは、パーティクルを破棄するとい非常に重要な事柄が抜け落ちています。これは GPU のメモリから削除されないということを意味しています。したがってコードは次のように修正すべきです。

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

また、パーティクルシステムを削除し、同時に破棄できる removeChild API を使うこともできます。

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

前の例では、パーティクルはロケットに割り当てられ、画面の外へ出ていきます。パーティクルの現在の位置を調べ、画面の境界から出たときにそれらを破棄するのは難しいことではありません。

しかし、パーティクルがランダムな位置で爆発するものの、画面の外へ出るのではなく、画面内で見えなくなる、という場合も考えられます。このような場合には、パーティクルの位置を調べても意味はなく、パーティクルのアニメーションの完了を調べて、アニメーションが完了したタイミングでそれを破棄する必要があります。

ありがたいことに、パーティクルシステムの完了を知らせてくれるイベントがあります。必要なときに、ParticleSystem オブジェクトで Event.COMPLETE イベントを監視するだけです。

```
mParticleSystem.addEventListener(Event.COMPLETE, onParticleComplete);
```

> 訳注：
> ParticleDesignerPS は ParticleSystem のサブクラスです。

実にシンプルです。以上ですべて準備は整いました！

われわれの Starling の紹介はこれで終わりです。Starling 上でどのようなすばらしいコンテンツを作成するかはみなさん次第です。