ADOBE® FLASH® PLATFORM の
パフォーマンスの最適化

法律上の注意

法律上の注意については、http://help.adobe.com/ja_JP/legalnotices/index.html を参照してください。

コンテンツ

第 1 章：はじめに

ランタイムコードの実行の基礎 . . . 1
認知パフォーマンスと実際のパフォーマンス . . . 2
最適化の対象決定 . . . 3

第 2 章：メモリの節約

表示オブジェクト . . . 4
プリミティブ型 . . . 4
オブジェクトの再利用 . . . 5
メモリの解放 . . . 10
ビットマップの使用 . . . 12
フィルターおよびビットマップの動的アンロード . . . 18
ダイレクトミップマッピング . . . 19
3D 効果の使用 . . . 19
テキストオブジェクトとメモリ . . . 21
イベントモデルとコールバック . . . 21

第 3 章：CPU 使用率の最小化

Flash Player 10.1 における CPU 使用率の強化機能 . . . 22
スリープモード . . . 24
オブジェクトのフリーズとフリーズ解除 . . . 25
イベントのアクティブ化と非アクティブ化 . . . 28
マウスの操作 . . . 29
タイマーと ENTER_FRAME イベント . . . 29
トゥイーン使用による問題 . . . 31

第 4 章：ActionScript 3.0 のパフォーマンス

Vector クラスと Array クラス . . . 32
描画 API . . . 33
イベントキャプチャとイベントバブリング . . . 34
ピクセルの操作 . . . 35
正規表現 . . . 37
その他の最適化 . . . 37

第 5 章：レンダリングのパフォーマンス

再描画領域 . . . 42
ステージ外のコンテンツ . . . 43
ムービーの画質 . . . 44
アルファブレンディング . . . 46

アプリケーションのフレームレート 47
ビットマップキャッシュ 48
手動によるビットマップキャッシュ 55
テキストオブジェクトのレンダリング 61
GPU 65
非同期操作 68
透明なウィンドウ 69
ベクターシェイプのスムージング 70

第 6 章：ネットワーク通信の最適化
ネットワーク通信の強化機能 72
外部コンテンツ 73
入出力エラー 76
Flash Remoting 77
不要なネットワーク操作 78

第 7 章：メディアの操作
ビデオ 80
StageVideo 80
オーディオ 80

第 8 章：SQL データベースのパフォーマンス
データベースのパフォーマンスためのアプリケーションデザイン 82
データベースファイルの最適化 85
不要なデータベースランタイム処理 85
効率的な SQL 構文 86
SQL ステートメントのパフォーマンス 87

第 9 章：ベンチマークおよびデプロイ
ベンチマーク 88
デプロイ 89

第 1 章：はじめに

Adobe® AIR® および Adobe® Flash® Player アプリケーションは、デスクトップ、モバイルデバイス、タブレット、テレビ端末など、様々なプラットフォームで実行されます。このドキュメントでは、コード例と使用例を通じて、これらのアプリケーションをデプロイする開発者向けにベストプラクティスを示します。このドキュメントのトピックは以下のとおりです。

- メモリの節約
- CPU 使用率の最小化
- ActionScript 3.0 のパフォーマンスの強化
- レンダリング速度の向上
- ネットワーク通信の最適化
- オーディオとビデオの操作
- SQL データベースのパフォーマンスの最適化
- ベンチマークアプリケーションとデプロイアプリケーション

これらの最適化のほとんどは、あらゆるのデバイス上のアプリケーションに適用され、AIR ランタイムと Flash Player ランタイムの両方を対象としています。特定のデバイスに関する追加事項と例外事項についても説明します。

これらの最適化の一部は、Flash Player 10.1 および AIR 2.5 で導入された機能を対象としています。ただし、これらの最適化のほとんどは、以前の AIR および Flash Player リリースにも適用されます。

ランタイムコードの実行の基礎

アプリケーションのパフォーマンスを改善する方法を理解するために重要な点の 1 つは、Flash Platform ランタイムでコードを実行する方法を理解することです。ランタイムは、各「フレーム」で発生する特定のアクションと共にループで実行されます。この場合のフレームとは、アプリケーションに指定されたフレームレートによって決まる、単なる時間のブロックです。各フレームに割り当てられる時間数は、フレームレートに直接関連します。例えば、30 フレーム / 秒のフレームレートを指定すると、ランタイムは最後の 30 分の 1 秒で各フレームの作成を試行します。

アプリケーションの初期フレームレートはオーサリング時に指定します。フレームレートは、Adobe® Flash® Builder™ または Flash Professional の設定を使用して設定できます。また、コードで初期フレームレートを指定することもできます。ActionScript のみのアプリケーションでフレームレートを設定するには、[SWF(frameRate="24")] メタデータタグをルートドキュメントクラスに適用します。MXML では、Application または WindowedApplication タグで frameRate 属性を設定します。

各フレームループは 2 フェーズで構成され、3 つのパートに分割されます。イベント、enterFrame イベント、およびレンダリングです。

第 1 フェーズには 2 つのパート（イベントおよび enterFrame イベント）が含まれ、そのどちらもコードで呼び出される場合があります。第 1 フェーズの第 1 パートでは、ランタイムイベントが到着し、送出されます。これらのイベントで、ネットワークでのデータのロードからの応答など、非同期操作の完了または進行を表すことができます。これらのイベントには、ユーザー入力からのイベントも含まれます。イベントが送出されると、登録したリスナーのコードがランタイムで実行されます。イベントが発生しない場合、アクションは実行されず、ランタイムはこの実行フェーズが完了するまで待機しますアクティビティがないので、フレームレートは上がりません。実行サイクルの他の部分でイベントが発生すると、それらのイベントはキューに格納され、次のフレームで送出されます。

第 1 フェーズの第 2 パートは enterFrame イベントです。このイベントが他のイベントと異なる点は、フレームごとに 1 度必ず送出される点です。

すべてのイベントが送出されると、フレームループのレンダリングフェーズが開始されます。その時点で、画面上のすべての可視エレメントの状態が計算され、それらのエレメントが画面に描画されます。その後で、競技場を周回する走者のように、プロセスが繰り返されます。

注意：updateAfterEvent プロパティが含まれるイベントについては、レンダリングフェーズを待たずに直ちにレンダリングを行うよう強制できます。ただし、updateAfterEvent によるパフォーマンス低下がひんぱんに発生する場合は、このプロパティを使用しないでください。

フレームループの 2 つのフェーズにかかる時間が同じになる場合を想定することは最も容易です。その場合、各フレームループの半分では、イベントハンドラーとアプリケーションコードが実行され、もう半分ではレンダリングが発生していると考えられます。ただし、多くの場合、実際にはこのような状況になっていません。フレームで使用できる時間のうち半分を超える時間がアプリケーションコードで使用されて、時間割り当てが増え、レンダリングに使用できる時間割り当てが減る場合があります。また、特にフィルターやブレンドモードなどの複雑な可視コンテンツでは、レンダリングがフレーム時間の半分よりも長くかかる場合もあります。フェーズにかかる実際の時間は柔軟なので、フレームループは一般的に「弾性レーストラック」と呼ばれます。

フレームループの組み合わせた操作（コードの実行とレンダリング）に時間が長くかかり過ぎる場合、ランタイムはフレームレートを維持できません。フレームは拡張され、割り当てられた時間よりも長くかかるので、次のフレームがトリガーされるまで遅延が発生します。例えば、フレームループにかかる時間が 1/30 秒を上回る場合、30 フレーム / 秒で画面を更新することはできません。フレームレートが低速になると、操作性が低下します。アニメーションが途切れ途切れになることや、状況によってはアプリケーションがフリーズし、ウィンドウが空になることがあります。

Flash Platform ランタイムコードの実行およびレンダリングモデルについて詳しくは、次のリソースを参照してください。

- 「Flash Player Mental Model - The Elastic Racetrack」（Ted Patrick による記事）
- 「Asynchronous ActionScript Execution」（Trevor McCauley による記事）
- 「Optimizing Adobe AIR for code execution, memory & rendering」（http://www.adobe.com/go/learn_fp_air_perf_tv_jp）（Sean Christmann による MAX 会議プレゼンテーションのビデオ）

認知パフォーマンスと実際のパフォーマンス

アプリケーションのパフォーマンスが適切かどうかの最終的な判断は、アプリケーションのユーザーによって決まります。開発者は、特定の操作にかかる実行時間や、作成されるオブジェクトのインスタンス数という点でアプリケーションのパフォーマンスを測定できます。ただし、このようなメトリクスはエンドユーザーにとって重要ではありません。ユーザーはパフォーマンスを別の基準で判断することがあります。例えば、アプリケーションはすばやく円滑に動作し、入力に対して迅速に応答しているでしょうか。また、アプリケーションがシステムのパフォーマンスに悪影響を及ぼしてはいないでしょうか。次の設問に自分で答えることにより、認知パフォーマンスを確認してください。

- アニメーションはスムーズですか、それとも途切れ途切れですか。
- ビデオコンテンツはスムーズですか、それとも途切れ途切れですか。
- オーディオクリップは連続再生されますか、それとも一時停止して再開しますか。
- 時間がかかる操作のときにウィンドウは迅速に動作しますか、それとも空になりますか。
- 入力速度にテキスト入力は追いつきますか、それとも遅延がありますか。
- クリックすると処理は即時に実行されますか、それとも遅延がありますか。
- このアプリケーションの実行時に CPU のファン音は大きくなりますか。
- ラップトップコンピュータまたはモバイルデバイスの場合、このアプリケーションを実行するとすぐにバッテリ切れになりますか。

- このアプリケーションの実行時に、他のアプリケーションの応答速度は低下しますか。

認知パフォーマンスと実際のパフォーマンスの区別は重要です。最高の認知パフォーマンスを達成する方法は、絶対的なパフォーマンスを最速にする方法と必ずしも同じではありません。画面の更新とユーザー入力の読み取りを十分な頻度で実行できなくなるほど多量のコードは決して実行しないようにします。場合によっては、このバランスを取るために、プログラムタスクを複数のパートに分割することでパートの間で画面を更新します（詳しくは、42 ページの「レンダリングのパフォーマンス」を参照してください。）。

ここで説明するヒントとテクニックは、実際のコード実行パフォーマンスと、ユーザーが感じるパフォーマンスの両方の改善を対象にしています。

最適化の対象決定

パフォーマンスを向上する方法によっては、向上効果がユーザーに認識されない場合があります。重要なのは、当該アプリケーションで実際に問題となる領域について集中的なパフォーマンス最適化を施すことです。パフォーマンスの最適化方法の中にはベストプラクティスとして一般的に通用し、常に使用できるものもありますが、そうでない最適化方法については、アプリケーションの要件と想定されるユーザーベースによって、有益な場合とそうでない場合があります。例えば、アニメーション、ビデオ、またはグラフィックフィルタや効果を使用しなければ、アプリケーションは常に優れたパフォーマンスを発揮できます。しかし、Flash Platform を使用してアプリケーションを構築する理由の 1 つは、表現力が豊かなアプリケーションを作成できるメディアとグラフィックの機能があるからです。アプリケーションに求めるリッチな表現力の程度と、そのアプリケーションを実行するコンピューターおよびデバイスのパフォーマンス特性が合っているかどうかを考えてください。

一般的なアドバイスの 1 つは、「早期に最適化しないこと」です。パフォーマンスの最適化方法によっては、読み取りが困難になり、柔軟性が低下する方法でコードを作成する必要があります。このようなコードでは、一度最適化すると、保守が困難になります。多くの場合、そうした種類のパフォーマンス最適化は早期に実行せず、コード内のどのセクションにパフォーマンスの問題があるかを見極めてから行うほうが有効です。

パフォーマンスの改善には、トレードオフが伴うこともあります。アプリケーションのメモリ消費量を減らすことが、そのままアプリケーションのタスク実行速度向上にもつながれば理想的です。ただし、このような理想的な改善が常に可能とは限りません。例えば、操作中にアプリケーションがフリーズする場合、解決するには、複数のフレームで実行するように作業の分割が必要になることがあります。作業を分割すると、全体的なプロセスの実行時間は長くなりがちです。しかし、アプリケーションが入力に対して応答し続け、フリーズしなければ、時間がかかっていることにユーザーは気が付かない可能性があります。

最適化する内容、および最適化が有効かどうかを把握するには、パフォーマンステストを実行する方法があります。パフォーマンステストのテクニックとヒントについては、88 ページの「ベンチマークおよびデプロイ」でいくつか説明されています。

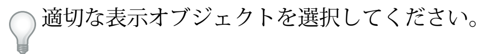
そのアプリケーションにおいて最適化の検討対象とする領域を判断する方法について詳しくは、次のリソースを参照してください。

- 「Performance-tuning apps for AIR」（http://www.adobe.com/go/learn_fp_goldman_tv_jp）（Oliver Goldman による MAX 会議プレゼンテーションのビデオ）
- 「Performance-tuning Adobe AIR applications」（http://www.adobe.com/go/learn_fp_air_perf_devnet_jp）（プレゼンテーションに基づいた、Oliver Goldman による Adobe Developer Connection の記事）

第 2 章：メモリの節約

メモリの節約は、アプリケーション開発において常に重要です。デスクトップ向けのアプリケーションであっても重要なことに変わりありません。モバイルデバイスでは、特にメモリ消費が重視されるので、アプリケーションで消費するメモリ容量を制限することが求められます。

表示オブジェクト


適切な表示オブジェクトを選択してください。

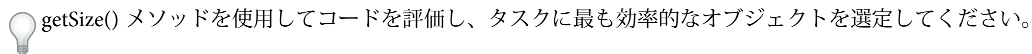
ActionScript 3.0 には多くの表示オブジェクトが含まれています。メモリ使用量を制限する最も簡単な最適化の方法は、適切なタイプの表示オブジェクトを使用することです。非インタラクティブで単純なシェイプには、Shape オブジェクトを使用します。タイムラインを必要としないインタラクティブなオブジェクトには、Sprite オブジェクトを使用します。タイムラインを使用するアニメーションには、MovieClip オブジェクトを使用します。アプリケーションでは、常に最も効率的なオブジェクトタイプを選択してください。

次のコードは、表示オブジェクト別のメモリ使用量を表示します。

```
trace(getSize(new Shape()));
// output: 236

trace(getSize(new Sprite()));
// output: 412

trace(getSize(new MovieClip()));
// output: 440
```

getSize() メソッドは、オブジェクトがメモリ内で消費するバイト数を示します。単純な Shape オブジェクトの代わりに複数の MovieClip オブジェクトを使用すると、MovieClip オブジェクトの機能が必要ない場合に、メモリが浪費されることがわかります。

プリミティブ型


getSize() メソッドを使用してコードを評価し、タスクに最も効率的なオブジェクトを選定してください。

String 型を除くすべてのプリミティブ型では、4 〜 8 バイトのメモリを使用します。プリミティブ型に特定のタイプを使用してメモリを最適化することはできません。

```
// Primitive types
var a:Number;
trace(getSize(a));
// output: 8

var b:int;
trace(getSize(b));
// output: 4

var c:uint;
trace(getSize(c));
// output: 4

var d:Boolean;
trace(getSize(d));
// output: 4

var e:String;
trace(getSize(e));
// output: 4
```

Number 型は 64 ビットの値を表し、値が割り当てられていない場合は、ActionScript 仮想マシン（AVM）から 8 バイトが割り当てられます。その他のプリミティブ型はすべて 4 バイトで格納されます。

```
// Primitive types
var a:Number = 8;
trace(getSize(a));
// output: 4

a = Number.MAX_VALUE;
trace(getSize(a));
// output: 8
```

この動作は、String 型では異なります。割り当てられる記憶域は、ストリングの長さに基づきます。

```
var name:String;
trace(getSize(name));
// output: 4

name = "";
trace(getSize(name));
// output: 24

name = "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the
industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and
scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into
electronic typesetting, remaining essentially unchanged. It was popularized in the 1960s with the release
of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like
Aldus PageMaker including versions of Lorem Ipsum.";
trace(getSize(name));
// output: 1172
```

getSize() メソッドを使用してコードを評価し、タスクに最も効率的なオブジェクトを選定してください。

オブジェクトの再利用

可能な場合は、同じオブジェクトを繰り返し作成する代わりに再利用してください。

メモリを最適化するもう一つの簡単な方法は、できる限りオブジェクトを再利用して、同じオブジェクトを繰り返し作成しないようにすることです。例えば、ループ内で次のコードは使用しないでください。

```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

var area:Rectangle;

for (var:int = 0; i < MAX_NUM; i++)
{
    // Do not use the following code
    area = new Rectangle(i,0,1,10);
    myBitmapData.fillRect(area,COLOR);
}
```

ループでの繰り返しのたびに Rectangle オブジェクトを再作成すると、より多くのメモリが使用され、処理が低速になります。これは、繰り返しごとに新しいオブジェクトが作成されるためです。次の手法を使用します。

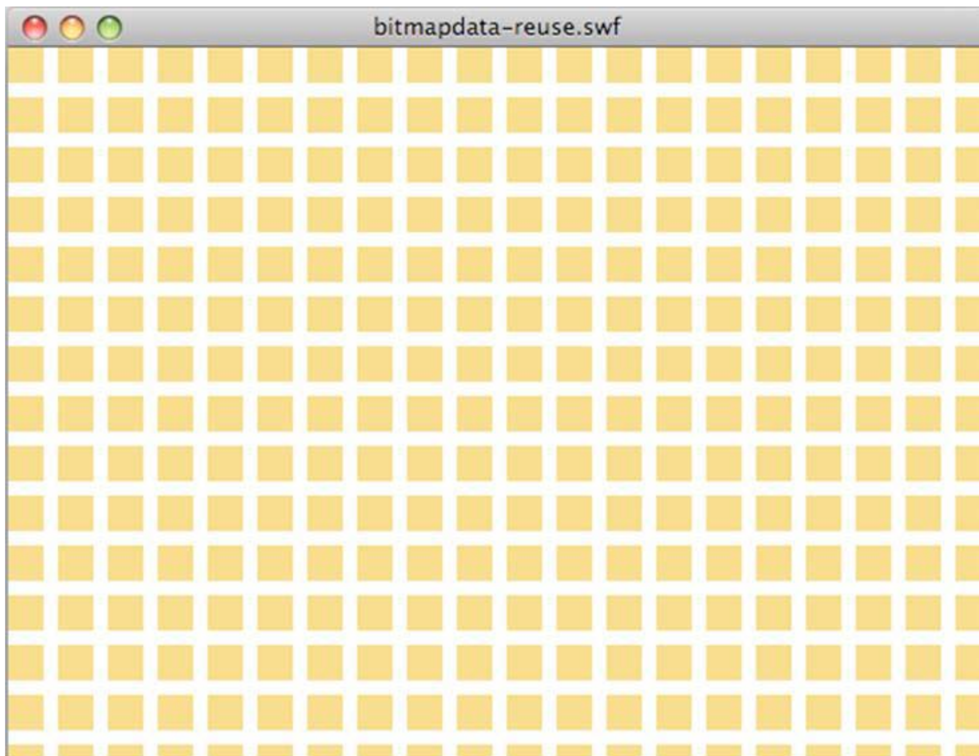
```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

// Create the rectangle outside the loop
var area:Rectangle = new Rectangle(0,0,1,10);

for (var:int = 0; i < MAX_NUM; i++)
{
    area.x = i;
    myBitmapData.fillRect(area,COLOR);
}
```

上述の例では、メモリへの影響が比較的小さいオブジェクトが使用されています。次の例では、BitmapData オブジェクトを再利用することにより、メモリを大幅に節約します。次のコードはタイリング効果を作成し、メモリを浪費します。

```
var myImage:BitmapData;
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a 20 x 20 pixel bitmap, non-transparent
    myImage = new BitmapData(20,20,false,0xF0D062);

    // Create a container for each BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

注意：正の値を使用する場合は、四捨五入された値を整数に型変換する方が Math.floor() メソッドを使用するよりもはるかに高速に処理されます。

次の図は、ビットマップタイリングの結果を示しています。



ビットマップタイリングの結果

最適化されたバージョンでは、BitmapData インスタンスを 1 つ作成して、それを複数の Bitmap インスタンスで参照することにより、同じ結果をもたらします。

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

この手法は、メモリを約 700 KB 節約します。これは、従来のモバイルデバイスにとって、大幅なメモリの節約になります。Bitmap プロパティを使用すれば、元の BitmapData インスタンスを変更せずに、各ビットマップコンテナを操作することができます。

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the DisplayList
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);

    // Set a specific rotation, alpha, and depth
    myContainer.rotation = Math.random()*360;
    myContainer.alpha = Math.random();
    myContainer.scaleX = myContainer.scaleY = Math.random();
}
```

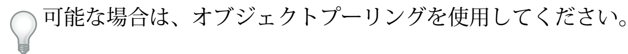
次の図は、ビットマップ変形の結果を示しています。

ビットマップ変形の結果

関連項目

48 ページの「ビットマップキャッシュ」

オブジェクトプーリング


可能な場合は、オブジェクトプーリングを使用してください。

重要な最適化の一つに、オブジェクトプーリングと呼ばれる手法があります。この手法では、長い期間にわたってオブジェクトを再利用します。アプリケーションの初期化時に、限られた数のオブジェクトを作成し、プール内に Array オブジェクトや Vector オブジェクトなどとして格納します。オブジェクトを格納したら、それらを無効化して CPU リソースを消費しないようにします。また、相互参照はすべて削除します。ただし、参照を null に設定しないでください。これを行うと、ガベージコレクションの対象となる可能性があります。オブジェクトをプールに戻し、新しいオブジェクトが必要になったときに、そのオブジェクトを取得します。

オブジェクトを再利用すると、オブジェクトをインスタンス化する必要が減ります。オブジェクトのインスタンス化には費用が掛かります。また、アプリケーションの動作速度を低下させる可能性のあるガベージコレクターの実行回数も削減されます。次のコードは、オブジェクトプーリング手法を示しています。

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

    public final class SpritePool
    {
        private static var MAX_VALUE:uint;
        private static var GROWTH_VALUE:uint;
        private static var counter:uint;
        private static var pool:Vector.<Sprite>;
        private static var currentSprite:Sprite;

        public static function initialize( maxPoolSize:uint, growthValue:uint ):void
        {
            MAX_VALUE = maxPoolSize;
            GROWTH_VALUE = growthValue;
            counter = maxPoolSize;

            var i:uint = maxPoolSize;

            pool = new Vector.<Sprite>(MAX_VALUE);
            while( --i > -1 )
                pool[i] = new Sprite();
        }

        public static function getSprite():Sprite
        {
            if ( counter > 0 )
                return currentSprite = pool[--counter];

            var i:uint = GROWTH_VALUE;
            while( --i > -1 )
                    pool.unshift ( new Sprite() );
            counter = GROWTH_VALUE;
            return getSprite();

        }

        public static function disposeSprite(disposedSprite:Sprite):void
        {
            pool[counter++] = disposedSprite;
        }
    }
}
```

アプリケーションの初期化時に、SpritePool クラスが新しいオブジェクトのプールを作成します。getSprite() メソッドでこれらのオブジェクトのインスタンスを返し、disposeSprite() メソッドでインスタンスを解放します。このコードでは、プールを使い切ったときにプールを拡大することができます。固定サイズのプールを作成することも可能です。この場合は、プールを使い切ると新しいオブジェクトを割り当てられなくなります。可能な場合は、ループ内での新しいオブジェクトの作成は避けてください。詳しくは、10 ページの「メモリの解放」を参照してください。次のコードでは、SpritePool クラスを使用して、新しいインスタンスを取得します。

```
const MAX_SPRITES:uint = 100;
const GROWTH_VALUE:uint = MAX_SPRITES >> 1;
const MAX_NUM:uint = 10;

SpritePool.initialize ( MAX_SPRITES,  GROWTH_VALUE );

var currentSprite:Sprite;
var container:Sprite = SpritePool.getSprite();

addChild ( container );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    for ( var j:int = 0; j< MAX_NUM; j++ )
    {
        currentSprite = SpritePool.getSprite();
        currentSprite.graphics.beginFill ( 0x990000 );
        currentSprite.graphics.drawCircle ( 10, 10, 10 );
        currentSprite.x = j * (currentSprite.width + 5);
        currentSprite.y = i * (currentSprite.width + 5);
        container.addChild ( currentSprite );
    }
}
```

次のコードでは、マウスをクリックしたときに、表示リストからすべての表示オブジェクトが削除されます。削除された表示オブジェクトは後で別のタスクに再利用します。

```
stage.addEventListener ( MouseEvent.CLICK, removeDots );

function removeDots ( e:MouseEvent ):void
{
    while (container.numChildren > 0 )
        SpritePool.disposeSprite (container.removeChildAt(0) as Sprite );
}
```

注意：プール内の Vector オブジェクトは常に Sprite オブジェクトを参照します。オブジェクトをメモリから完全に削除する場合は、SpritePool クラスの dispose() メソッドが必要です。このメソッドは、残っているすべての参照を削除します。

メモリの解放

オブジェクトに対する参照をすべて削除し、必ずガベージコレクションがトリガーされるようにしてください。

リリースバージョンの Flash Player では、ガベージコレクターを直接起動することはできません。オブジェクトが必ずガベージコレクションの対象になるようにするには、オブジェクトに対するすべての参照を削除します。ActionScript 1.0 および 2.0 で使用される古い delete 演算子は、ActionScript 3.0 では動作が異なり、動的オブジェクトの動的プロパティを削除するためだけに使用できます。

注意：ガベージコレクターは、Adobe® AIR® およびデバッグバージョンの Flash Player で直接呼び出すことができます。

例えば、次のコードは、スプライト参照を null に設定します。

```
var mySprite:Sprite = new Sprite();

// Set the reference to null, so that the garbage collector removes
// it from memory
mySprite = null;
```

オブジェクトが null に設定されている場合は、必ずしもメモリから削除する必要はありません。使用可能なメモリ量がそれほど小さくないと見なされる場合は、ガベージコレクターが実行されないことがあります。ガベージコレクションの実行は事前に予測できません。オブジェクトの削除ではなく、メモリの割り当てによって、ガベージコレクションがトリガーされます。ガベージコレクターは、実行時に、収集されていないオブジェクトのグラフを検出します。グラフ内の無効なオブジェクトを検出するには、相互に参照しているオブジェクトのうち、アプリケーションで使用されなくなったオブジェクトを探します。この方法で検出された無効なオブジェクトが削除されます。

大規模なアプリケーションでは、この方法は CPU を消費して、パフォーマンスに影響を与えるので、アプリケーションの処理速度が著しく低下します。オブジェクトをできる限り再利用して、ガベージコレクションの実行を抑制します。また、可能な場合は参照を null に設定して、ガベージコレクターがオブジェクトの検出にかける時間を短縮します。ガベージコレクションを念のための備えと考え、可能な場合は、常にオブジェクトの存続期間を明示的に管理します。

注意：表示オブジェクトへの参照を null に設定しても、オブジェクトがフリーズされるとは限りません。オブジェクトはガベージコレクションの対象となるまでは引き続き CPU サイクルを消費します。オブジェクトを適切に非アクティブ化してから、参照を null に設定するようにしてください。

ガベージコレクターは System.gc() メソッドで起動できます。このメソッドは Adobe AIR およびデバッグバージョンの Flash Player で使用できます。Adobe® Flash® Builder にバンドルされているプロファイラーを使用すると、ガベージコレクターを手動で開始できます。ガベージコレクターを実行すると、アプリケーションの応答内容およびオブジェクトがメモリから正常に削除されたかどうかを確認できます。

注意：オブジェクトがイベントリスナーとして使用されていた場合は、別のオブジェクトがそれを参照している可能性があります。その場合は、removeEventListener() メソッドを使用してイベントリスナーを削除してから、null への参照を設定します。

ビットマップによって使用されるメモリの量は瞬時に減らすことができます。例えば、BitmapData クラスには、dispose() メソッドが用意されています。次の例では、1.8 MB の BitmapData インスタンスを作成します。現在のメモリ使用量が 1.8 MB 増加し、System.totalMemory プロパティから返される値が小さくなります。

```
trace(System.totalMemory / 1024);
// output: 43100

// Create a BitmapData instance
var image:BitmapData = new BitmapData(800, 600);

trace(System.totalMemory / 1024);
// output: 44964
```

次に、BitmapData をメモリから手動で削除（廃棄）し、メモリ使用量を再度チェックします。

```
trace(System.totalMemory / 1024);
// output: 43100

// Create a BitmapData instance
var image:BitmapData = new BitmapData(800, 600);

trace(System.totalMemory / 1024);
// output: 44964

image.dispose();
image = null;

trace(System.totalMemory / 1024);
// output: 43084
```

dispose() メソッドはメモリからピクセルを削除しますが、メモリを完全に解放するには、さらに参照を null に設定する必要があります。BitmapData オブジェクトが必要なくなったときは、常に dispose() メソッドを呼び出し、さらに参照を null に設定します。これにより、メモリが直ちに解放されます。

注意：Flash Player 10.1 および AIR 1.5.2 では、System クラスに disposeXML() と呼ばれる新しいメソッドが導入されています。このメソッドでは、XML ツリーをパラメーターとして渡すことにより、ガベージコレクションにすぐに XML オブジェクトを使用できるようになります。

関連項目

25 ページの「オブジェクトのフリーズとフリーズ解除」

ビットマップの使用

ビットマップの代わりにベクターを使用することは、メモリを節約するために有効な方法です。しかし、ベクターを使用すると、特にその数が多い場合は、必要な CPU または GPU リソースが大幅に増加します。ランタイムが画面上でピクセルを描画するのに必要な処理リソースは、ベクターコンテンツをレンダリングする場合よりも少なくて済むので、ビットマップの使用はレンダリングの最適化に適しています。

関連項目

55 ページの「手動によるビットマップキャッシュ」

ビットマップのダウンサンプリング

メモリの使用を改善するため、Flash Player で 16 ビット画面を検出すると、32 ビットの不透明イメージは 16 ビットのイメージに縮小されます。このダウンサンプリングによって、メモリリソースの消費量が半減し、イメージのレンダリングが高速化します。この機能は、Windows Mobile 用 Flash Player 10.1 でのみ使用できます。

注意：Flash Player 10.1 より前では、メモリ内に作成されたすべてのピクセルが 32 ビット（4 バイト）で格納されていました。300 x 300 ピクセルのシンプルなロゴは、350 KB（300 * 300 * 4 / 1,024）のメモリを消費していました。この新しい機能を使用すれば、同じ不透明なロゴによって消費されるメモリは 175 KB のみです。ロゴが透明な場合は、16 ビットにダウンサンプリングされないので、メモリのサイズは変わりません。この機能が適用されるのは、埋め込みビットマップまたはランタイムによりロードされた画像（PNG、GIF、JPG）のみです。

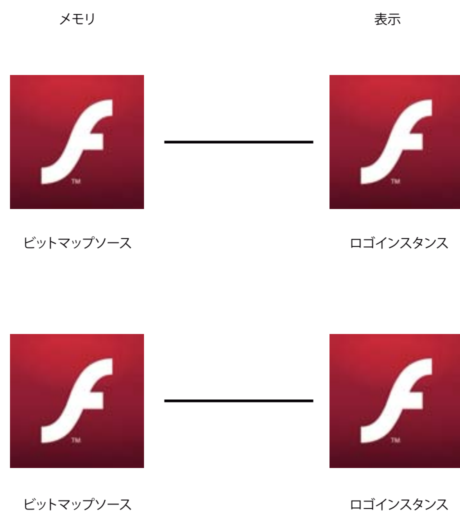
モバイルデバイスでは、同じイメージを 16 ビットと 32 ビットでレンダリングした場合の違いを見分けるのは困難です。2、3 色からなるシンプルなイメージでは、目で見てわかるような違いはありません。より複雑なイメージでも、その違いを見分けるのは困難です。ただし、イメージを拡大すると、一部に不均一な色のグラデーションが見られる場合があります。また、16 ビットのグラデーションの方が 32 ビットよりも滑らかさに劣るように見えることがあります。

BitmapData の単一参照

可能な限りインスタンスを再利用して、BitmapData クラスの使用を最適化することが重要です。Flash Player 10.1 および AIR 2.5 には、すべてのプラットフォームを対象に、BitmapData 単一参照と呼ばれる新機能が導入されています。埋め込み画像から BitmapData インスタンスを作成するときは、ビットマップの単一バージョンがすべての BitmapData インスタンスに使用されます。後でビットマップを変更すると、メモリ内に固有のビットマップが作成されます。埋め込み画像は、ライブラリから使用するか、[Embed] タグとすることができます。

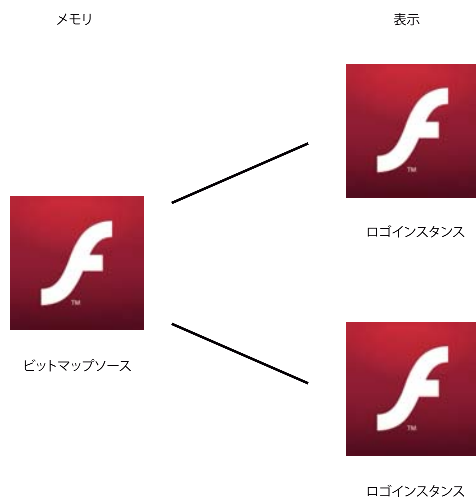
注意：Flash Player 10.1 および AIR 2.5 は、自動的にビットマップを再利用するので、既存のコンテンツもこの新機能を活用できます。

埋め込み画像をインスタンス化するときは、関連するビットマップがメモリ内に作成されます。Flash Player 10.1 および AIR 2.5 より前では、次の図に示すように、メモリ内でインスタンスごとに別々のビットマップが割り当てられていました。



Flash Player 10.1 および AIR 2.5 より前のメモリ内のビットマップ

Flash Player 10.1 および AIR 2.5 では、同じイメージのインスタンスが複数作成されると、すべての BitmapData インスタンスに対して、単一のビットマップが使用されます。次の図に、この概念を示します。

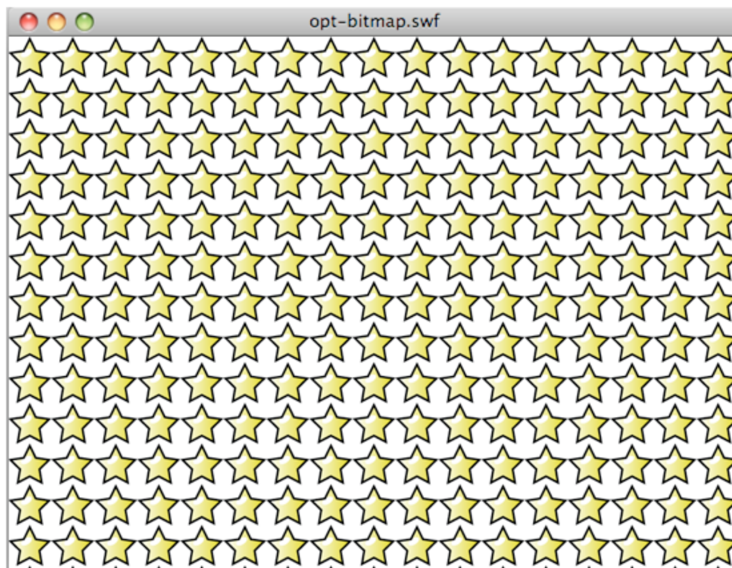


メモリ内のビットマップ（Flash Player 10.1 および AIR 2.5）

この手法は、ビットマップが多数含まれているアプリケーションのメモリ使用量を大幅に低減します。次のコードは、Star シンボルのインスタンスを複数作成します。

```
const MAX_NUM:int = 18;

var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}
```

次の図は、コードの結果を示しています。



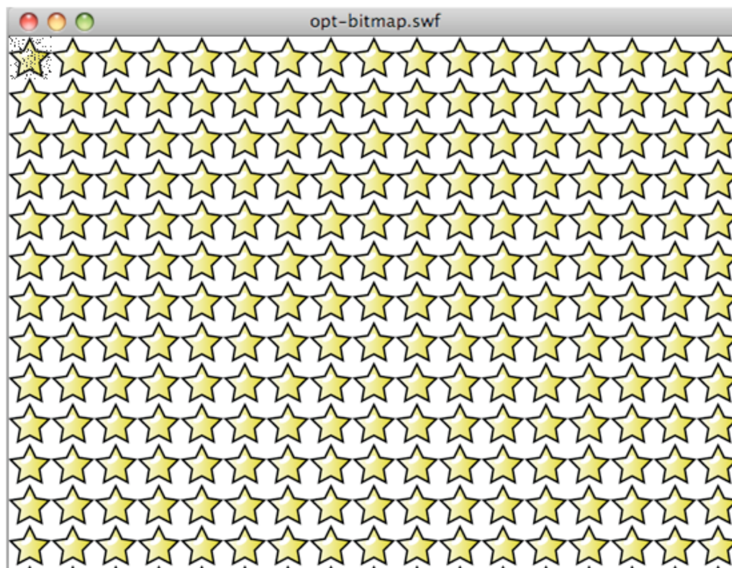
シンボルのインスタンスを複数作成するコードの結果

例えば、Flash Player 10 で上記のアニメーションを実行すると、約 1008 KB のメモリが使用されます。Flash Player 10.1 では、同じアニメーションをデスクトップとモバイルデバイスのどちらで実行した場合でも、わずか 4 KB のメモリしか使用されません。

次のコードでは、1 つの BitmapData インスタンスを変更します。

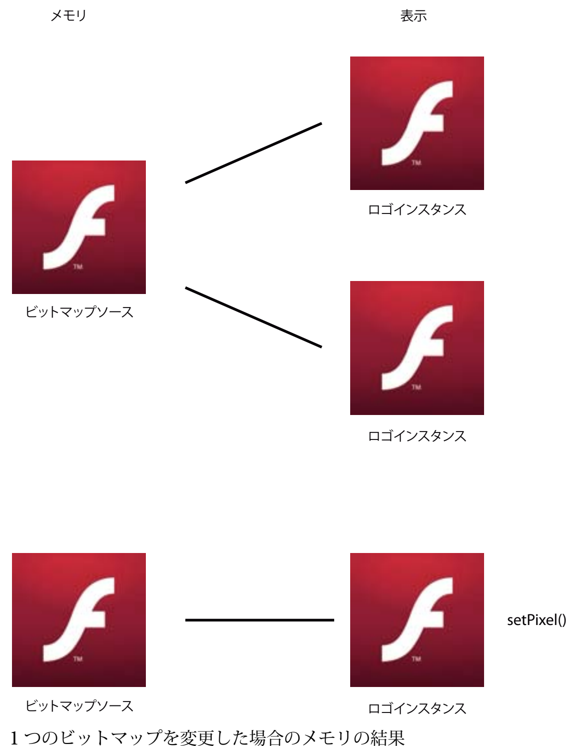
```
const MAX_NUM:int = 18;

var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}

var ref:Bitmap = getChildAt(0) as Bitmap;

ref.bitmapData.pixelDissolve(ref.bitmapData, ref.bitmapData.rect, new
Point(0,0),Math.random()*200,Math.random()*200, 0x990000);
```

次の図は、1 つの Star インスタンスを変更した結果を示しています。



1 つのインスタンスを変更した結果

内部的には、ランタイムがメモリ内でビットマップの割り当てと作成を自動的に行い、ピクセルの変更を処理します。BitmapData クラスのメソッドが呼び出され、ピクセルの変更につながると、メモリ内に新しいインスタンスが作成されます。それ以外のインスタンスは更新されません。次の図に、この概念を示します。



1 つのビットマップを変更した場合のメモリの結果

星を 1 つ変更すると、新しいコピーがメモリ内に作成されます。生成されるアニメーションは、Flash Player 10.1 および AIR 2.5 で約 8 KB のメモリを使用します。

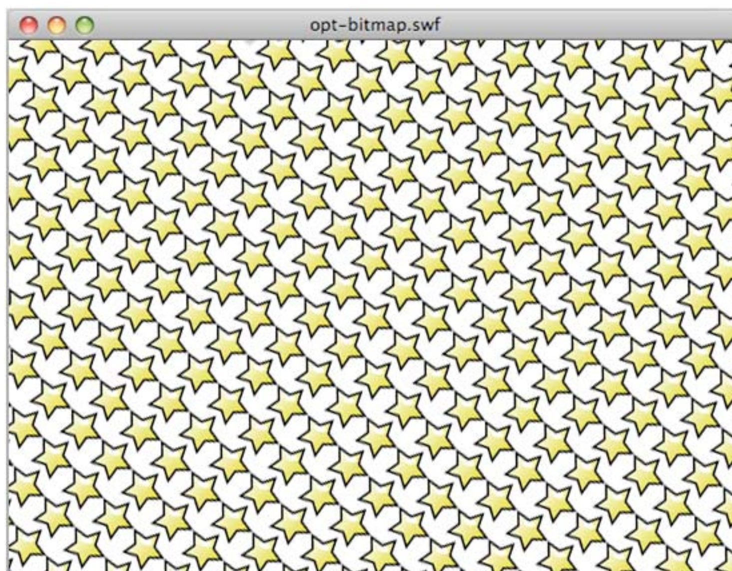
上述の例では、各ビットマップは個別に変換に利用できるようになります。タイル効果のみを作成するには、beginBitmapFill() メソッドが最適なメソッドです。

```
var container:Sprite = new Sprite();

var source:BitmapData = new Star(0,0);

// Fill the surface with the source BitmapData
container.graphics.beginBitmapFill(source);
container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);

addChild(container);
```

この手法は、BitmapData インスタンスを 1 つ作成するだけで、同じ結果をもたらします。各 Star インスタンスにアクセスするのではなく、連続して星を回転させるには、各フレームで回転する Matrix オブジェクトを使用します。Matrix オブジェクトを beginBitmapFill() メソッドに渡します。

```
var container:Sprite = new Sprite();

container.addEventListener(Event.ENTER_FRAME, rotate);

var source:BitmapData = new Star(0,0);
var matrix:Matrix = new Matrix();

addChild(container);

var angle:Number = .01;

function rotate(e:Event):void
{
    // Rotate the stars
    matrix.rotate(angle);

    // Clear the content
    container.graphics.clear();

    // Fill the surface with the source BitmapData
    container.graphics.beginBitmapFill(source,matrix,true,true);
    container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);
}
```

このテクニックを使用すると、効果を作成するために ActionScript ループは不要になります。ランタイムにより、すべてが内部的に実行されます。次の図は、星の変形の結果を示しています。



星の回転の結果

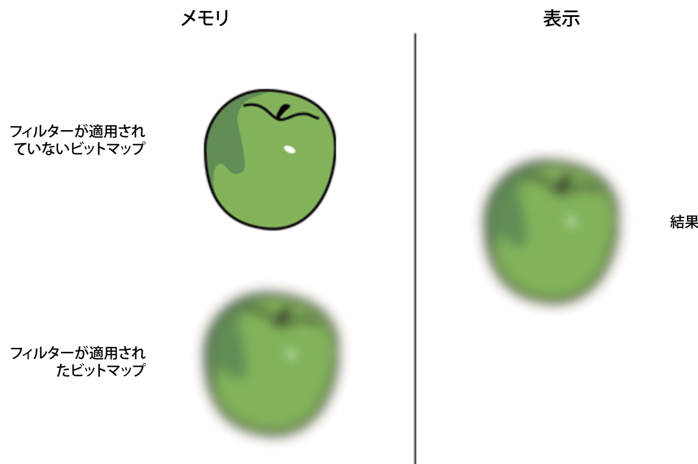
この手法では、BitmapData の元のソースオブジェクトを更新すると、ステージ上でこのオブジェクトを使用した別の表示が自動的に更新されるので、強力なテクニックになり得ます。この手法では、上述の例のように、それぞれの星を個別に拡大 / 縮小することはできません。

注意：同じイメージのインスタンスを複数使用する場合は、メモリ内の元のビットマップにクラスが関連付けられているかどうかによって、描画方法は異なります。ビットマップに関連付けられているクラスがない場合、イメージは Shape オブジェクトとして描画され、ビットマップで塗りつぶされます。

フィルターおよびビットマップの動的アンロード

Pixel Bender で処理したフィルターも含め、フィルターは使用しないようにしてください。

フィルター（Pixel Bender を使用してモバイルデバイスで処理したフィルターを含む）などの効果の使用は最小限に抑えてください。表示オブジェクトにフィルターが適用されている場合、メモリ内にビットマップが 2 つ作成されます。これらのビットマップは、どちらも表示オブジェクトと同じサイズです。1 番目のビットマップは表示オブジェクトのラスタライズバージョンとして作成されます。このラスタライズバージョンを使用して 2 番目のビットマップを生成し、それにフィルターが適用されます。



フィルター適用時にメモリ内に作成された 2 つのビットマップ

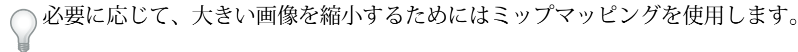
フィルターのいずれかのプロパティを変更すると、両方のビットマップがメモリ内で更新され、結果のビットマップが作成されます。このプロセスは CPU 処理を伴い、2 つのビットマップがメモリを大量に使用する可能性があります。

Flash Player 10.1 および AIR 2.5 では、すべてのプラットフォームで新しいフィルター動作が導入されています。フィルターが 30 秒以内に変更されない、または非表示かオフスクリーンの場合、フィルターが適用されていない方のビットマップが使用しているメモリは解放されます。

この機能により、すべてのプラットフォームで、フィルターが使用するメモリが半減します。例えば、ぼかしフィルターが適用されたテキストオブジェクトについて考えてみましょう。この場合、テキストを簡単な装飾に使用し、変更は行いません。30 秒後、メモリ内の、フィルターが適用されていないビットマップが解放されます。テキストを 30 秒間非表示にするか、オフスクリーンにした場合も、同じ結果が示されます。フィルターのいずれかのプロパティが変更された場合は、メモリ内の、フィルターが適用されていないビットマップが再作成されます。この機能は、ビットマップの動的アンロードと呼ばれています。これらの最適化を使用しても、フィルターの扱いには注意してください。フィルターの変更時に、膨大な CPU または GPU 処理が必要とされることに変わりありません。

ベストプラクティスとして、可能な場合は Adobe® Photoshop® などのオーサリングツールで作成されたビットマップを使用して、フィルターをエミュレートします。ActionScript で実行時に作成された動的ビットマップを使用することは避けます。外部的に作成されたビットマップを使用すると、特にフィルタープロパティが長時間変更されない場合に、ランタイムでは CPU または GPU のロードを低減できます。可能な場合は、オーサリングツールでビットマップに必要な効果を作成します。これにより、ランタイムで何も処理を実行せずにビットマップを表示できるようになり、格段に高速化されます。

ダイレクトミップマッピング


必要に応じて、大きい画像を縮小するためにはミップマッピングを使用します。

すべてのプラットフォームで使用可能な Flash Player 10.1 および AIR 2.5 の新機能の 1 つに､ミップマッピングに関する機能があります。Flash Player 9 および AIR 1.0 で導入されたミップマッピング機能によって、縮小されたビットマップの画質とパフォーマンスが向上しました。

注意：ミップマッピング機能が適用されるのは、動的にロードされた画像または埋め込みビットマップのみです。フィルターが適用された表示オブジェクトまたはキャッシュされた表示オブジェクトは対象外です。ミップマッピングは、ビットマップの幅と高さが偶数の場合にのみ処理できます。幅または高さが奇数になると、ミップマッピングは終了します。例えば、250 x 250 のイメージは 125 x 125 にミップマップできますが、それ以上は処理できません。この場合は、1 つ以上の寸法が奇数です。ビットマップの寸法が 2 のべき乗（例えば、256 x 256、512 x 512、1,024 x 1,024 など）の場合に、最善の結果を得ることができます。

例えば、1,024 x 1,024 のイメージがロードされているとします。開発者はこのイメージを縮小して、ギャラリー用のサムネイルを作成しようとしています。ミップマッピング機能では、ダウンサンプリングしたビットマップの中間バージョンをテクスチャとして使用することにより、縮小されたイメージを適切にレンダリングします。以前のバージョンのランタイムでは、縮小されたビットマップの中間バージョンがメモリ内に作成されていました。1,024 x 1,024 のイメージをロードして 64 x 64 で表示する場合、以前のバージョンのランタイムでは、すべてのビットマップについて、2 分の 1 に縮小したものを作成していました。例えば、この場合は、512 x 512、256 x 256、128 x 128、64 x 64 の各ビットマップが作成されます。

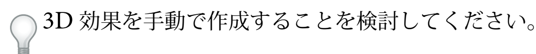
Flash Player 10.1 および AIR 2.5 では、元のソースから目的のサイズに直接ミップマップする機能をサポートするようになりました。上述の例では、4 MB（1,024 x 1,024）の元のビットマップと、16 KB（64 x 64）のミップマップされたビットマップのみが作成されます。

また、ミップマッピングのロジックは、ビットマップの動的アンロード機能とも連携します。64 x 64 のビットマップしか使用していない場合は、4 MB の元のビットマップはメモリから解放されます。ミップマップの再作成が必要な場合は、元のビットマップが再ロードされます。また、ミップマップされたビットマップについて、様々なサイズが必要な場合は、ビットマップのミップマップチェーンを使用してビットマップを作成します。例えば、8 分の 1 のビットマップを作成する必要がある場合、まず 4 分の 1、2 分の 1、1 分の 1 のビットマップを調べて、メモリにロードするビットマップを選定します。他のサイズが見つからない場合は、リソースから元のビットマップ（1 分の 1）をロードして使用します。

JPEG 解凍は独自の形式内でミップマッピングを実行できます。このダイレクトミップマッピングを使用すると、解凍済みの完全なイメージをロードすることなく、サイズの大きいビットマップをミップマップ形式に直接解凍できます。これにより、ミップマップの生成が大幅に高速化し、サイズの大きいビットマップに使用されていたメモリは、割り当てられずに解放されます。JPEG イメージの画質は、一般的なミップマッピング手法の画質と比べても遜色ありません。

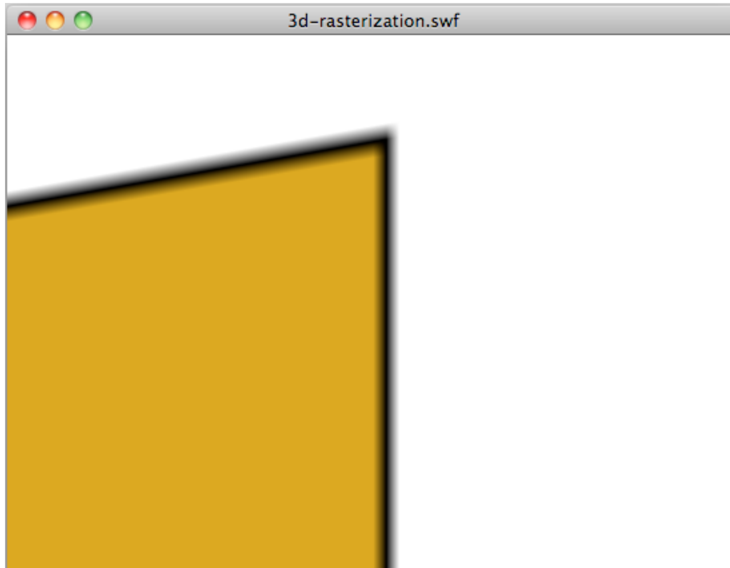
注意：ミップマッピングは慎重に使用します。これによって、縮小したビットマップの画質は向上しますが、帯域幅、メモリ、速度に影響があります。場合によっては、外部ツールからビットマップのあらかじめ縮小されたバージョンを使用し、それをアプリケーションに読み込むオプションが有効です。縮小することだけが目的の場合は、大きいビットマップで作業を開始しないでください。

3D 効果の使用


3D 効果を手動で作成することを検討してください。

Flash Player 10 および AIR 1.5 で導入された 3D エンジンを使用すると、表示オブジェクトに遠近法による変形を適用できます。この変形を適用するには、rotationX プロパティおよび rotationY プロパティを使用するか、Graphics クラスの drawTriangles() メソッドを使用します。また、z プロパティで深度を適用することもできます。遠近法による変形が適用された表示オブジェクトは、ビットマップとしてラスタライズされるので、多くのメモリを必要とすることに注意してください。

次の図は、遠近法による変形の使用時に、ラスタライズによって適用されたアンチエイリアスを示しています。



遠近法による変形に起因するアンチエイリアス

ベクターコンテンツをビットマップとして動的にラスタライズすると、アンチエイリアスが発生します。アンチエイリアスが発生するのは、3D 効果を、デスクトップ用の AIR および Flash Player と、モバイル用の AIR 2.0.1 および AIR 2.5 で使用する場合のみです。ただし、アンチエイリアスはモバイルデバイス用の Flash Player には適用されません。

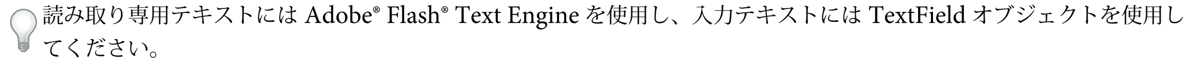
ネイティブ API に依存せずに 3D 効果を手動で作成できれば、メモリ使用量を減らすことができます。Flash Player 10 および AIR 1.5 で導入された 3D の新機能を使用すれば、テクスチャマッピングがさらに簡単になります。これは、drawTriangles() などのメソッドを使用して、テクスチャマッピングをネイティブに処理できるからです。

開発者は、3D 効果を作成するとき、パフォーマンスを向上させるためには、ネイティブ API または手動のどちらで処理すべきかを判断してください。ActionScript の実行パフォーマンス、レンダリングパフォーマンスおよびメモリ使用量について検討する必要があります。

renderMode アプリケーションプロパティを GPU に設定している AIR 2.0.1 および AIR 2.5 のモバイルアプリケーションでは、GPU が 3D 変形を実行します。ただし、renderMode が CPU に設定されている場合には、GPU ではなく CPU が 3D 変形を実行します。Flash Player 10.1 アプリケーションでは、CPU が 3D 変形を実行します。

CPU で 3D 変形を実行する場合、表示オブジェクトにいずれかの 3D 変形を適用すると、メモリ内にビットマップが 2 つ必要になることを考慮してください。1 つはソースビットマップ用、もう 1 つは遠近法に基づく変形後のバージョン用です。このようにして、3D 変形はフィルターに対しても同様に動作します。そのため、CPU で 3D 変形を実行する場合は、3D プロパティを多用しないようにしてください。

テキストオブジェクトとメモリ


読み取り専用テキストには Adobe® Flash® Text Engine を使用し、入力テキストには TextField オブジェクトを使用してください。

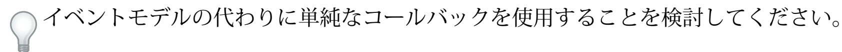
Flash Player 10 および AIR 1.5 で導入された新しい強力なテキストエンジン、Adobe Flash Text Engine（FTE）は、システムメモリを節約します。ただし、FTE は低レベル API なので、ActionScript 3.0 レイヤーを上位に追加する必要があります。FTE は flash.text.engine パッケージで提供されます。

読み取り専用テキストには、Flash Text Engine の使用が最も適しています。これを使用すると、メモリ使用量が低減し、レンダリング品質が向上します。入力テキストには、TextField オブジェクトが適しています。ActionScript コードをあまり使用せずに、一般的な動作（入力処理や禁則処理など）を作成できます。

関連項目

61 ページの「テキストオブジェクトのレンダリング」

イベントモデルとコールバック


イベントモデルの代わりに単純なコールバックを使用することを検討してください。

ActionScript 3.0 イベントモデルは、オブジェクト送出の概念に基づいています。イベントモデルはオブジェクト指向であり、コードの再利用に対して最適化されています。dispatchEvent() メソッドはリスナーのリストをループ処理し、登録された各オブジェクトでイベントハンドラーメソッドを呼び出します。ただし、イベントモデルの欠点の 1 つとして、アプリケーションの有効期間にわたって多くのオブジェクトを作成する可能性が高いことが挙げられます。

アニメーションシーケンスの終了を示すために、タイムラインからイベントを送出する必要があることを考えてみてください。通知を完了するには、次のコードに示すように、タイムラインの特定のフレームからイベントを送出することができます。

```
dispatchEvent( new Event ( Event.COMPLETE ) );
```

Document クラスは、次のコードを使用してこのイベントを監視できます。

```
addEventListener( Event.COMPLETE, onAnimationComplete );
```

この手法は正しいのですが、ネイティブイベントモデルの使用は低速となり、従来のコールバック関数を使用するよりも多くのメモリを消費する可能性があります。イベントオブジェクトを作成してメモリ内に割り当てる必要がありますが、これによりパフォーマンスが低下します。例えば、Event.ENTER_FRAME イベントを監視する場合、新しいイベントオブジェクトが、イベントハンドラーの各フレームで作成されます。キャプチャフェーズおよびバブリングフェーズが原因で、表示オブジェクトに対するパフォーマンスが特に低速になる可能性があります。表示リストが複雑である場合、これには特に費用が掛かります。

第 3 章：CPU 使用率の最小化

最適化の重要な領域の一つに、CPU 使用率があります。CPU 処理を最適化すると、パフォーマンスを改善し、モバイルデバイスのバッテリー寿命を延ばすことができます。

Flash Player 10.1 における CPU 使用率の強化機能

Flash Player 10.1 には、CPU 処理の軽減に役立つ 2 つの新機能が導入されています。SWF コンテンツが画面外に移動したときに再生を一時停止および再開する機能と、1 ページあたりの Flash Player のインスタンス数を制限する機能です。

一時停止、スロットル、再開

注意：一時停止、スロットルおよび再開機能は、Adobe® AIR® アプリケーションには適用されません。

CPU 使用率とバッテリー使用率を最適化するため、Flash Player 10.1 には非アクティブなインスタンスに関する新機能が導入されています。この機能を使用すると、コンテンツが画面に表示されたり表示されなくなったときに SWF ファイルを一時停止および再開することで、CPU 使用率を制限できます。Flash Player は、この機能を使用して、コンテンツ再生を再開したときに再作成すればよいオブジェクトをすべて削除することにより、できる限り多くのメモリを解放します。コンテンツ全体が画面外にある場合、コンテンツはオフスクリーンと見なされます。

SWF コンテンツがオフスクリーンになるシナリオは 2 つあります。

- ユーザーがページをスクロールして、SWF コンテンツを画面外に移動させた場合。

  オーディオまたはビデオ再生があるときは、コンテンツの再生は続けられますが、レンダリングは停止します。再生中のオーディオまたはビデオがない場合に、再生または ActionScript の実行が一時停止されないようにするには、hasPriority HTML パラメーターを true に設定します。ただし、SWF コンテンツがオフスクリーンまたは非表示の場合は、hasPriority HTML パラメーターの値に関わらず、コンテンツのレンダリングが一時停止することに注意してください。

- ブラウザーのタブが開かれ、SWF コンテンツがバックグラウンドに移動した場合。

  hasPriority HTML タグの値に関わらず、SWF コンテンツの速度が 2 ～ 8 fps に下げられます（つまり、「スロットル」されます）。SWF コンテンツが再度表示されるまで、オーディオとビデオの再生は停止し、コンテンツのレンダリングは処理されません。

Flash Player 11.2 以降で、Windows および Mac のデスクトップブラウザーで実行されている場合は、アプリケーションで ThrottleEvent を使用できます。ThrottleEvent は、Flash Player が再生を一時停止、スロットルまたは再開したときに送出されます。

ThrottleEvent はブロードキャストイベントです。したがって、このイベントに対して登録されているリスナーを持つすべての EventDispatcher オブジェクトがこのイベントを送出します。ブロードキャストイベントについて詳しくは、「DisplayObject」クラスを参照してください。

インスタンスの管理

注意：インスタンスの管理機能は、Adobe® AIR® アプリケーションには適用されません。

オフスクリーンの SWF ファイルのロードを遅らせるには、hasPriority HTML パラメーターを使用してください。

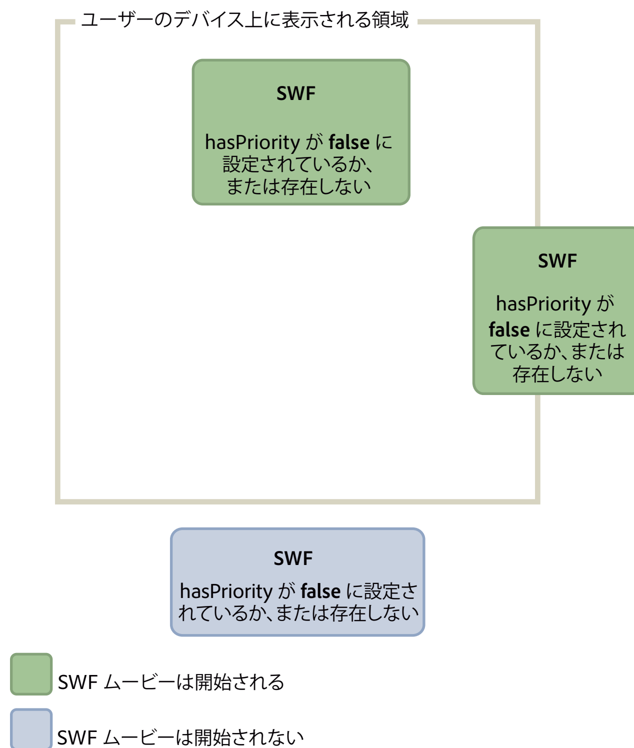
Flash Player 10.1 には、hasPriority という新しい HTML パラメーターが導入されています。

```
<param name="hasPriority" value="true" />
```

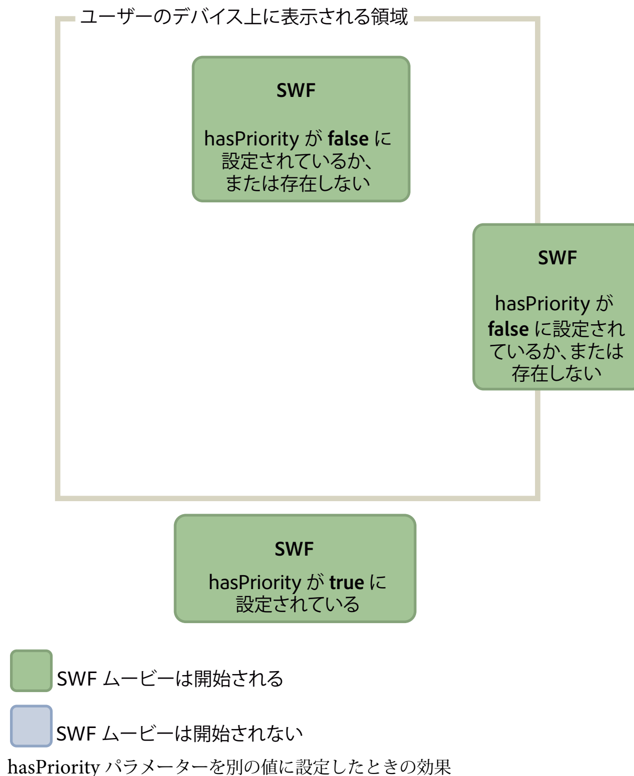
この機能を使用すると、ページ上で開始される Flash Player のインスタンス数を制限できます。インスタンス数を制限すると、CPU およびバッテリーリソースを節約することができます。この意図は、SWF コンテンツに特定の優先度を割り当て、ページ上にある一部のコンテンツを他のコンテンツよりも優先させることにあります。次の簡単な例について考えてみましょう。例えば、ユーザーが閲覧中の Web サイトで、インデックスページに 3 つの異なる SWF ファイルがホストされているとします。そのうちの 1 つは表示され、もう 1 つは部分的に表示されています。最後のファイルは画面外にあり、表示するにはスクロールする必要があります。最初の 2 つのアニメーションは正常に開始されますが、最後のアニメーションは表示されるまで開始が保留されます。このシナリオは、hasPriority パラメーターが存在しないか、false に設定されている場合のデフォルトの動作です。SWF ファイルが画面外にあっても開始されるようにするには、hasPriority パラメーターを true に設定します。ただし、hasPriority パラメーターの値に関わらず、ユーザーには表示されない SWF ファイルでは、常にレンダリングが一時停止されます。

注意：使用可能な CPU リソースが減少すると、hasPriority パラメーターが true に設定されていても、Flash Player のインスタンスは自動的に開始されません。ページのロード後、JavaScript 経由で新しいインスタンスが作成されると、その新しいインスタンスは hasPriority フラグを無視します。Web 管理者が hasPriority フラグを含めることに失敗した場合、任意の 1 x 1 ピクセルまたは 0 x 0 ピクセルのコンテンツが開始され、SWF ヘルパー ファイルを保留できなくなります。ただし、SWF ファイルはクリックすると開始できます。この動作を、「クリックして再生」と呼びます。

次の図は、hasPriority パラメーターを別の値に設定したときの効果を示しています。



hasPriority パラメーターを別の値に設定したときの効果



hasPriority パラメーターを別の値に設定したときの効果

スリープモード

Flash Player 10.1 および AIR 2.5 には、モバイルデバイス向けの新機能が導入されています。この機能を使用すれば、CPU 処理を軽減してバッテリ寿命を延ばすことができます。この機能では、多くのモバイルデバイスに実装されているバックライトの動作を利用します。例えば、モバイルアプリケーションを実行中のユーザーが、何らかの理由でデバイスの使用を中止すると、ランタイムはバックライトのスリープモードへの移行を検出します。スリープモードを検出すると、フレームレートを 4 フレーム / 秒（fps）に下げ、レンダリングを一時停止します。AIR アプリケーションの場合、アプリケーションがバックグラウンドに移動したときにもスリープモードが開始されます。

ActionScript コードはスリープモードで引き続き実行されます。つまり、Stage.frameRate プロパティを 4 fps に設定したときと同様の動作です。ただし、レンダリング処理は行われないので、ユーザーはプレーヤーが 4 fps で実行中であることに気づきません。フレームレートを 0（ゼロ）ではなく 4 fps に設定するのは、すべての接続（NetStream、Socket および NetConnection）で開いた状態を維持できるからです。0（ゼロ）に切り替えると、開いている接続が切断されます。更新頻度に 250 ミリ秒（4 fps）が選択されている理由は、多くのデバイスメーカーがこのフレームレートを更新頻度として使用しているからです。この値を使用することにより、ランタイムとデバイスのフレームレートを同じ範囲内に収めることができます。

注意 : ランタイムがスリープモードのときは、Stage.frameRate プロパティは 4 fps ではなく、オリジナルの .SWF ファイルのフレームレートを返します。

バックライトがオンモードに戻ると、レンダリングが再開されます。フレームレートは元の値に戻ります。ユーザーがメディアプレーヤーアプリケーションで音楽を再生している場合について考えてみましょう。画面がスリープモードに移行すると、ランタイムは再生中のコンテンツの種類に基づいて応答します。状況に応じたランタイムの動作を以下に示します。

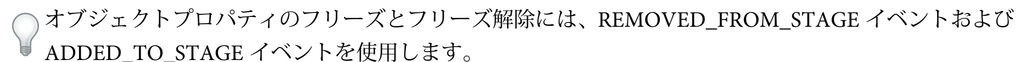
- バックライトがスリープモードに移行し、非 A/V コンテンツが再生中の場合：レンダリングは一時停止され、フレームレートが 4 fps に設定されます。

- バックライトがスリープモードに移行し、A/V コンテンツが再生中の場合：ランタイムはバックライトを強制的に常時オンモードにし、ユーザーエクスペリエンスを維持します。
- バックライトがスリープモードからオンモードに移行した場合：ランタイムはフレームレートを元の SWF ファイルのフレームレート設定に合わせ、レンダリングを再開します。
- A/V コンテンツの再生中に Flash Player を一時停止した場合：Flash Player は、A/V コンテンツが再生されていないので、バックライトの状態をデフォルトのシステム動作にリセットします。
- A/V コンテンツの生成中にモバイルデバイスが電話の呼び出しを着信した場合：レンダリングは一時停止され、フレームレートが 4 fps に設定されます。
- モバイルデバイスでバックライトのスリープモードを無効にしている場合：ランタイムは通常の動作をします。

バックライトがスリープモードに移行すると、レンダリングが一時停止され、フレームレートが低速に設定されます。この機能は CPU 処理を軽減しますが、ゲームアプリケーションなどでは一時停止が実際に正しく機能しない場合もあります。

注意：ランタイムがスリープモードに移行するか、スリープモードを終了するときは、ActionScript イベントは送出されません。

オブジェクトのフリーズとフリーズ解除

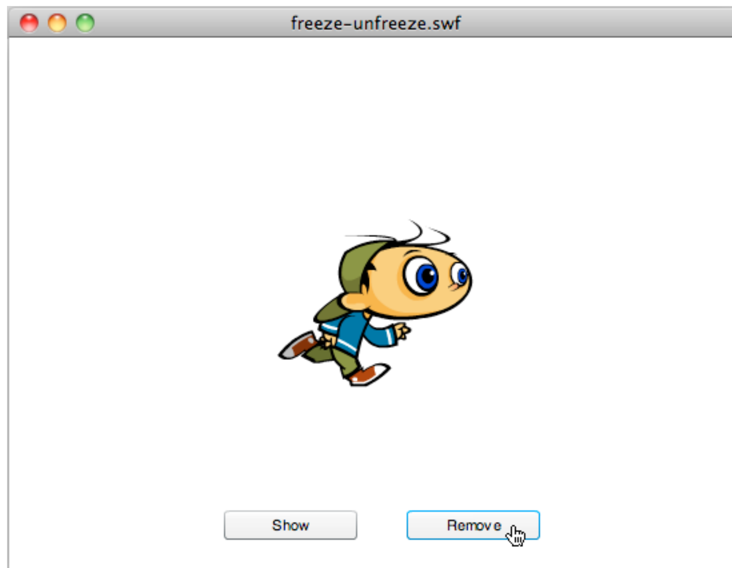


オブジェクトプロパティのフリーズとフリーズ解除には、REMOVED_FROM_STAGE イベントおよび ADDED_TO_STAGE イベントを使用します。

コードを最適化するには、オブジェクトのフリーズとフリーズ解除を常に実行します。フリーズとフリーズ解除はすべてのオブジェクトにとって重要ですが、特に表示オブジェクトにとって重要です。表示オブジェクトは、既に表示リストになく、ガベージコレクションで収集されるのを待機している状態でも、CPU 負荷の高いコードを使用している可能性があります。例えば、Event.ENTER_FRAME を引き続き使用している場合などです。その結果、Event.REMOVED_FROM_STAGE イベントおよび Event.ADDED_TO_STAGE イベントを使用してオブジェクトを正しくフリーズおよびフリーズ解除することが重要です。次の例は、キーボードの操作によってステージ上で再生されるムービークリップを示しています。

```
// Listen to keyboard events
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyIsDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyIsUp);

// Create object to store key states
var keys:Dictionary = new Dictionary(true);

function keyIsDown(e:KeyboardEvent):void
{
    // Remember that the key was pressed
    keys[e.keyCode] = true;

    if (e.keyCode==Keyboard.LEFT || e.keyCode==Keyboard.RIGHT)
    {
        runningBoy.play();
    }
}

function keyIsUp(e:KeyboardEvent):void
{
    // Remember that the key was released
    keys[e.keyCode] = false;

    for each (var value:Boolean in keys)
        if ( value ) return;
    runningBoy.stop();
```

```
}

runningBoy.addEventListener(Event.ENTER_FRAME, handleMovement);
runningBoy.stop();

var currentState:Number = runningBoy.scaleX;
var speed:Number = 15;

function handleMovement(e:Event):void
{
    if (keys[Keyboard.RIGHT])
    {
        e.currentTarget.x += speed;
        e.currentTarget.scaleX = currentState;
    } else if (keys[Keyboard.LEFT])
    {
        e.currentTarget.x -= speed;
        e.currentTarget.scaleX = -currentState;
    }
}
```



キーボードの操作によるムービークリップ

「Remove」ボタンをクリックすると、表示リストからムービークリップが削除されます。

```
// Show or remove running boy
showBtn.addEventListener (MouseEvent.CLICK,showIt);
removeBtn.addEventListener (MouseEvent.CLICK,removeIt);

function showIt (e:MouseEvent):void
{
    addChild (runningBoy);
}

function removeIt(e:MouseEvent):void
{
    if (contains(runningBoy)) removeChild(runningBoy);
}
```

ムービークリップは、表示リストから削除された状態でも、引き続き Event.ENTER_FRAME イベントを送出します。ムービークリップはまだ実行されていますが、レンダリングは行われません。この状況を正しく処理するには、適切なイベントを監視し、イベントリスナーを削除して、CPU を集中的に使用するコードが実行されないようにします。

```
// Listen to Event.ADDED_TO_STAGE and Event.REMOVED_FROM_STAGE
runningBoy.addEventListener(Event.ADDED_TO_STAGE,activate);
runningBoy.addEventListener(Event.REMOVED_FROM_STAGE,deactivate);

function activate(e:Event):void
{
    // Restart everything
    e.currentTarget.addEventListener(Event.ENTER_FRAME,handleMovement);
}

function deactivate(e:Event):void
{
    // Freeze the running boy - consumes fewer CPU resources when not shown
    e.currentTarget.removeEventListener(Event.ENTER_FRAME,handleMovement);
    e.currentTarget.stop();
}
```

「Show」ボタンをクリックすると、ムービークリップが再び開始され、再度 Event.ENTER_FRAME イベントが監視されて、キーボードは正しくムービークリップを制御します。

注意：表示リストから表示オブジェクトを削除する場合、削除後にその参照先を null に設定しても、オブジェクトがフリーズされるとは限りません。ガベージコレクションを実行しないと、オブジェクトが表示されていない場合でも、そのオブジェクトはメモリおよび CPU を消費し続けます。オブジェクトの CPU 消費量を最小限に抑えるには、オブジェクトを表示リストから削除するときに、オブジェクトが完全にフリーズされていることを確認してください。

また、Flash Player 10 および AIR 1.5 以降では、次のように動作します。再生ヘッドが空のフレームを検出すると、フリーズ機能が実装されていない場合でも、表示オブジェクトが自動的にフリーズされます。

フリーズの概念は、Loader クラスでリモートコンテンツを読み込む場合にも重要です。Flash Player 9 および AIR 1.0 で Loader クラスを使用している場合は、コンテンツを手動でフリーズする必要があります。これを行うには、LoaderInfo オブジェクトが送出する Event.UNLOAD イベントを監視します。すべてのオブジェクトを手動でフリーズするのは面倒な作業です。Flash Player 10 および AIR 1.5 では、Loader クラスに unloadAndStop() と呼ばれる新しい重要なメソッドが導入されています。このメソッドを使用すると、SWF ファイルをアンロードし、ロード済みの SWF ファイルに含まれているすべてのオブジェクトを自動的にフリーズして、ガベージコレクターを強制的に実行できます。

次のコードは、SWF ファイルをロードした後で unload() メソッドを使用してアンロードします。この処理はリソースを消費し、手動によるフリーズが必要になります。

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with no automatic object deactivation
    // All deactivation must be processed manually
    loader.unload();
}
```

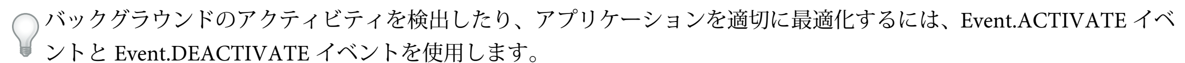
ベストプラクティスは、unloadAndStop() メソッドを使用して、フリーズをネイティブに処理し、ガベージコレクション処理を強制的に実行することです。

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with automatic object deactivation
    // All deactivation is handled automatically
    loader.unloadAndStop();
}
```

unloadAndStop() メソッドが呼び出されたときに発生するアクションを以下に示します。

- サウンドが停止します。
- SWF ファイルのメインタイムラインに登録されたリスナーが削除されます。
- Timer オブジェクトが停止します。
- ハードウェア周辺機器（カメラやマイクなど）が解放されます。
- すべてのムービークリップが停止します。
- Event.ENTER_FRAME、Event.FRAME_CONSTRUCTED、Event.EXIT_FRAME、Event.ACTIVATE および Event.DEACTIVATE イベントが送出されなくなります。

イベントのアクティブ化と非アクティブ化



バックグラウンドのアクティビティを検出したり、アプリケーションを適切に最適化するには、Event.ACTIVATE イベントと Event.DEACTIVATE イベントを使用します。

2 つのイベント（Event.ACTIVATE と Event.DEACTIVATE）を使用して、最小限の CPU サイクルを使用するようにアプリケーションを微調整できます。これらのイベントを使用すると、ランタイムがフォーカスを取得したことや失ったことを検出できます。そのため、コンテキストの変更に応じてコードを最適化できます。次のコードでは、両方のイベントをリッスンし、アプリケーションがフォーカスを失ったときに、動的にフレームレートをゼロに変更しています。例えば、ユーザーが別のタブに切り替えたり、アプリケーションをバックグラウンドに移動したときに、アニメーションのフォーカスを失うことができます。

```
var originalFrameRate:uint = stage.frameRate;
var standbyFrameRate:uint = 0;

stage.addEventListener ( Event.ACTIVATE, onActivate );
stage.addEventListener ( Event.DEACTIVATE, onDeactivate );

function onActivate ( e:Event ):void
{
    // restore original frame rate
    stage.frameRate = originalFrameRate;
}

function onDeactivate ( e:Event ):void
{
    // set frame rate to 0
    stage.frameRate = standbyFrameRate;
}
```

アプリケーションが再びフォーカスを取得すると、フレームレートは元の値にリセットされます。フレームレートを動的に変更する代わりに、オブジェクトのフリーズとフリーズ解除などのほかの最適化を作成しておくこともできます。

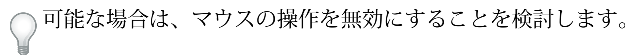
アクティブ化イベントと非アクティブ化イベントを使用すると、一部のモバイルデバイスやネットブックで提供されている「一時停止と再開」機能と同様のメカニズムを実装できます。

関連項目

47 ページの「アプリケーションのフレームレート」

25 ページの「オブジェクトのフリーズとフリーズ解除」

マウスの操作

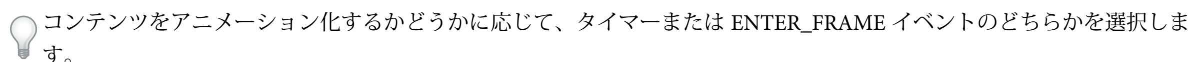

可能な場合は、マウスの操作を無効にすることを検討します。

MovieClip、Sprite オブジェクトなどのインタラクティブオブジェクトを使用すると、ランタイムはネイティブコードを実行してマウスの操作を検出および処理します。数多くのインタラクティブオブジェクトが画面に表示されていると（特にオーバーラップする場合）、マウス操作の検出により、CPU が集中的に使用される場合があります。この処理を避ける簡単な方法は、マウスの操作を必要としないオブジェクトではマウス操作を無効にすることです。次のコードは、mouseEnabled プロパティおよび mouseChildren プロパティの使用方法を示しています。

```
// Disable any mouse interaction with this InteractiveObject
myInteractiveObject.mouseEnabled = false;
const MAX_NUM:int = 10;

// Create a container for the InteractiveObjects
var container:Sprite = new Sprite();

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    // Add InteractiveObject to the container
    container.addChild( new Sprite() );
}

// Disable any mouse interaction on all the children
container.mouseChildren = false;
```

可能な場合は、マウスの操作を無効にすることを検討します。これにより、アプリケーションによる CPU 処理の使用が減り、結果としてモバイルデバイス上でのバッテリー使用が減ります。

タイマーと ENTER_FRAME イベント


コンテンツをアニメーション化するかどうかに応じて、タイマーまたは ENTER_FRAME イベントのどちらかを選択します。

長時間実行されるアニメーション化されないコンテンツには、Event.ENTER_FRAME イベントよりもタイマーをお勧めします。

ActionScript 3.0 では、特定の間隔で関数を呼び出すために 2 つの方法があります。最初の方法は、表示オブジェクト（DisplayObject）によって送出される Event.ENTER_FRAME イベントの使用です。2 番目の方法は、タイマーの使用です。ActionScript 開発者は、ENTER_FRAME イベントによる方法をよく使用します。ENTER_FRAME イベントは各フレームで送出されます。その結果、関数を呼び出す間隔は、現在のフレームレートに関連します。フレームレートは、

Stage.frameRate プロパティによってアクセス可能です。ただし、場合によっては、タイマーを使用する方が ENTER_FRAME イベントを使用するよりも良い選択であることがあります。例えば、アニメーションを使用しないが、特定の間隔でコードを呼び出す場合は、タイマーを使用することをお勧めします。

タイマーは ENTER_FRAME イベントと同様に動作しますが、フレームレートとは無関係にイベントを送出できます。このビヘイビアーにより、大幅な最適化を実現できる場合があります。例えば、ビデオプレーヤーアプリケーションを考えてみましょう。この場合、高いフレームレートを使用する必要はありません。移動するのはアプリケーションコントロールのみであるためです。

注意：ビデオはタイムラインに埋め込まれないので、フレームレートはビデオに影響しません。その代わり、ビデオはプログレッシブダウンロードまたはストリーミングによって動的にロードされます。

この例では、フレームレートは 10 fps の低い値に設定されます。タイマーは、1 秒あたり 1 回の更新レートでコントロールを更新します。より高い更新レートは、updateAfterEvent() メソッドによって可能になります。このメソッドは TimerEvent オブジェクトで利用できます。このメソッドはタイマーがイベントを送出するたびに、必要に応じて画面を更新します。次のコードは、この考えを示しています。

```
// Use a low frame rate for the application
stage.frameRate = 10;

// Choose one update per second
var updateInterval:int = 1000;
var myTimer:Timer = new Timer(updateInterval,0);

myTimer.start();
myTimer.addEventListener( TimerEvent.TIMER, updateControls );

function updateControls( e:TimerEvent ):void
{
    // Update controls here
    // Force the controls to be updated on screen
    e.updateAfterEvent();
}
```

updateAfterEvent() イベントを呼び出しても、フレームレートは変更されません。ランタイムが、変更されたコンテンツを画面上で更新するだけです。タイムラインは引き続き 10 fps で実行されます。低パフォーマンスのデバイスの場合や、イベントハンドラー関数に負荷の高い処理が含まれている場合、タイマーおよび ENTER_FRAME イベントは完全には正確ではありません。SWF ファイルフレームレートと同様に、タイマーの更新フレームレートは、特定の状況では変化する可能性があります。

アプリケーション内の Timer オブジェクト数および登録する enterFrame ハンドラー数は最小限に抑えます。

各フレームでは、ランタイムから表示リストの各表示オブジェクトに対して enterFrame イベントが送出されます。複数の表示オブジェクトで enterFrame イベントのリスナーを登録できますが、この場合、各フレームで実行されるコードが増えます。代わりに、一元的に enterFrame ハンドラーを使用して、各フレームで実行する必要があるコードをすべて実行する方法があります。このコードを一元的にまとめることにより、頻繁に実行するすべてのコードを管理しやすくなります。

また、Timer オブジェクトを使用している場合、複数の Timer オブジェクトによる作成や送出のイベントに関係するオーバヘッドが発生します。異なる操作を異なる間隔でトリガーする必要がある場合の実現方法をいくつか次に示します。

- 発生する頻度に従って、Timer オブジェクトおよびグループ操作の数を最小限に抑えます。

  例えば、頻度が高い操作に 1 つの Timer を使用し、100 ミリ秒ごとにトリガーするように設定します。頻度が低い操作またはバックグラウンド操作には別の Timer を使用し、2000 ミリ秒ごとにトリガーするように設定します。

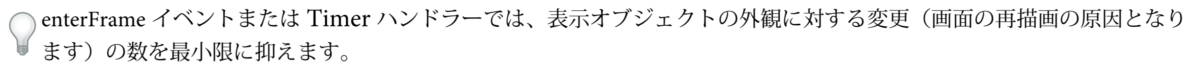
- 単一の Timer オブジェクトを使用し、Timer オブジェクトの delay プロパティ間隔の倍数で操作をトリガーします。

例えば、100 ミリ秒ごとに発生させる操作と、200 ミリ秒ごとに発生させる操作があるとします。この場合、100 ミリ秒の delay 値を指定した単一の Timer オブジェクトを使用します。timer イベントハンドラーに、1 回おきに 200 ミリ秒の操作だけを実行するという条件文を追加します。次の例は、この方法を示しています。

```
var timer:Timer = new Timer(100);
timer.addEventListener(TimerEvent.Timer, timerHandler);
timer.start();

var offCycle:Boolean = true;

function timerHandler(event:TimerEvent):void
{
    // Do things that happen every 100 ms

    if (!offCycle)
    {
        // Do things that happen every 200 ms
    }

    offCycle = !offCycle;
}
```

使用していないときは Timer オブジェクトを停止します。

Timer オブジェクトの timer イベントハンドラーが特定の条件でのみ操作を実行する場合、該当する条件が 1 つも成立しないときは stop() メソッドを呼び出します。


enterFrame イベントまたは Timer ハンドラーでは、表示オブジェクトの外観に対する変更（画面の再描画の原因となります）の数を最小限に抑えます。

各フレームのレンダリングフェーズでは、そのフレームで、変更されたステージの部分が再描画されます。再描画の領域が大きい場合や、小さくても大量な、または複雑な表示オブジェクトが含まれる場合、ランタイムのレンダリングにかかる時間は長くなります。必要な再描画の量をテストするには、デバッグの Flash Player または AIR で「再描画する領域を表示」機能を使用します。

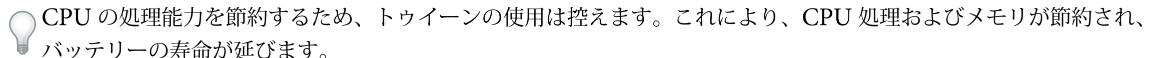
繰り返しのアクションに関するパフォーマンスを改善する方法について詳しくは、次の記事を参照してください。

- 「Writing well-behaved, efficient, AIR applications」（Arno Gourdol による記事およびサンプルアプリケーション）

関連項目

58 ページの「動作の分離」

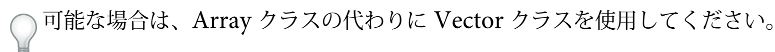
トゥイーン使用による問題


CPU の処理能力を節約するため、トゥイーンの使用は控えます。これにより、CPU 処理およびメモリが節約され、バッテリーの寿命が延びます。

デスクトップ上の Flash 用コンテンツを作成するデザイナーや開発者は、アプリケーションで多くのモーショントゥイーンを使用する傾向があります。低パフォーマンスのモバイルデバイス用コンテンツを作成するときは、モーショントゥイーンの使用を最小限にするようにしてください。これにより、低パフォーマンスのデバイスでコンテンツの実行速度が高まります。

第 4 章：ActionScript 3.0 のパフォーマンス

Vector クラスと Array クラス


可能な場合は、Array クラスの代わりに Vector クラスを使用してください。

Vector クラスを使用すると、Array クラスよりも高速な読み込みおよび書き込みアクセスが可能です。

単純なベンチマークによって、Array クラスに対する Vector クラスの優位性が示されます。次のコードは Array クラスのベンチマークを示しています。

```
var coordinates:Array = new Array();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 107
```

次のコードは Vector クラスのベンチマークを示しています。

```
var coordinates:Vector.<Number> = new Vector.<Number>();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 72
```

この例はさらに最適化できます。それには、Vector に特定の長さを割り当て、その長さを固定値に設定します。

```
// Specify a fixed length and initialize its length
var coordinates:Vector.<Number> = new Vector.<Number>(300000, true);

var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 48
```

Vector のサイズが時間よりも先に指定されない場合、Vector の容量が不足すると、サイズが増えます。Vector のサイズが増えるたびに、メモリの新しいブロックが割り当てられます。Vector の最新の内容がメモリの新しいブロックにコピーされます。データを余分に割り当てて複製することにより、パフォーマンスに影響があります。上述のコードでは、Vector の初期サイズを指定することでパフォーマンスが最適化されています。ただし、コードの保守性は最適化されていません。保守性も改善するには、再利用された値を定数に保存します。

```
// Store the reused value to maintain code easily
const MAX_NUM:int = 300000;

var coordinates:Vector.<Number> = new Vector.<Number>(MAX_NUM, true);
var started:Number = getTimer();

for (var i:int = 0; i< MAX_NUM; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 47
```

可能な場合は、Vector オブジェクト API を使用します。これにより、実行速度が高まる可能性があります。

描画 API

コードの実行を高速化するには、描画 API を使用してください。

Flash Player 10 および AIR 1.5 には新しい描画 API が用意されており、コード実行パフォーマンスを改善することができます。この新しい API でレンダリングパフォーマンスは向上しませんが、記述が必要なコードの行数を大幅に削減できます。コードの行数が少ないと、ActionScript の実行パフォーマンスが向上します。

新しい描画 API には、次のメソッドが含まれています。

- drawPath()
- drawGraphicsData()
- drawTriangles()

注意：この章では、3D 関連のメソッドである drawTriangles() については扱いません。ただし、このメソッドはネイティブのテクスチャマッピングを処理するので、ActionScript のパフォーマンスを向上させることができます。

次のコードは、描画する線ごとに適切なメソッドを明示的に呼び出します。

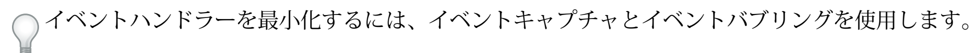
```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.moveTo ( coords[0], coords[1] );
container.graphics.lineTo ( coords[2], coords[3] );
container.graphics.lineTo ( coords[4], coords[5] );
container.graphics.lineTo ( coords[6], coords[7] );
container.graphics.lineTo ( coords[8], coords[9] );

addChild( container );
```

次のコードは、上述の例よりも実行コードの行数が少ないので、処理が高速化されます。パスが複雑化するに従って、drawPath() メソッドを使用することによって得られるパフォーマンスの向上効果が高くなります。

```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var commands:Vector.<int> = Vector.<int>([1,2,2,2,2]);
var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.drawPath(commands, coords);

addChild( container );
```

drawGraphicsData() メソッドでも、同様のパフォーマンス向上を実現できます。

イベントキャプチャとイベントバブリング


イベントハンドラーを最小化するには、イベントキャプチャとイベントバブリングを使用します。

ActionScript 3.0 のイベントモデルでは、イベントキャプチャとイベントバブリングという概念が導入されています。イベントバブリングの利点を活用すると、ActionScript コードの実行時間を最適化するうえで役立ちます。パフォーマンスを改善するには、複数のオブジェクトではなく、1 つのオブジェクトでイベントハンドラーを登録します。

例えば、ユーザーができるだけ速くりんごをクリックしてつぶす必要があるゲームの作成を考えてみましょう。このゲームでは、クリックするとりんごが画面から消え、ユーザーの得点が加算されます。りんごが送出する MouseEvent.CLICK イベントを監視するには、次のコードを記述することができます。

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    addChild ( currentApple );

    // Listen to the MouseEvent.CLICK event
    currentApple.addEventListener ( MouseEvent.CLICK, onAppleClick );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.currentTarget as InteractiveObject;
    currentAppleClicked.removeEventListener(MouseEvent.CLICK, onAppleClick );
    removeChild ( currentAppleClicked );
}
```

このコードでは、各 Apple インスタンスで addEventListener() メソッドを呼び出します。また、りんごがクリックされると、removeEventListener() メソッドを使用して各リスナーが削除されます。ただし、ActionScript 3.0 のイベントモデルは、一部のイベントに対してキャプチャフェーズおよびバブリングフェーズを提供し、親 InteractiveObject からそれらを監視できるようにします。その結果、上述のコードを最適化し、addEventListener() メソッドおよび removeEventListener() メソッドの呼び出し回数を最小化することが可能になります。次のコードでは、キャプチャフェーズを使用して親オブジェクトからイベントを監視します。

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;
var container:Sprite = new Sprite();

addChild ( container );

// Listen to the MouseEvent.CLICK on the apple's parent
// Passing true as third parameter catches the event during its capture phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, true );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    container.addChild ( currentApple );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

このコードは単純化されていて、はるかに最適化されています。親コンテナで addEventListener() メソッドを 1 回呼び出すだけです。リスナーは Apple インスタンスに登録されなくなるので、りんごがクリックされたときに Apple インスタンスを削除する必要はありません。イベントの反映を中止することで、onAppleClick() ハンドラーをさらに最適化できます。これにより、それ以上のイベントの反映を防ぐことができます。

```
function onAppleClick ( e:MouseEvent ):void
{
    e.stopPropagation();
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

バブリングフェーズはイベントのキャッチにも使用できます。そのためには、addEventListener() メソッドの 3 つ目のパラメーターとして false を渡します。

```
// Listen to the MouseEvent.CLICK on apple's parent
// Passing false as third parameter catches the event during its bubbling phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, false );
```

キャプチャフェーズパラメーターのデフォルト値は false なので、省略できます。

```
container.addEventListener ( MouseEvent.CLICK, onAppleClick );
```

ピクセルの操作

ピクセルのペイントには setVector() メソッドを使用してください。

ピクセルのペイント時に、BitmapData クラスの適切なメソッドを使用することにより、簡単な最適化を行うことができます。setVector() メソッドを使用すると、ピクセルを簡単にペイントできます。

```
// Image dimensions
var wdth:int = 200;
var hght:int = 200;
var total:int = wdth*hght;

// Pixel colors Vector
var pixels:Vector.<uint> = new Vector.<uint>(total, true);

for ( var i:int = 0; i< total; i++ )
{
    // Store the color of each pixel
    pixels[i] = Math.random()*0xFFFFFF;
}

// Create a non-transparent BitmapData object
var myImage:BitmapData = new BitmapData ( wdth, hght, false );
var imageContainer:Bitmap = new Bitmap ( myImage );

// Paint the pixels
myImage.setVector ( myImage.rect, pixels );
addChild ( imageContainer );
```

処理の遅いメソッド（setPixel() または setPixel32() など）を使用している場合は、lock() メソッドおよび unlock() メソッドを使用して処理を高速化します。次のコードでは、lock() メソッドおよび unlock() メソッドを使用してパフォーマンスを向上させています。

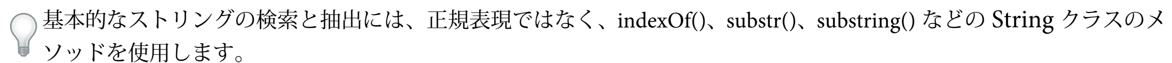
```
var buffer:BitmapData = new BitmapData(200,200,true,0xFFFFFFFF);
var bitmapContainer:Bitmap = new Bitmap(buffer);
var positionX:int;
var positionY:int;

// Lock update
buffer.lock();
var starting:Number=getTimer();

for (var i:int = 0; i<2000000; i++)
{
    // Random positions
    positionX = Math.random()*200;
    positionY = Math.random()*200;
    // 40% transparent pixels
    buffer.setPixel32( positionX, positionY, 0x66990000 );
}

// Unlock update
buffer.unlock();
addChild( bitmapContainer );

trace( getTimer () - starting );
// output : 670
```

BitmapData クラスの lock() メソッドでイメージをロックし、BitmapData オブジェクトが変更されたときに、ロックされたイメージを参照しているオブジェクトが更新されないようにします。例えば、Bitmap オブジェクトが BitmapData オブジェクトを参照している場合、その BitmapData オブジェクトをロックしてから変更した後で、ロック解除します。Bitmap オブジェクトは、BitmapData のロックが解除されるまで変更されません。パフォーマンスを向上させるには、setPixel() メソッドまたは setPixel32() メソッドが繰り返し呼び出される前と後に、このメソッドと unlock() メソッドを組み合わせて使用します。lock() メソッドおよび unlock() メソッドを呼び出すことにより、必要以上に画面が更新されるのを防ぐことができます。

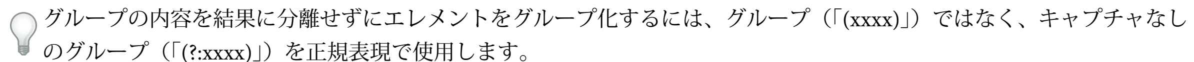
注意：表示リストではなくビットマップ上でピクセルを処理する場合（ダブルバッファリング）、このテクニックではパフォーマンスを向上できない場合があります。ビットマップオブジェクトがビットマップバッファーを参照しない場合は、lock() および unlock() を使用してもパフォーマンスは向上しません。Flash Player ではバッファーが参照されていないことを検出し、ビットマップは画面にレンダリングされません。

getPixel(9、getPixel32()、setPixel()、setPixel32() など、ピクセルで繰り返されるメソッドは、特に低速になる可能性があります（特にモバイルデバイス上）。可能な場合は、1 回の呼び出しですべてのピクセルを取得するメソッドを使用します。ピクセルの読み込みには、getVector() メソッドを使用します。このメソッドは getPixels() メソッドよりも高速です。また、可能な場合は、Vector オブジェクトに依存している API を使用します。これにより、処理が高速化されます。

正規表現


基本的なストリングの検索と抽出には、正規表現ではなく、indexOf()、substr()、substring() などの String クラスのメソッドを使用します。

正規表現を使用して実行できる操作には、String クラスのメソッドを使用して実行できるものもあります。例えば、文字列に別の文字列が含まれるかどうかを検索するには、String.indexOf() メソッドまたは正規表現を使用できます。ただし、String クラスメソッドが使用できる場合、同等の正規表現よりも実行速度が速く、別のオブジェクトを作成する必要がありません。


グループの内容を結果に分離せずにエレメントをグループ化するには、グループ（「(xxxx)」）ではなく、キャプチャなしのグループ（「(?:xxxx)」）を正規表現で使用します。

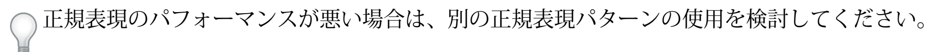
複雑さが中程度の正規表現では、式の一部をグループ化することがよく行われます。例えば、次の正規表現パターンでは、括弧を使用してテキスト「ab」を囲むグループを作成しています。結果として、「+」限量詞は単一の文字ではなくグループに適用されます。

```
/(ab)+/
```

デフォルトでは、各グループの内容が「キャプチャ」されます。正規表現を実行した結果の一部として、パターン内の各グループの内容を取得できます。このようなグループ結果の取得には時間がかかり、必要なメモリも増えます。これは、グループの結果を含むオブジェクトが作成されるからです。代替方法として、開始括弧の後に疑問符とコロンを含めることで、キャプチャなしのグループ構文を使用できます。この構文では、複数の文字がグループとして動作しますが、結果にはキャプチャされません。

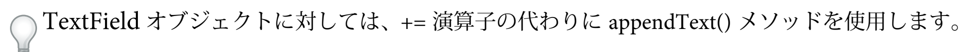
```
/(?:ab)+/
```

キャプチャなしのグループ構文を使用する方が、標準的なグループ構文を使用するよりも高速で、使用するメモリも減ります。


正規表現のパフォーマンスが悪い場合は、別の正規表現パターンの使用を検討してください。

同じテキストパターンのテストや特定に、複数の正規表現パターンを使用できる場合があります。様々な原因により、あるパターンとその代替パターンでは実行速度が異なります。正規表現によりコードの実行速度が必要以上に遅くなっていると判断した場合は、同じ結果を得られる代替正規表現パターンを使用することを検討してください。考えられる各種の代替パターンをテストし、どのパターンが最速かを判断してください。

その他の最適化


TextField オブジェクトに対しては、+= 演算子の代わりに appendText() メソッドを使用します。

TextField クラスの text プロパティを操作する場合は、+= 演算子の代わりに appendText() メソッドを使用します。appendText() メソッドを使用すると、パフォーマンスが向上します。

例えば、次のコードでは += 演算子を使用しており、ループ処理の実行時間は 1,120 ミリ秒です。

```
addChild ( myTextField );

myTextField.autoSize = TextFieldAutoSize.LEFT;
var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.text += "ActionScript 3";
}

trace( getTimer() - started );
// output : 1120
```

次の例では、+= 演算子を appendText() メソッドに置き換えています。

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.appendText ( "ActionScript 3" );
}

trace( getTimer() - started );
// output : 847
```

これで、このコードの実行時間は 847 ミリ秒になります。

可能な場合は、ループの外でテキストフィールドを更新してください。

このコードは、簡単なテクニックを利用してさらに最適化することができます。各ループの内部でテキストフィールドを更新すると、ループ内処理が増加します。ストリングを結合し、ループの外でストリングにテキストフィールドに割り当てるだけで、コードの実行時間が大幅に短縮されます。これで、コードの実行時間は 2 ミリ秒になります。

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.text;

for (var i:int = 0; i< 1500; i++ )
{
    content += "ActionScript 3";
}

myTextField.text = content;

trace( getTimer() - started );
// output : 2
```

HTML テキストを操作する場合は、前者の方法では処理に時間がかかり、場合によっては Flash Player で Timeout 例外がスローされます。例えば、基になるハードウェアが非常に低速の場合、例外がスローされます。

注意：Adobe® AIR® では、この例外はスローされません。

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.htmlText += "ActionScript <b>2</b>";
}

trace( getTimer() - started );
```

ループの外でストリングに値を代入することにより、コードの実行時間はわずか 29 ミリ秒に短縮されます。

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.htmlText;

for (var i:int = 0; i< 1500; i++ )
{
    content += "<b>ActionScript<b> 3";
}

myTextField.htmlText = content;

trace ( getTimer() - started );
// output : 29
```

注意：Flash Player 10.1 および AIR 2.5 では、String クラスが改善され、ストリングのメモリ使用量が低減しています。

可能な場合は、角括弧演算子の使用を避けてください。

角括弧演算子を使用すると、パフォーマンスが低下する可能性があります。参照をローカル変数に格納することにより、この演算子の使用を回避できます。次のコード例は、角括弧演算子の非効率な使用を示しています。

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();

for ( i = 0; i< lng; i++ )
{
    arraySprite[i].x = Math.random()*stage.stageWidth;
    arraySprite[i].y = Math.random()*stage.stageHeight;
    arraySprite[i].alpha = Math.random();
    arraySprite[i].rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 16
```

次のように最適化して、角括弧演算子の使用回数を減らすことができます。

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();
var currentSprite:Sprite;

for ( i = 0; i< lng; i++ )
{
    currentSprite = arraySprite[i];
    currentSprite.x = Math.random()*stage.stageWidth;
    currentSprite.y = Math.random()*stage.stageHeight;
    currentSprite.alpha = Math.random();
    currentSprite.rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 9
```

可能な場合はコードをインライン化して、コード内の関数呼び出しの回数を減らしてください。

関数呼び出しは負荷を増加させる可能性があります。コードをインライン化して、関数呼び出しの回数を削減するようにします。コードのインライン化は、純粋なパフォーマンスの最適化に適した方法です。ただし、インラインコードではコードの再利用が困難で、SWF ファイルのサイズが大きくなる可能性があるので注意してください。一部の関数呼び出し（Math クラスのメソッドなど）は、簡単にインライン化できます。次のコードでは、Math.abs() メソッドを使用して絶対値を計算します。

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = Math.abs ( currentValue );
}

trace( getTimer() - started );
// output : 70
```

Math.abs() による計算は、手動で記述して、インライン化することができます。

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = currentValue > 0 ? currentValue : -currentValue;
}

trace( getTimer() - started );
// output : 15
```

関数呼び出しをインライン化すると、コードは 4 倍以上も高速化されます。この方法は、様々な状況で役立ちます。ただし、コードの再利用や管理容易性に与える影響について注意する必要があります。

注意：コードサイズにより、プレーヤーの全体的な実行に大きな影響があります。アプリケーションに大量の ActionScript コードが含まれている場合、仮想マシンはコードの検証と JIT コンパイルに相当の時間を費やします。プロパティ参照が遅くなる可能性があります。これは、継承階層が深くなるのと、内部キャッシュのスラッシュが大きくなる傾向があるためです。コードサイズを減らすには、Adobe® Flex® フレームワーク、TLF フレームワークライブラリまたはサードパーティ製 ActionScript ライブラリの使用を避けてください。

ループでステートメントを評価することは避けます。

ループ内でステートメントを評価しないようにして、最適化を図ることもできます。次のコードは、配列に対して反復処理を行います。反復処理ごとに配列の長さを評価するので、コードは最適化されていません。

```
for (var i:int = 0; i< myArray.length; i++)
{
}
```

値を保存して再利用する方が確実です。

```
var lng:int = myArray.length;

for (var i:int = 0; i< lng; i++)
{
}
```

while ループには反転を使用します。

反転の while ループは前進ループよりも高速です。

```
var i:int = myArray.length;

while (--i > -1)
{
}
```

これらのヒントは、ActionScript を最適化するいくつかの方法を示しており、1 行のコードによってパフォーマンスとメモリに大きな影響があることがわかります。その他にも、ActionScript を最適化する多くの方法があります。詳しくは、http://www.rozengain.com/blog/2007/05/01/some-actionscript-30-optimizations/ を参照してください。

第 5 章：レンダリングのパフォーマンス

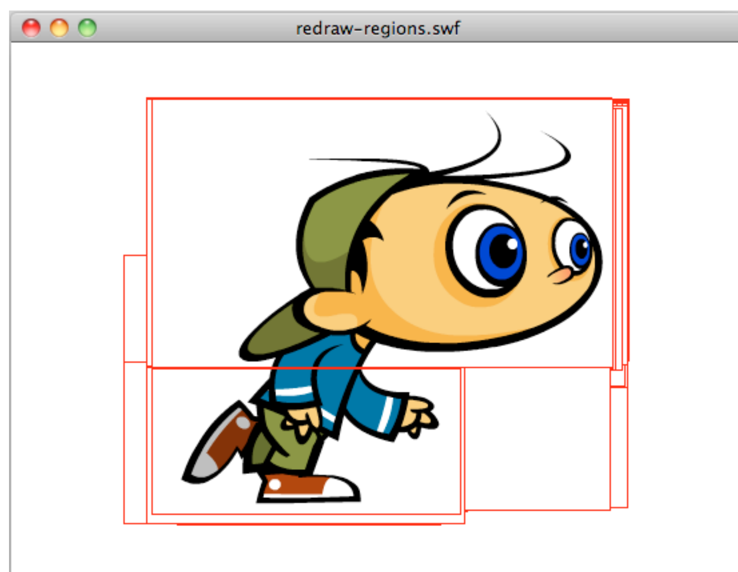
再描画領域

プロジェクトを作成するときは必ず再描画領域オプションを使用します。

レンダリングを改善するには、プロジェクトを作成するときに再描画領域オプションを使用することが重要です。このオプションを使用すると、Flash Player がレンダリングおよび処理している領域を確認できます。このオプションを有効にするには、デバッグバージョンの Flash Player のコンテキストメニューで「再描画する領域を表示」を選択します。

注意：「再描画する領域を表示」オプションは、Adobe AIR およびリリースバージョンの Flash Player では使用できません（Adobe AIR の場合、このコンテキストメニューはデスクトップアプリケーションでのみ使用できますが、「再描画する領域を表示」などの組み込みまたは標準のメニューアイテムは用意されていません)。

以下の画像は、タイムライン上のシンプルなアニメーション MovieClip に対してこのオプションを有効にしたところです。



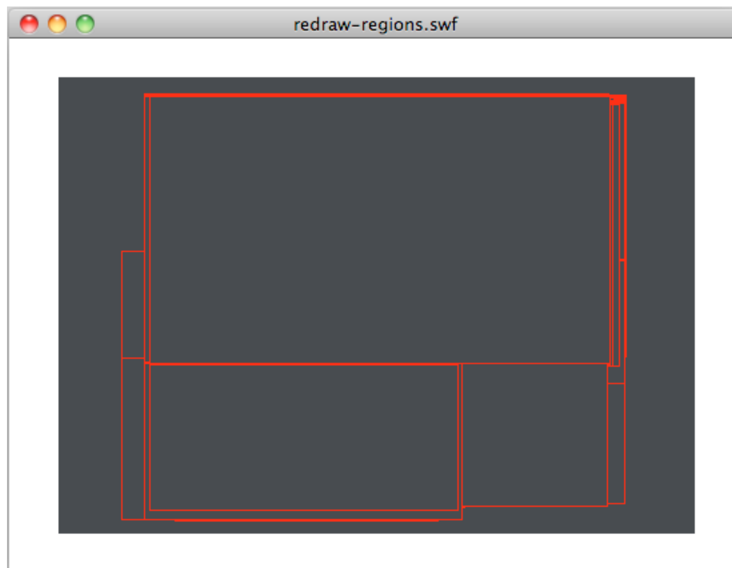
再描画領域オプションの有効化

このオプションは、プログラムから flash.profiler.showRedrawRegions() メソッドを使用して有効にすることもできます。

```
// Enable Show Redraw Regions
// Blue color is used to show redrawn regions
flash.profiler.showRedrawRegions ( true, 0x0000FF );
```

Adobe AIR アプリケーションでは、このメソッド以外に再描画領域オプションを有効にする方法はありません。

再描画領域を使用して、最適化に適した機会を特定します。表示オブジェクトの中には表示されないものがありますが、レンダリングは行われているので、そうしたオブジェクトが CPU サイクルを消費する可能性があります。次の画像は、これを示しています。走るキャラクターのアニメーションが、黒いベクターシェイプに覆い隠されています。この画像は、表示オブジェクトが表示リストから削除されておらず、レンダリングが続行していることを示しています。これにより、CPU サイクルが浪費されています。



再描画領域

パフォーマンスを改善するには、非表示の走るキャラクターの visible プロパティを false に設定するか、表示リストから削除します。また、そのタイムラインを停止します。これにより、表示オブジェクトがフリーズし、CPU の使用を最小限に抑えることができます。

開発サイクル中は必ず再描画領域オプションを使用してください。このオプションを使用すると、不必要な再描画領域や未対策の最適化領域がプロジェクトの最後に見つかってあわてるという事態を予防できます。

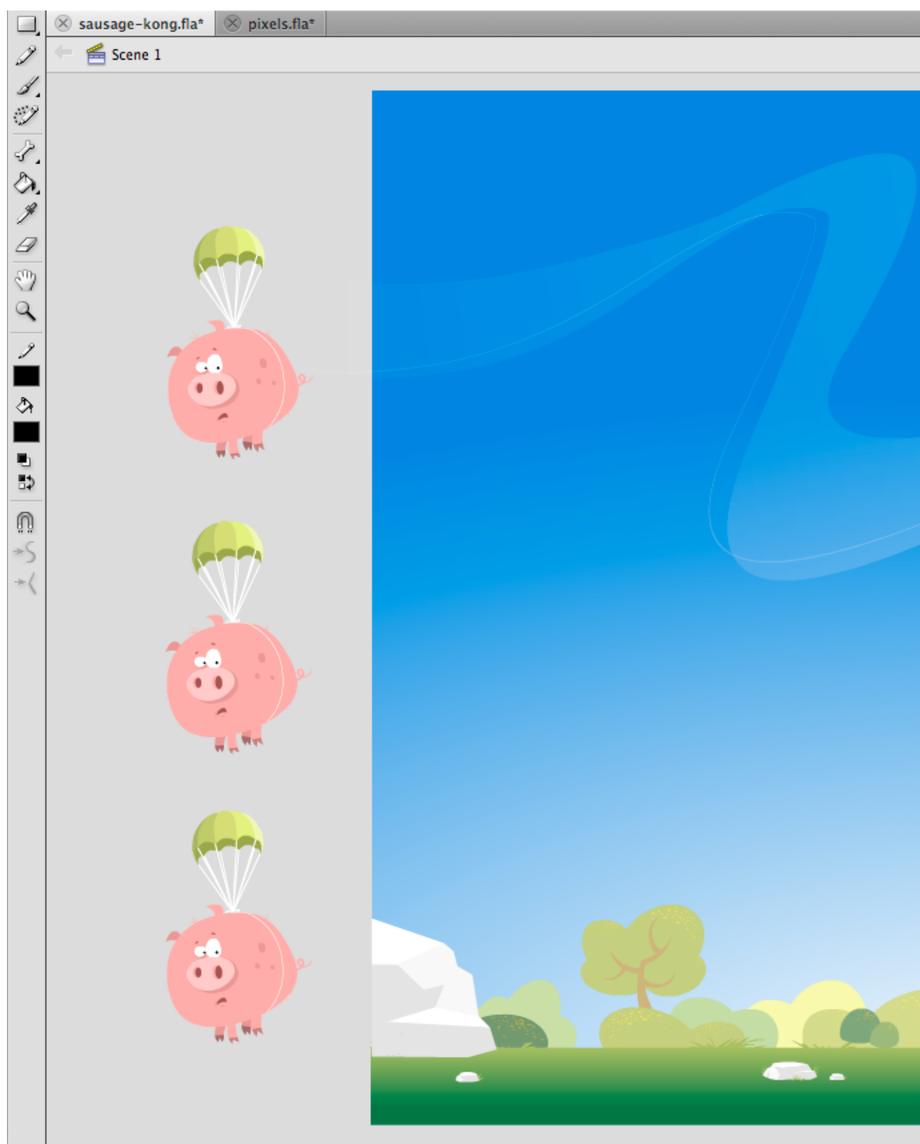
関連項目

25 ページの「オブジェクトのフリーズとフリーズ解除」

ステージ外のコンテンツ

ステージ外にコンテンツを配置することは避けてください。代わりに、必要に応じて表示リストにオブジェクトを配置します。

可能であれば、グラフィカルコンテンツをステージ外に配置しないようにしてください。一般的に、デザイナーと開発者は、アプリケーションの有効期間中、ステージ外にエレメントを配置して、アセットを再利用します。次の図は、このような一般的なテクニックを示しています。



ステージ外のコンテンツ

ステージ外のエレメントは、画面上に表示されず、レンダリングされていなくても、表示リストには存在します。ランタイムは、これらのエレメントの内部テストを実行し続け、エレメントがステージ外に存在していること、およびユーザーがエレメントを操作していないことを確認します。そのため、可能な限り、オブジェクトをステージ外に配置することは避け、代わりにオブジェクトを表示リストから削除するようにしてください。

ムービーの画質

ステージの適切な品質設定を使用して、レンダリングを改善してください。

携帯電話など、画面の小さいモバイルデバイス用のコンテンツ開発では、デスクトップアプリケーション開発と比較して、画質はそれほど重要ではありません。ステージの品質を適切に設定すれば、レンダリングパフォーマンスを向上させることができます。

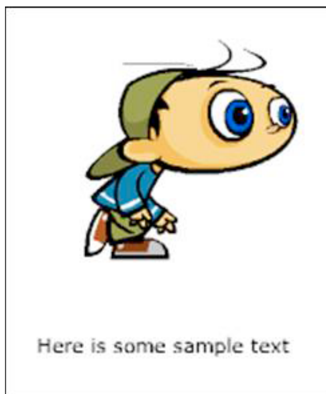
ステージの品質には、次の設定を使用できます。

- StageQuality.LOW：画質よりも再生スピードを優先し、アンチエイリアスを使用しません。この設定は、デスクトップ用および TV 用の Adobe AIR ではサポートされません。
- StageQuality.MEDIUM：アンチエイリアスを使用しますが、拡大 / 縮小されたビットマップはスムージング処理されません。この設定は、モバイルデバイス上での AIR のデフォルト値ですが、デスクトップ用 AIR またはテレビ用 AIR ではサポートされていません。
- StageQuality.HIGH：（デスクトップではデフォルト）再生スピードよりも画質を優先し、必ずアンチエイリアスを使用します。SWF ファイルにアニメーションが組み込まれていない場合は、ビットマップがスムージング処理され、アニメーションが組み込まれている場合は、拡大 / 縮小されたビットマップがスムージングされません。
- StageQuality.BEST：最高の画質を実現し、再生スピードは考慮しません。すべての出力に対してアンチエイリアスが行われ、拡大 / 縮小されたビットマップが常にスムージングされます。

通常は、StageQuality.MEDIUM を使用すれば、モバイルデバイス用のアプリケーションに十分な画質を確保できます。StageQuality.LOW でも十分な画質を提供できる場合があります。Flash Player 8 以降は、ステージの品質を LOW に設定している場合でも、アンチエイリアス処理されたテキストを正確にレンダリングできます。

注意：一部のモバイルデバイスでは、品質が HIGH に設定されている場合でも、Flash Player アプリケーションのパフォーマンスを向上させるために MEDIUM が使用されます。ただし、品質を HIGH に設定しても、モバイル画面の dpi は通常高いので、ほとんど見分けがつきません（dpi はデバイスによって異なる場合があります）。

次の図では、ムービーの画質は MEDIUM に、テキストレンダリングは「アンチエイリアス（アニメーション優先）」に設定されています。



ステージの品質が MEDIUM で、テキストレンダリングを「アンチエイリアス（アニメーション優先）」に設定した場合

適切なテキストレンダリング設定が使用されていないと、ステージの品質設定がテキスト品質に影響します。

テキストレンダリングを「アンチエイリアス（読みやすさ優先）」に設定することもできます。この設定は、使用しているステージの品質に関わらず、（アンチエイリアスが適用された）最高品質のテキストを維持します。

ステージの品質が LOW で、テキストレンダリングを「アンチエイリアス（読みやすさ優先）」に設定した場合

テキストレンダリングを「ビットマップテキスト（アンチエイリアスなし）」に設定することにより、同様のレンダリング品質が得られます。

ステージの品質が LOW で、テキストレンダリングを「ビットマップテキスト（アンチエイリアスなし）」に設定した場合

最後の 2 例は、使用しているステージの品質設定に関わらず、高品質のテキストを得ることができることを示しています。この機能は Flash Player 8 以降で利用でき、モバイルデバイス上で使用できます。Flash Player 10.1 では、パフォーマンスを向上させるため、一部のデバイスでは自動的に StageQuality.MEDIUM に切り替われます。

アルファブレンディング

可能な場合は、アルファプロパティの使用を避けます。

alpha プロパティを使用するときは、アルファブレンディングを必要とする効果（フェード効果など）の使用を避けます。表示オブジェクトがアルファブレンディングを使用している場合、ランタイムは積み重ねられた表示オブジェクトのカラー値と背景色を組み合わせて、最終的なカラーを決定する必要があります。したがって、アルファブレンディングは不透明なカラーの描画よりも CPU に負荷がかかる可能性があります。こうした余分の計算によって、低速デバイスではパフォーマンスに影響する可能性があります。可能な場合は、alpha プロパティの使用を避けます。

関連項目

48 ページの「ビットマップキャッシュ」

61 ページの「テキストオブジェクトのレンダリング」

アプリケーションのフレームレート

一般的に、パフォーマンスを改善するには、可能な限り低いフレームレートを使用します。

アプリケーションのフレームレートによって、「アプリケーションコードおよびレンダリング」の各サイクルに使用できる時間が決まります。詳しくは、1 ページの「ランタイムコードの実行の基礎」を参照してください。フレームレートが高いほど、アニメーションの動きがスムーズになります。ただし、多くの場合、アニメーションや他の視覚的な変化がないときに高いフレームレートを使用する理由はありません。フレームレートが高くなると、CPU サイクルとバッテリーの消費が増えます。

アプリケーションに適したデフォルトフレームレートを選択する際の一般的なガイドラインを次に示します。

- Flex フレームワークを使用している場合、開始のフレームレートをデフォルト値のままにします。
- アプリケーションにアニメーションが含まれる場合、フレームレートとして適切なのは 20 フレーム / 秒以上です。通常、30 フレーム / 秒を超えるレートは必要ありません。
- アプリケーションにアニメーションが含まれない場合、ほとんどは 12 フレーム / 秒のフレームレートで十分です。

「可能な限り低いフレームレート」は、アプリケーションの現在のアクティビティによって変わります。詳しくは、次のヒント「アプリケーションのフレームレートの動的な変更」を参照してください。

アプリケーションの動的なコンテンツがビデオだけの場合、低いフレームレートを使用します。

ロードされたビデオコンテンツは、アプリケーションのフレームレートに関係なく、ネイティブフレームレートで実行されます。アプリケーションにアニメーションや他の高速に変化するビジュアルコンテンツがない場合、低いフレームレートを使用してもユーザーインターフェイスの操作性は低下しません。

アプリケーションのフレームレートを動的に変更します。

プロジェクトまたはコンパイラーの設定でアプリケーションの初期フレームレートを定義しますが、フレームレートはその値で固定されません。フレームレートを変更するには、Stage.frameRate プロパティ（または Flex での WindowedApplication.frameRate プロパティ）を設定します。

フレームレートは、アプリケーションの現在の要件に従って変更します。例えば、アニメーションが実行中ではないときは、フレームレートを低くします。アニメーションのトランジションが始まるときは、フレームレートを上げます。同様に、アプリケーションがバックグラウンドで実行されている場合（フォーカスを失った後）、通常は、フレームレートをさらに低くできます。ユーザーの意識は、他のアプリケーションやタスクに移っていると考えられます。

各種アクティビティについて、適切なフレームレートを決定するときの出発点に使用できる一般的なガイドラインを次に示します。

- Flex フレームワークを使用している場合、開始のフレームレートをデフォルト値のままにします。
- アニメーションの再生中は、フレームレートを 20 フレーム / 秒以上に設定します。通常、30 フレーム / 秒を超えるレートは必要ありません。
- アニメーションが再生中ではない場合、ほとんどは 12 フレーム / 秒のフレームレートで十分です。

- ロードされたビデオは、アプリケーションのフレームレートに関係なく、ネイティブフレームレートで再生されます。アプリケーションで動くコンテンツがビデオのみの場合、ほとんどは 12 フレーム / 秒のフレームレートで十分です。
- アプリケーションに入力フォーカスがない場合、ほとんどは 5 フレーム / 秒のフレームレートで十分です。
- AIR アプリケーションが表示されていない場合、ほとんどは 2 フレーム / 秒以下のフレームレートで十分です。例えば、このガイドラインは、アプリケーションが最小化されている場合に適用されます。また、ネイティブウィンドウの visible プロパティが false に設定されている場合にも適用されます。

Flex に組み込まれているアプリケーションの場合、spark.components.WindowedApplication クラスには、動的に変化するアプリケーションのフレームレートのサポートが組み込まれています。backgroundFrameRate プロパティに、アプリケーションがアクティブではない場合のアプリケーションのフレームレートを定義します。デフォルト値は 1 です。これは、Spark フレームワークで構築されたアプリケーションのフレームレートが 1 フレーム / 秒に変更されることを意味します。バックグラウンドのフレームレートを変更するには、backgroundFrameRate プロパティを設定します。このプロパティには他の値を設定するか、-1 を設定して自動フレームレートのスロットルをオフすることができます。

アプリケーションのフレームレートの動的な変更について詳しくは、次の記事を参照してください。

- 「Reducing CPU usage in Adobe AIR」（Jonnie Hallman によるアドビデベロッパーセンターの記事およびサンプルコード）
- 「Writing well-behaved, efficient, AIR applications」（Arno Gourdol による記事およびサンプルアプリケーション）

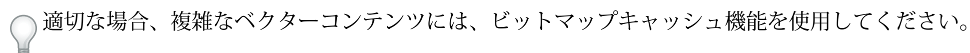
Grant Skinner 氏は、フレームレートスロットルクラスを作成しました。このクラスをアプリケーションで使用すると、アプリケーションがバックグラウンドで実行されているときに、フレームレートを自動的に低くできます。詳細および FramerateThrottler クラスのソースコードのダウンロードについては、Grant 氏の記事「Idle CPU Usage in Adobe AIR and Flash Player」を http://gskinner.com/blog/archives/2009/05/idle_cpu_usage.html で参照してください。

アダプティブフレームレート

SWF ファイルのコンパイル時に、ムービーには特定のフレームレートを設定します。CPU 速度が低く制限された環境では、再生中にフレームレートが低下する場合があります。ユーザーが許容できるフレームレートを維持するため、ランタイムでは一部のフレームのレンダリングをスキップします。一部のフレームのレンダリングをスキップすることにより、フレームレートが許容値を下回るのを防ぐことができます。

注意：この場合、ランタイムはフレームレートをスキップするのではなく、フレームのコンテンツのレンダリングのみをスキップします。コードは引き続き実行され、表示リストは更新されます。ただし、更新内容は画面に表示されません。ランタイムでフレームレートを一定に保てないと、スキップするフレーム数を示すしきい値（fps）を指定できません。

ビットマップキャッシュ



適切な場合、複雑なベクターコンテンツには、ビットマップキャッシュ機能を使用してください。

ビットマップキャッシュ機能を使用することで、有効な最適化を実行できます。この機能は、ベクターオブジェクトをキャッシュし、内部ビットマップとして表現して、そのビットマップをレンダリングに使用します。この機能を使用することで、レンダリングのパフォーマンスが大幅に向上します。ただし、大量のメモリを消費する可能性があります。ビットマップキャッシュ機能は、複雑なベクターコンテンツ（複雑なグラデーションやテキストなど）に使用します。

複雑なベクターグラフィック（テキスト、グラデーションなど）を含むアニメーション化されたオブジェクトのビットマップキャッシュを有効にすると、パフォーマンスが向上します。ただし、タイムライン再生が含まれるムービークリップなどの表示オブジェクトでビットマップキャッシュを有効にすると、逆効果になります。ランタイムが、キャッシュされたビッ

トマップをフレームごとに更新し、オンスクリーンで再描画しなければならないので、多くの CPU サイクルが必要になります。ビットマップキャッシュ機能が役立つのは、キャッシュされたビットマップを 1 回生成でき、更新の必要がなくそのビットマップを使用する場合のみです。

Sprite オブジェクトに対してビットマップキャッシュを有効にすると、キャッシュされたビットマップをランタイムが再生成することなく、オブジェクトを移動できるようになります。オブジェクトの x プロパティおよび y プロパティを変更しても、再生成は行われません。ただし、その回転、拡大 / 縮小またはアルファ値の変更を試みると、ランタイムはキャッシュされたビットマップを再生成し、その結果、パフォーマンスに影響があります。

注意：AIR および Packager for iPhone で利用できる DisplayObject.cacheAsBitmapMatrix プロパティには、この制限はありません。cacheAsBitmapMatrix プロパティを使用すると、ビットマップの再生成をトリガーすることなく、表示オブジェクトの回転、拡大 / 縮小、傾斜、回転、アルファ値の変更が行えます。

キャッシュされたビットマップは、通常のムービークリップインスタンスよりも多くのメモリを使用します。例えば、ステージ上のムービークリップが 250 x 250 ピクセルの場合、キャッシュされると約 250 KB（キャッシュされない場合は 1 KB）を使用します。

次の例では、りんごのイメージが含まれている Sprite オブジェクトを操作します。次のクラスは、りんごのシンボルにアタッチされます。

```
package org.bytearray.bitmap
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Apple extends Sprite
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function Apple ()
        {
            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener (Event.ENTER_FRAME,handleMovement);
        }

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

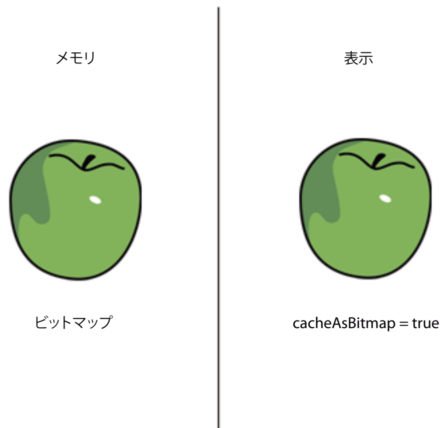
```
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

りんごにはタイムラインが必要ないので、このコードでは MovieClip クラスの代わりに Sprite クラスを使用します。パフォーマンスを最大にするには、最も軽量のオブジェクトを使用します。次に、以下に示すコードでクラスをインスタンス化します。

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(MouseEvent.CLICK,createApples);
stage.addEventListener(KeyboardEvent.KEY_DOWN,cacheApples);

const MAX_NUM:int = 100;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

function createApples(e:MouseEvent):void
{
    for (var i:int = 0; i< MAX_NUM; i++)
    {
        apple = new Apple();

        holder.addChild(apple);
    }
}

function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holder.numChildren;

        for (var i:int = 0; i < lng; i++)
        {
            apple = holder.getChildAt (i) as Apple;

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

ユーザーがマウスをクリックしたときに、キャッシュせずにりんごが作成されます。ユーザーが C キー（キーコード 67）を押すと、りんごのベクターがビットマップとしてキャッシュされ、画面に表示されます。CPU 速度が遅い場合は、この手法によって、デスクトップとモバイルデバイスの両方でレンダリングパフォーマンスが大幅に向上します。

ただし、ビットマップキャッシュ機能を使用することによりレンダリングパフォーマンスが向上しても、メモリ消費量が急激に増加する場合があります。オブジェクトがキャッシュされるとすぐに、オブジェクトのサーフェスが透明なビットマップとしてキャプチャされ、メモリに格納されます。次の図を参照してください。

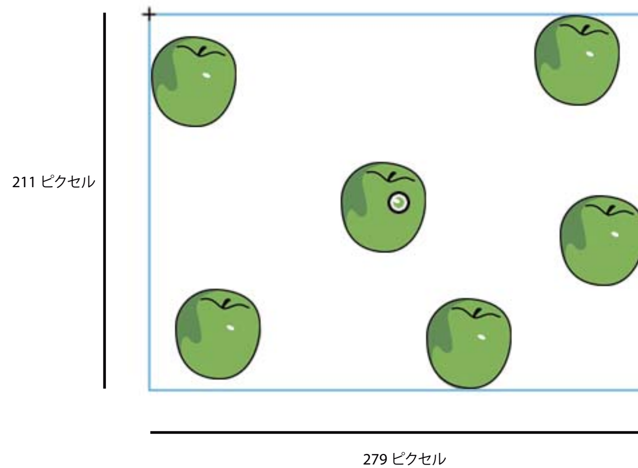


メモリに格納されたオブジェクトとそのサーフェスのビットマップ

Flash Player 10.1 および AIR 2.5 は、18 ページの「フィルターおよびビットマップの動的アンロード」に説明されているのと同じ手法を使用することにより、メモリの使用を最適化します。キャッシュされた表示オブジェクトが非表示またはオフスクリーンの場合、メモリ内のビットマップは、しばらく使用されないと解放されます。

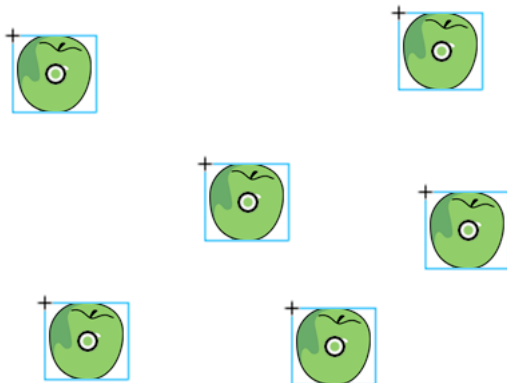
注意：表示オブジェクトの opaqueBackground プロパティが特定の色に設定されていると、ランタイムは不透明な表示オブジェクトと見なします。ランタイムは、cacheAsBitmap プロパティと共に使用すると、メモリ内に不透明な 32 ビットのビットマップを作成します。アルファチャンネルは 0xFF に設定されます。これにより、ビットマップを画面上に描画する際、透明度が必要ないので、パフォーマンスが向上します。アルファブレンディングを使用しないと、レンダリングがさらに高速化されます。現在の画面深度が 16 ビットに制限されている場合、メモリ内のビットマップは 16 ビットイメージとして格納されます。opaqueBackground プロパティを使用しても、ビットマップキャッシュは暗黙的にアクティブにはなりません。

メモリを節約するには、cacheAsBitmap プロパティを使用して、コンテナではなく各表示オブジェクト上でこのプロパティを有効にします。コンテナ上でビットマップキャッシュを有効にすると、メモリ内で最終ビットマップのサイズが大幅に増大し、211 x 279 ピクセルの 透明なビットマップが作成されます。このイメージは、約 229 KB のメモリを使用します。



コンテナ上でのビットマップキャッシュの有効化

さらに、コンテナをキャッシュすることで、りんごがフレーム上で移動し始めたときに、ビットマップ全体がメモリ内で更新されるリスクがあります。個々のインスタンス上でビットマップキャッシュを有効にすると、7 KB のサーフェスがメモリ内に 6 個キャッシュされます。つまり、42 KB のメモリしか使用されません。


インスタンス上でのビットマップキャッシュの有効化

表示リストから各りんごのインスタンスにアクセスして、getChildAt() メソッドを呼び出すと、Vector オブジェクトに参照が格納され、アクセスが容易になります。

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(KeyboardEvent.KEY_DOWN, cacheApples);

const MAX_NUM:int = 200;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<Apple> = new Vector.<Apple>(MAX_NUM, true);

for (var i:int = 0; i< MAX_NUM; i++)
{
    apple = new Apple();

    holder.addChild(apple);

    holderVector[i] = apple;
}

function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holderVector.length

        for (var i:int = 0; i < lng; i++)
        {
            apple = holderVector[i];

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

キャッシュされたコンテンツが各フレームで回転、拡大 / 縮小または変更されない場合、ビットマップキャッシュによりレンダリングが改善します。ただし、x 軸および y 軸上の移動以外の変形があると、レンダリングは改善されません。この場合、Flash Player は、表示オブジェクト上で変形が発生するたびに、キャッシュされたビットマップのコピーを更新します。キャッシュされたコピーを更新することにより、CPU 使用率が増加し、パフォーマンスが低下して、バッテリが消耗します。この場合も、AIR または Packager for iPhone の cacheAsBitmapMatrix プロパティにはこの制限がありません。

次のコードでは、移動メソッドのアルファ値を変更して、各フレーム上のりんごの不透明度を変更します。

```
private function handleMovement(e:Event):void
{
        alpha = Math.random();
        x -= (x - destinationX)*.5;
        y -= (y - destinationY)*.5;

        if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
            initPos();
}
```

ビットマップキャッシュを使用すると、パフォーマンスが低下する原因となります。ランタイムは、アルファ値が変更されるたびに、メモリ内のキャッシュされたビットマップを更新します。

フィルターは、キャッシュされたムービークリップの再生ヘッドが移動するたびに更新されるビットマップに依存します。したがって、フィルターを使用すると自動的に cacheAsBitmap プロパティが true に設定されます。次の図は、アニメーション化されたムービークリップを示しています。

アニメーション化されたムービークリップ

パフォーマンスの問題が発生する可能性があるので、アニメーションコンテンツではフィルターを使用しないようにします。次の図には、デザイナーによってドロップシャドウフィルターが追加されています。

アニメーション化されたムービークリップ（ドロップシャドウフィルター付き）

その結果、ムービークリップ内のタイムラインが再生中の場合、ビットマップは再生成される必要があります。単純な x 変形または y 変形以外の方法でコンテンツが変更される場合も、ビットマップは再生成される必要があります。ランタイムは各フレームでビットマップを再描画します。これにより、多くの CPU リソースが必要になり、パフォーマンスが低下して、バッテリが消耗します。

以下のトレーニングビデオでは、Paul Trani が Flash Professional と ActionScript を使用して、ビットマップを使用したグラフィックを最適化する例を示しています。

- Optimizing Graphics
- Optimizing Graphics with ActionScript

AIR でキャッシュされたビットマップの変換マトリックス

モバイル AIR アプリケーションで、キャッシュされたビットマップを使用する場合は、cacheAsBitmapMatrix プロパティを設定してください。

AIR モバイルのプロファイルでは、Matrix オブジェクトを表示オブジェクトの cacheAsBitmapMatrix プロパティに割り当てることができます。このプロパティを設定すると、キャッシュされたビットマップを再生成することなく、2 次元の変形をオブジェクトに適用できます。キャッシュされたビットマップを再生成することなく、アルファプロパティを変更することもできます。cacheAsBitmap プロパティには、true を設定し、そのオブジェクトの 3D プロパティは設定しないでください。

cacheAsBitmapMatrix プロパティを設定すると、表示オブジェクトが画面の外にある場合、非表示の場合、または visible プロパティに false が設定されている場合でも、キャッシュされたビットマップは生成されます。異なる変形を含むマトリックスオブジェクトを使用して cacheAsBitmapMatrix プロパティを再設定しても、キャッシュされたビットマップは生成されます。

cacheAsBitmapMatrix プロパティに適用するマトリックスの変形は、表示オブジェクトがビットマップキャッシュにレンダリングされるときに適用されます。そのため、2 倍のスケールを含む変形の場合、ビットマップのレンダリングはベクターのレンダリングの 2 倍のサイズになります。レンダリングでは、インバースの変形をキャッシュされたビットマップに適用して、最終の表示が同じになるようにします。キャッシュされたビットマップのサイズを縮小すると、レンダリングの正確性は損なわれる可能性がありますが、メモリの使用量を削減することができます。ビットマップのサイズを拡大すると、メモリの使用量は増えますが、レンダリングの品質が向上する場合があります。通常は、次の例で示すように、単位マトリックス（変形が適用されないマトリックス）を使用し、外観の変更は避けます。

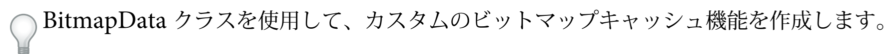
```
displayObject.cacheAsBitMap = true;
displayObject.cacheAsBitmapMatrix = new Matrix();
```

cacheAsBitmapMatrix プロパティを設定すると、ビットマップの再生成をトリガーすることなく、オブジェクトを拡大 / 縮小、傾斜、回転、移動できます。

また、アルファ値を 0 と 1 の範囲内で変更することもできます。カラー変換を含む transform.colorTransform プロパティを使用してアルファ値を変更する場合、変形オブジェクトで使用するアルファは 0 ～ 255 の範囲内である必要があります。他の方法でカラー変換を変更すると、キャッシュされたビットマップは再生成されます。

モバイルデバイス用に作成されたコンテンツで cacheAsBitmap を true に設定するときは、必ず cacheAsBitmapMatrix プロパティを設定します。ただし、次の潜在的な欠点について考慮してください。オブジェクトを回転、拡大 / 縮小または傾斜させた場合、通常のベクターレンダリングと比較して、最終的なレンダリング結果でビットマップが拡大 / 縮小したり、ギザギザが生じたりするおそれがあります。

手動によるビットマップキャッシュ


BitmapData クラスを使用して、カスタムのビットマップキャッシュ機能を作成します。

次の例は、表示オブジェクトのラスタライズされた単一のビットマップを再利用して、同じ BitmapData オブジェクトを参照します。各表示オブジェクトを拡大 / 縮小したときに、メモリ内にある元の BitmapData オブジェクトの更新や再描画は行われません。この手法は、CPU リソースを節約してアプリケーションの実行を高速化します。表示オブジェクトを拡大 / 縮小すると、含まれているビットマップが伸縮します。

更新された BitmapApple クラスを次に示します。

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super(buffer);

            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function deactivation(e:Event):void
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            alpha = Math.random();

            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if ( Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

アルファ値は引き続き各フレームで変更されます。次のコードでは、オリジナルのソースバッファーを各 BitmapApple インスタンスに渡しています。

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x,-bounds.y);

var buffer:BitmapData = new BitmapData(source.width+1, source.height+1, true, 0);
buffer.draw(source,mat);

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}
```

このテクニックでは、少量のメモリのみが使用されます。これは、キャッシュされた単一のビットマップのみがメモリで使用され、すべての BitmapApple インスタンスによって共有されるためです。また、BitmapApple インスタンスにアルファ、回転、拡大 / 縮小などの変更が加えられても、元のソースビットマップは更新されません。この手法を使用すると、パフォーマンスの低下を防ぐことができます。

最終ビットマップにスムージングを適用する場合は、smoothing プロパティを true に設定します。

```
public function BitmapApple(buffer:BitmapData)
{
    super (buffer);

    smoothing = true;

    addEventListener(Event.ADDED_TO_STAGE, activation);
    addEventListener(Event.REMOVED_FROM_STAGE, deactivation);
}
```

また、ステージの品質を調整すると、パフォーマンスが向上します。ラスタライズ処理を行う前に、ステージの品質を HIGH に設定し、処理が終了したら LOW に切り替えます。

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild ( holder );

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds ( source );

var mat:Matrix = new Matrix();
mat.translate ( -bounds.x, -bounds.y );

var buffer:BitmapData = new BitmapData ( source.width+1, source.height+1, true, 0 );

stage.quality = StageQuality.HIGH;

buffer.draw ( source, mat );

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++ )
{
    bitmapApple = new BitmapApple( buffer );

    holderVector[i] = bitmapApple;

    holder.addChild ( bitmapApple );
}
```

ベクターをビットマップに描画する前および後にステージの品質を切り替えることは、アンチエイリアスを適用したコンテンツを画面上で取得するための強力なテクニックです。このテクニックは、最終的なステージの品質に関わらず有効です。例えば、ステージの品質を LOW に設定していても、アンチエイリアスを適用したテキストを含む、アンチエイリアスを適用したビットマップを取得できます。この手法で、cacheAsBitmap プロパティは使用できません。この場合、ステージの品質を LOW に設定すると、ベクターの画質が更新されます。これにより、メモリ内のビットマップサーフェスと最終品質が更新されます。

動作の分離

可能な場合、単一のハンドラーで Event.ENTER_FRAME などのイベントを分離します。

Apple クラスの Event.ENTER_FRAME イベントを単一ハンドラーに分離することによって、コードをさらに最適化できます。このテクニックにより、CPU のリソースが節約されます。次の例は、この異なる手法を示しています。BitmapApple クラスは移動ビヘイビアーを処理しなくなります。

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super (buffer);

            smoothing = true;
        }
}
```

次のコードではりんごをインスタンス化し、その動きを単一のハンドラーで処理します。

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x,-bounds.y);

stage.quality = StageQuality.BEST;

var buffer:BitmapData = new BitmapData(source.width+1,source.height+1, true,0);
buffer.draw(source,mat);

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    bitmapApple.destinationX = Math.random()*stage.stageWidth;
    bitmapApple.destinationY = Math.random()*stage.stageHeight;

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}

stage.addEventListener(Event.ENTER_FRAME,onFrame);
```

```
var lng:int = holderVector.length

function onFrame(e:Event):void
{
        for (var i:int = 0; i < lng; i++)
        {
            bitmapApple = holderVector[i];
            bitmapApple.alpha = Math.random();

            bitmapApple.x -= (bitmapApple.x - bitmapApple.destinationX) *.5;
            bitmapApple.y -= (bitmapApple.y - bitmapApple.destinationY) *.5;

            if (Math.abs(bitmapApple.x - bitmapApple.destinationX ) < 1 &&
                Math.abs(bitmapApple.y - bitmapApple.destinationY ) < 1)
            {
                    bitmapApple.destinationX = Math.random()*stage.stageWidth;
                    bitmapApple.destinationY = Math.random()*stage.stageHeight;
            }
        }
}
```

その結果、200 個のハンドラーで各りんごの動きを処理する代わりに、単一の Event.ENTER_FRAME イベントで動きを処理します。アニメーション全体を簡単に一時停止できます。これはゲームで役立ちます。

例えば、単純なゲームでは次のハンドラーを使用できます。

```
stage.addEventListener(Event.ENTER_FRAME, updateGame);
function updateGame (e:Event):void
{
    gameEngine.update();
}
```

次の手順は、りんごをマウスまたはキーボードで操作できるようにすることです。これには、BitmapApple クラスを変更する必要があります。

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;

    public class BitmapApple extends Sprite
    {
        public var destinationX:Number;
        public var destinationY:Number;
        private var container:Sprite;
        private var containerBitmap:Bitmap;

        public function BitmapApple(buffer:BitmapData)
        {
            container = new Sprite();
            containerBitmap = new Bitmap(buffer);
            containerBitmap.smoothing = true;
            container.addChild(containerBitmap);
            addChild(container);
        }
}
```

その結果、従来の Sprite オブジェクトのような、インタラクティブな BitmapApple インスタンスが作成されます。ただし、インスタンスは単一のビットマップにリンクされているので、表示オブジェクトが変形されても再サンプルされません。

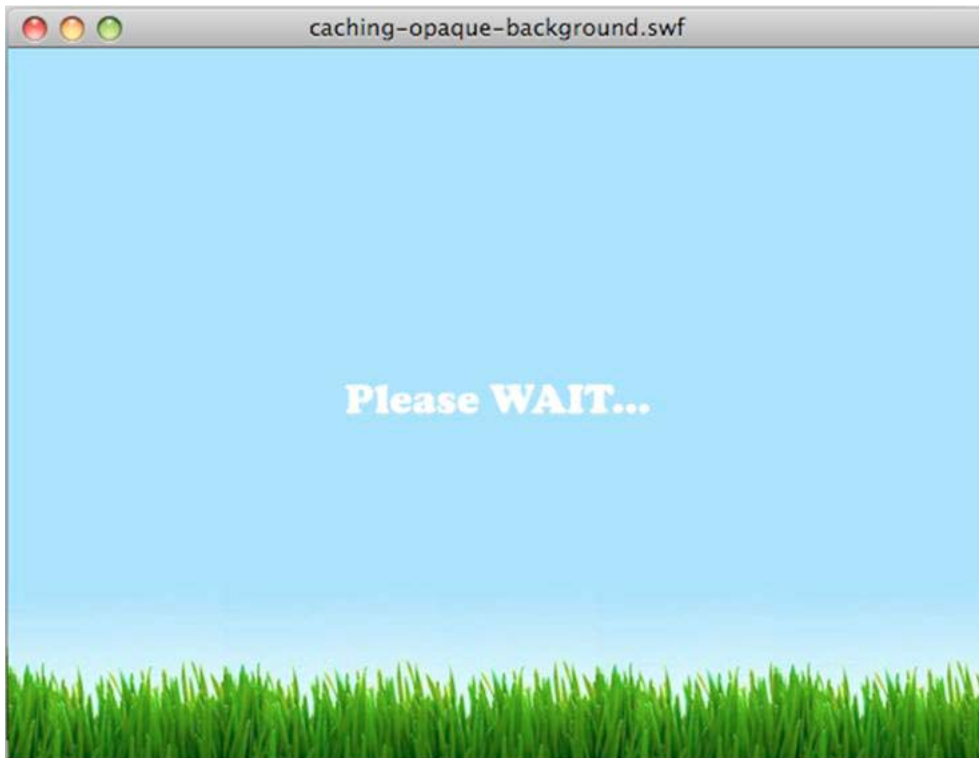
テキストオブジェクトのレンダリング

ビットマップキャッシュ機能と opaqueBackground プロパティを使用してテキストレンダリングのパフォーマンスを向上させます。

Flash Text Engine によって、重要な最適化が実現されます。ただし、1 行のテキストを表示するのに多数のクラスが必要です。したがって、TextLine クラスを使用して編集可能テキストフィールドを作成する場合は、大量のメモリを消費し、ActionScript コードを何行も記述する必要があります。TextLine クラスは、静的な編集不可のテキストで使用する場合に適しています。このようなテキストでは、高速なレンダリングが可能で、メモリをそれほど必要としません。

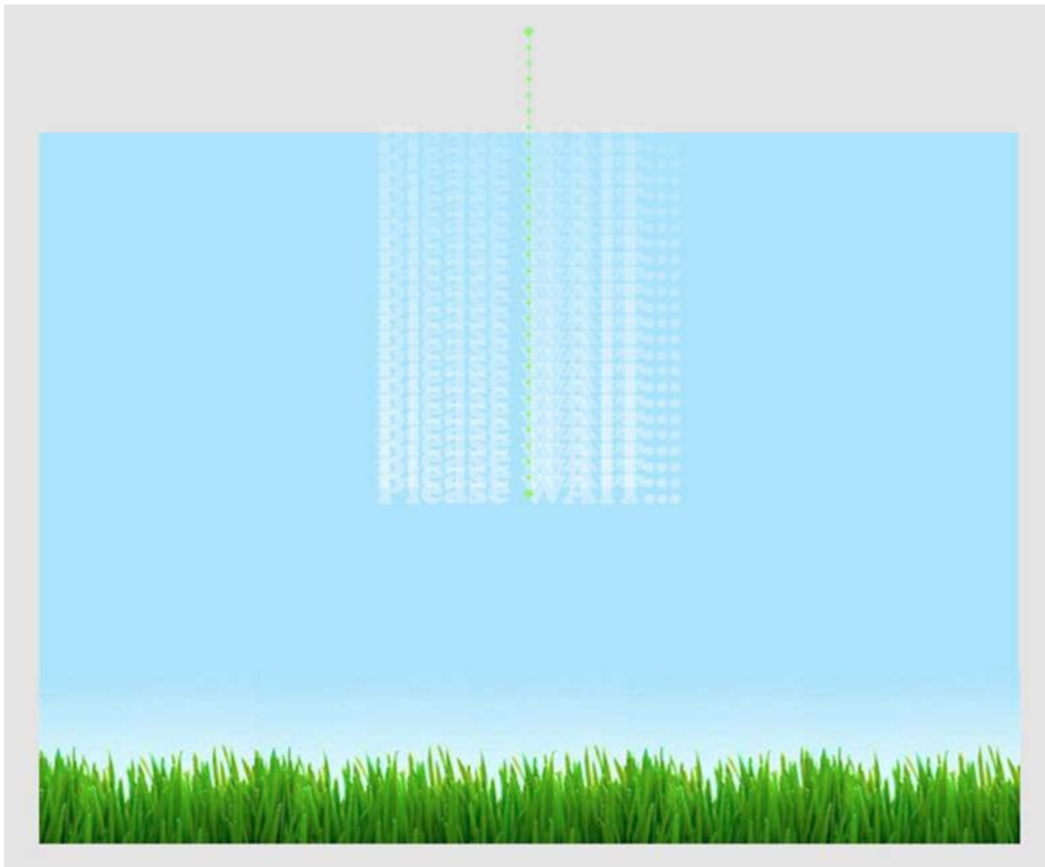
ビットマップキャッシュ機能を使用すると、ベクターコンテンツをビットマップとしてキャッシュできるので、レンダリングパフォーマンスが向上します。この機能は、複雑なベクターコンテンツで役に立ちます。また、高負荷のレンダリングが必要なテキストコンテンツにも使用できます。

次の例は、ビットマップキャッシュ機能と opaqueBackground プロパティを使用して、レンダリングパフォーマンスを向上させる方法を示しています。次の図は、一般的なスタートアップスクリーンを示しています。このスクリーンは、ユーザーがロードの待機中に表示されます。



スタートアップスクリーン

次の図は、プログラムによって TextField オブジェクトに適用されるイージング効果を示しています。テキストは、シーンの先頭から中央に向かってゆっくりと移動します。



テキストのイージング

次のコードでは、イージングを作成します。preloader 変数は現在のターゲットオブジェクトを保存し、パフォーマンスに影響する可能性のあるプロパティ参照を最小化します。

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

var destX:Number=stage.stageWidth/2;
var destY:Number=stage.stageHeight/2;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(preloader.y-destY)<1)
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
}
```

Math.abs() 関数をインラインでここに移動して、関数呼び出しの回数を減らし、さらにパフォーマンスを向上させることができます。ベストプラクティスでは、destX プロパティおよび destY プロパティに int 型を使用して、固定小数点値を格納します。int 型を使用すると、Math.ceil() または Math.round() のような時間のかかるメソッドを使用して手動で値の丸め込みを行うことなく、完全にピクセルへ吸着させることができます。このコードでは、座標を int に丸め込みません。これは、

値を常に丸め込むことで、オブジェクトがスムーズに移動しなくなるためです。オブジェクトの動きがぎこちなくなることがあります。これは、座標が各フレームで丸め込まれた最も近い整数に吸着されるためです。ただし、このテクニックは、表示オブジェクトの最終位置を設定するときに有効な場合があります。次のコードは使用しないでください。

```
// Do not use this code
var destX:Number = Math.round ( stage.stageWidth / 2 );
var destY:Number = Math.round ( stage.stageHeight / 2);
```

次のコードでは、処理が大幅に高速化されます。

```
var destX:int = stage.stageWidth / 2;
var destY:int = stage.stageHeight / 2;
```

上述のコードは、値の除算にビット単位のシフト演算子を使用すれば、さらに最適化できます。

```
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
```

ビットマップキャッシュ機能によって、ランタイムは動的ビットマップを使用して、オブジェクトをより簡単にレンダリングできます。現在の例では、TextField オブジェクトを含んでいるムービークリップがキャッシュされています。

```
wait_mc.cacheAsBitmap = true;
```

パフォーマンスを向上させる別の方法に、アルファ透明度を削除する方法があります。アルファ透明度を使用すると、透明なビットマップイメージの描画時に、ランタイムの負荷が増加します。上述のコードを参照してください。opaqueBackground プロパティを使用して、色を背景色として指定することにより、この問題を回避できます。

opaqueBackground プロパティを使用する場合、メモリ内に作成されているビットマップサーフェスは 32 ビットを使用し続けます。ただし、アルファのオフセットは 255 に設定され、透明度は使用されません。結果として、opaqueBackground プロパティを使用しても、メモリ使用量は低減しませんが、ビットマップキャッシュ機能を使用している場合は、レンダリングパフォーマンスが向上します。次のコードには、すべての最適化が含まれています。

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;

// Set the background to the color of the scene background
wait_mc.opaqueBackground = 0x8AD6FD;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;

function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
        e.currentTarget.removeEventListener ( Event.ENTER_FRAME, movePosition );
}
```

ここでは、アニメーションが最適化されています。また、透明度を削除することにより、ビットマップキャッシュが最適化されています。モバイルデバイスで、ビットマップキャッシュ機能を使用し、様々なアニメーションの状態でステージの品質の LOW と HIGH を切り替える場合について考えてみましょう。

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;
wait_mc.opaqueBackground = 0x8AD6FD;

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth>>1;
var destY:int = stage.stageHeight>>1;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(e.currentTarget.y-destY)<1)
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
    }
}
```

ただし、この場合にステージの品質を変更すると、ランタイムは現在のステージの品質に合わせて TextField オブジェクトのビットマップサーフェスを再生成します。したがって、ビットマップキャッシュ機能を使用するときは、ステージの品質を変更しないことをお勧めします。

ここでは、手動によるビットマップキャッシュ手法を使用してもよいでしょう。opaqueBackground プロパティをシミュレートするには、ムービークリップを不透明な BitmapData オブジェクトに描画します。これにより、ランタイムがビットマップサーフェスを再生成することはありません。

この手法は、長時間変更しないコンテンツに効果があります。ただし、テキストフィールドのコンテンツが変更可能な場合は、別の方法を検討してください。例えば、アプリケーションのロードの進捗率が継続的に更新されるテキストフィールドについて考えてみましょう。テキストフィールドまたは内部の表示オブジェクトがビットマップとしてキャッシュされている場合、コンテンツが変更されるたびにそのサーフェスを生成する必要があります。表示オブジェクトのコンテンツは継続的に変更されるので、手動によるビットマップキャッシュはここでは使用できません。このコンテンツ変更により、BitmapData.draw() メソッドを手動で呼び出して、キャッシュされたビットマップを更新する必要があります。

Flash Player 8（および AIR 1.0）以降では、ステージの品質の値に関わらず、レンダリング設定が「アンチエイリアス（読みやすさ優先)」に設定されているテキストフィールドは、完全にアンチエイリアス処理されます。この手法により、メモリ消費量は低減しますが、CPU 処理は増大します。また、ビットマップキャッシュ機能と比較して、レンダリングに多少時間がかかります。

次のコードでは、この手法を使用しています。

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;
function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener ( Event.ENTER_FRAME, movePosition );
    }
}
```

動いているテキストに「アンチエイリアス（読みやすさ優先）」オプションを使用することは推奨されません。テキストを拡大 / 縮小する際にこのオプションを使用すると、テキストは行揃えされた状態を維持しようとして、シフト効果が発生します。ただし、表示オブジェクトのコンテンツが常に変更されていて、テキストを拡大 / 縮小する必要がある場合は、品質を LOW に設定することにより、モバイルアプリケーションでのパフォーマンスが向上します。モーションが完了したら、品質を HIGH に戻します。

GPU

Flash Player アプリケーションでの GPU レンダリング

Flash Player 10.1 の重要な新機能は、GPU を使用してモバイルデバイス上にグラフィカルコンテンツをレンダリングできることです。これまでは、グラフィックは CPU のみを使用してレンダリングされていました。GPU を使用すると、フィルター、ビットマップ、ビデオおよびテキストのレンダリングが最適化されます。GPU によるレンダリングは、ソフトウェアによるレンダリングほど正確であるとは限りません。ハードウェアレンダラーを使用した場合、コンテンツが少し大きく表示されることがあります。さらに、Flash Player 10.1 には、オンスクリーンの Pixel Bender 効果がレンダリングされなくなる制限があります。これらの効果は、ハードウェアアクセラレーションを使用すると、黒い四角でレンダリングされます。

Flash Player 10 には GPU アクセラレーション機能が搭載されていたにもかかわらず、グラフィックの計算に GPU は使用されませんでした。すべてのグラフィックを画面に送るためだけに使用されました。Flash Player 10.1 では、グラフィックの計算にも GPU が使用されます。これにより、レンダリング速度が大幅に向上します。また、CPU の負荷が減ります。これは、モバイルデバイスなどリソースに制限のあるデバイスで有効です。

可能な限り最善のパフォーマンスを得るため、モバイルデバイスでコンテンツを実行するときは GPU モードが自動的に設定されます。GPU によるレンダリングを実行するために wmode を gpu に設定する必要はなくなりましたが、wmode を opaque または transparent に設定すると、GPU アクセラレーションが無効になります。

注意：デスクトップ上の Flash Player は、引き続き CPU を使用してソフトウェアレンダリングを実行します。ソフトウェアレンダリングが使用されるのは、デスクトップではドライバーが大きく異なり、ドライバーによってレンダリングの違いが目立つためです。また、デスクトップと一部のモバイルデバイスでは、レンダリングの違いがあることもあります。

モバイル AIR アプリケーションでの GPU レンダリング

AIR アプリケーションでハードウェアのグラフィックアクセラレーションを有効にするには、アプリケーション記述子に <renderMode>gpu</renderMode> を含めます。実行時にレンダリングモードを変更することはできません。デスクトップコンピューターでは、GPU グラフィックアクセラレーションがサポートされていないので、renderMode の設定は無視されます。

GPU レンダリングモードの制限事項

AIR 2.5 で GPU レンダリングモードを使用する場合、次の制限事項があります。

- GPU がオブジェクトをレンダリングできない場合、そのオブジェクトはまったく表示されません。CPU レンダリングに置き換えることはできません。
- ブレンドモードのレイヤー、アルファ、消去、オーバーレイ、ハードライト、比較（明)、比較（暗）はサポートされません。
- フィルターはサポートされません。
- PixelBender はサポートされません。
- 多くの GPU ユニットでは、最大テクスチャサイズは 1024 x 1024 です。ActionScript の場合、この値は、すべての変換の後で、最終的にレンダリングされる表示オブジェクトの最大サイズに変換されます。
- ビデオを再生する AIR アプリケーションで GPU レンダリングモードを使用することはお勧めしません。
- GPU レンダリングモードでは、仮想キーボードを開いたときに、テキストフィールドが必ず表示位置に移動されるとは限りません。ユーザーがテキストを入力するときにテキストフィールドが必ず表示されるようにするには、次のどちらかを実行します。テキストフィールドを画面の上半分に配置するか、または、フォーカスを受け取ったときにテキストフィールドを画面の上半分に移動します。
- GPU レンダリングモードが正常に動作しない一部のデバイスでは、このモードは無効です。最新情報については、AIR 開発者向けリリースノートを参照してください。

GPU レンダリングモードのベストプラクティス

より高速な GPU レンダリングを行うには、次のガイドラインに従います。

- ステージに表示するアイテムの数を制限してください。アイテムはそれぞれ、レンダリングおよび周囲の他のアイテムとの合成に時間がかかります。表示オブジェクトを表示しないようにする場合は、visible プロパティを false に設定します。· 単にステージ外に移動したり、ほかのオブジェクトの背面に隠したり、alpha プロパティを 0 に設定するだけの処理は行わないでください。表示オブジェクトが完全に不要になった場合は、removeChild() を使用してステージから削除します。
- オブジェクトを作成し、破棄するのではなく、オブジェクトを再利用してください。
- ビットマップは、2^n x 2^m ビットに近い、それ未満のサイズにしてください。サイズは正確に 2 の累乗とする必要はありませんが、2 の累乗を超えない範囲で、それに近いものである必要があります。例えば、31 x 15 ピクセルのイメージは、33 x 17 ピクセルのイメージよりも高速にレンダリングされます（31 と 15 は、2 の累乗である 32 と 16 の直前の整数です)。
- 可能であれば、Graphic.beginBitmapFill() メソッドを呼び出すときは、repeat パラメーターを false に設定してください。
- 描画を誇張しないでください。背景色は背景として使用し、大きいシェイプは互いに重ね合わせないでください。描画時には、ピクセルごとに負荷がかかります。
- 細長くとがった部分や自己交差しているエッジを含むシェイプ、エッジ部分が細密になっているシェイプの作成は避けてください。これらのシェイプは、スムーズなエッジを持つ表示オブジェクトよりもレンダリングに時間がかかります。
- 表示オブジェクトのサイズを制限してください。

- 頻繁にグラフィックが更新されない表示オブジェクトに対して cacheAsBitMap および cacheAsBitmapMatrix を有効にしてください。
- ActionScript 描画 API（Graphics クラス）を使用してグラフィックを作成することは避けてください。可能であれば、代わりにオーサリング時にそれらのオブジェクトを静的に作成します。
- ビットマップアセットをインポートする前に、最終的なサイズに拡大 / 縮小してください。

モバイルの AIR 2.0.3 での GPU レンダリングモード

GPU レンダリングは、Packager for iPhone を使用して作成されたモバイルの AIR アプリケーションではより制限されます。GPU は、cacheAsBitmap プロパティが設定されたビットマップ、塗りつぶしシェイプ、表示オブジェクトのみに影響します。cacheAsBitmap および cacheAsBitmapMatrix が設定されたオブジェクトに対しては、GPU は回転または拡大 / 縮小するオブジェクトを効率的にレンダリングできます。その他の表示オブジェクトに対して、GPU は連携して使用されますが、この場合はレンダリングパフォーマンスが低下します。

GPU レンダリングのパフォーマンスを最適化するためのヒント

GPU レンダリングによって SWF コンテンツのパフォーマンスを大幅に向上できますが、コンテンツのデザインが重要な役割を果たします。ソフトウェアレンダリングでこれまで問題なく機能していた設定が、GPU レンダリングではうまく機能しない場合があります。次のヒントにより、ソフトウェアレンダリングでのパフォーマンスを低下させずに、GPU レンダリングで良好なパフォーマンスを実現することができます。

注意：ハードウェアレンダリングをサポートするモバイルデバイスでは、多くの場合 Web から SWF コンテンツにアクセスします。したがって、ベストプラクティスとして、すべての画面で最適なエクスペリエンスを確保するために、SWF コンテンツの作成時に次のヒントについて考慮するようにしてください。

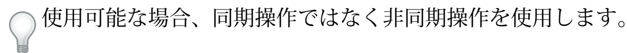
- HTML 埋め込みパラメーターで wmode=transparent または wmode=opaque の使用を避けます。これらのモードを使用すると、パフォーマンスが低下することがあります。また、ソフトウェアレンダリングとハードウェアレンダリングの両方において、オーディオとビデオの同期に小さなロスが生じることがあります。さらに、多くのプラットフォームではこれらのモードが有効な場合の GPU レンダリングがサポートされていないので、パフォーマンスが大幅に低下します。
- 通常モードまたはアルファのブレンドモードのみを使用します。他のブレンドモード、特にレイヤーブレンドモードの使用は避けてください。GPU を使用したレンダリングでは、一部のブレンドモードが正確に再現されない可能性があります。
- GPU レンダリングでは、ベクターは小さな三角形で構成されるメッシュに分解されてから、描画されます。このプロセスは、テッセレーションと呼ばれます。テッセレーションにはパフォーマンスの負荷が少しかかりますが、この負荷の量はシェイプが複雑になるほど増加します。パフォーマンスの影響を最小限に抑えるには、モーフィングシェイプの使用を避けてください。モーフィングシェイプでは、GPU レンダリングによってフレームごとにテッセレーションが実行されます。
- 自己交差曲線、非常に細い曲線領域（細い三日月など）およびシェイプのエッジに沿った複雑なディテールの使用を避けてください。これらのシェイプは、GPU でテッセレーションにより三角形のメッシュに変換するには複雑です。この理由を理解するため、500 x 500 の四角形と 100 x 10 の三日月という 2 つのベクターについて考えます。大きい四角形は 2 つの三角形となるだけなので、GPU で簡単にレンダリングできます。一方、三日月の曲線を描画するには、多くの三角形が必要です。このため、三日月シェイプに必要なピクセル数は少ないにもかかわらず、そのレンダリングはより複雑になります。
- 拡大 / 縮小の大幅な変更を避けてください。この変更でも、GPU によってグラフィックのテッセレーションが実行される場合があります。
- 過剰な描画をできる限り避けてください。過剰な描画によって、複数のグラフィックエレメントがレイヤー化され、その結果一方が他方を覆い隠すことになります。ソフトウェアレンダリングを使用すれば、各ピクセルはそれぞれ 1 回のみ描画されます。このため、ソフトウェアレンダリングの場合は、各ピクセル位置につき、どれほど多くのグラフィックエレメントが他のエレメントを覆い隠していても、アプリケーションのパフォーマンスに影響はありません。一方、ハード

ウェアレンダリングでは、ある領域を他のエレメントが覆っているかどうかにかかわらず、各エレメントのそれぞれのピクセルが描画されます。2 つの長方形が重なり合っている場合、ハードウェアレンダリングでは重なり合った領域が 2 回描画されますが、ソフトウェアレンダリングでは 1 回のみ描画されます。

このため、ソフトウェアレンダリングを使用するデスクトップでは、通常は過剰な描画によるパフォーマンスへの影響に気が付くことはありません。しかし、重なり合うシェイプが多いと、GPU レンダリングを使用するデバイス上でのパフォーマンスに悪影響を及ぼします。ベストプラクティスとしては、オブジェクトを非表示にするのではなく、表示リストから削除してください。

- 塗りつぶされた大きな長方形を背景として使用することを避けてください。代わりに、Stage の背景色を設定します。
- ビットマップの繰り返しに関するデフォルトのビットマップの塗りモードの使用をできる限り避けてください。パフォーマンスを向上するには、代わりにビットマップクランプモードを使用します。

非同期操作


使用可能な場合、同期操作ではなく非同期操作を使用します。

同期操作は、コードの指示直後に実行され、その操作が完了してから次の操作に移行されます。そのため、同期操作はフレームループのアプリケーションコードフェーズで実行されます。同期操作に長くかかる場合、フレームループのサイズが大きくなり、ディスプレイの表示がフリーズしたり、滑らかではなくなります。

非同期操作を実行するコードは即時に実行されるとは限りません。現在の実行スレッド内のコードと他のアプリケーションコードは、実行を続行します。非同期操作のコードは、レンダリングの問題を回避しながら、できるだけ早く実行されます。場合によっては、バックグラウンドで実行され、ランタイムフレームループの一部としてまったく実行されないこともあります。最終的に、操作が完了すると、ランタイムからイベントが送出され、そのイベントが以降の作業を実行することを監視できます。

レンダリングの問題を防ぐために、非同期操作はスケジュールが指定され、分割されます。このため、非同期の操作を使用するとアプリケーションの応答性を高めることが容易になります。詳しくは、2 ページの「認知パフォーマンスと実際のパフォーマンス」を参照してください。

ただし、非同期操作には若干のオーバーヘッドが伴います。特に、短時間で完了する操作については、非同期に実行すると余分に時間がかかる場合があります。

ランタイムでは、多くの操作が同期または非同期に固定されており、その実行方法は選択できません。ただし、Adobe AIR では、同期操作または非同期操作を選択できる 3 種類の操作があります。

- File クラスおよび FileStream クラスの操作

  File クラスの多くの操作は、同期または非同期で実行できます。例えば、ファイルやディレクトリのコピーまたは削除を行うメソッドや、ディレクトリの内容を列挙するメソッドには、いずれも非同期バージョンがあります。これらのメソッドには、非同期バージョンの名前に付いている「Async」という接尾辞が付いています。例えば、ファイルを非同期に削除するには、File.deleteFile() メソッドではなく File.deleteFileAsync() メソッドを呼び出します。

  ファイルの読み取りまたは書き込みに FileStream オブジェクトを使用する場合、FileStream オブジェクトを開く方法によって、操作を非同期に実行するかどうかが決まります。非同期操作には FileStream.openAsync() メソッドを使用します。データの書き込みは非同期に実行されます。データの読み取りはチャンク単位で実行されるので、同時に一部のデータを使用できます。対照的に、同期モードでは、FileStream オブジェクトはファイル全体を読み取ってから、コードの実行を続行します。

- ローカル SQL データベース操作

ローカル SQL データベースを操作する場合、SQLConnection オブジェクトを介して実行されるすべての操作は、同期モードまたは非同期モードで実行されます。非同期操作での実行を指定するには、SQLConnection.open() メソッドではなく SQLConnection.openAsync() メソッドを使用して、データベースに対する接続を開きます。データベース操作を非同期に実行すると、バックグラウンドで実行されます。データベースエンジンはランタイムフレームループ内で一切実行されないので、データベース操作によってレンダリングの問題が発生する可能性はほとんどありません。

ローカルの SQL データベースのパフォーマンスを改善するその他の方法については、82 ページの「SQL データベースのパフォーマンス」を参照してください。

- Pixel Bender スタンドアロンシェーダー

  ShaderJob クラスを使用すると、Pixel Bender シェーダーを介してイメージまたはデータセットを実行し、未処理の結果データにアクセスできます。デフォルトでは、ShaderJob.start() メソッドを呼び出すと、シェーダーは非同期に実行されます。実行は、ランタイムフレームループを使用せず、バックグラウンドで行われます。ShaderJob オブジェクトの同期実行を強制するには（これは推奨されません）、start() メソッドの最初のパラメーターに値 true を渡します。

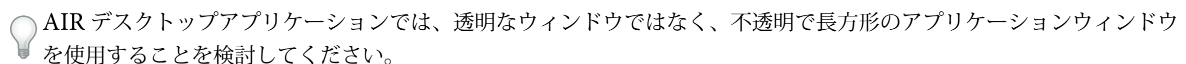
非同期にコードを実行するこのような組み込みのメカニズム以外に、同期ではなく非同期に実行する独自のコードを構築することもできます。実行時間が長くなる可能性があるタスクを実行するコードを作成する場合、複数のパートで実行するようにコードを構築できます。コードを複数のパートに分割すると、コードの実行ブロック間にレンダリング操作を実行できるようになるので、レンダリングの問題が発生する可能性が低くなります。

コードを分割する技術の一部を次に示します。これらすべての技術の背景には、常に作業の一部のみを実行するコードを記述するという考え方が中心にあります。コードの実行内容と、作業を停止する箇所を追跡します。Timer オブジェクトなどのメカニズムを使用して、作業が残っているかどうかを繰り返しチェックし、チャンク単位で追加の作業を完了するまで実行します。

このように作業を分割するコードを構築するには、いくつかの確立したパターンがあります。次の記事およびコードライブラリでは、こうしたパターンについて説明しています。また、アプリケーションに実装するときに役立つコードも用意されています。

- 「Asynchronous ActionScript Execution」（バックグラウンドの詳細といくつかの実装例が記載された Trevor McCauley による記事）
- 「Parsing & Rendering Lots of Data in Flash Player」（バックグラウンドの詳細と、「ビルダーパターン」および「グリーンスレッド」という 2 つの方法の例が記載された Jesse Warden による記事）
- 「Green Threads」（ソースコード例を使用して「グリーンスレッド」について説明した Drew Cummins による記事）
- 「greenthreads」（ActionScript で「グリーンスレッド」を実装するための Charlie Hubbard によるオープンソースコードライブラリ。詳しくは、「greenthreads Quick Start」を参照してください)
- 「Threads in ActionScript 3」（http://www.adobe.com/go/learn_fp_as3_threads_jp）（「疑似スレッド」技法の実装例が記載された Alex Harui による記事）

透明なウィンドウ


AIR デスクトップアプリケーションでは、透明なウィンドウではなく、不透明で長方形のアプリケーションウィンドウを使用することを検討してください。

AIR デスクトップアプリケーションの初期ウィンドウに不透明のウィンドウを使用するには、アプリケーション記述 XML ファイルで次の値を設定します。

```
<initialWindow>
    <transparent>false</transparent>
</initialWindow>
```

アプリケーションコードでウィンドウを作成する場合、false（デフォルト）に設定した transparent プロパティを持つ NativeWindowInitOptions オブジェクトを作成します。それを NativeWindow コンストラクターに渡し、さらに NativeWindow オブジェクトを作成します。

```
// NativeWindow: flash.display.NativeWindow class

var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
initOptions.transparent = false;
var win:NativeWindow = new NativeWindow(initOptions);
```

Flex Window コンポーネントの場合、コンポーネントの transparent プロパティが false （デフォルト）に設定されていることを確認してから、Window オブジェクトの open() メソッドを呼び出します。

```
// Flex window component: spark.components.Window class

var win:Window = new Window();
win.transparent = false;
win.open();
```

透明なウィンドウには、アプリケーションウィンドウの背後にあるユーザーのデスクトップや他のアプリケーションの一部が表示される場合があります。そのため、透明なウィンドウをレンダリングするには、より多くのリソースが必要になります。長方形の透明ではないウィンドウでは、オペレーティングシステムのクロムまたはカスタムクロムのどちらを使用していても、同じレンダリングの負荷になりません。

長方形ではない表示をすること、またはアプリケーションウィンドウを通してバックグラウンドコンテンツを表示することが重要な場合にのみ、透明なウィンドウを使用します。

ベクターシェイプのスムージング

シェイプのスムージングによってレンダリングのパフォーマンスが向上します。

ビットマップとは異なり、ベクターコンテンツのレンダリングには多くの演算が必要です。これは特に、多くの制御ポイントを含むグラデーションや複雑なパスを使用する場合に当てはまります。デザイナーや開発者は、シェイプを十分に最適化する必要があります。次の図は、制御ポイントを多く含む単純化されていないパスを示しています。

最適化されていないパス

Flash Professional のスムーズツールを使用して、余分な制御ポイントを削除できます。同様の機能を持つツールを Adobe® Illustrator® で利用できます。また、ポイントおよびパスの総数は文書情報パネルで確認できます。

スムージングによって余分な制御ポイントが削除され、SWF ファイルの最終的なサイズが減少し、レンダリングのパフォーマンスが向上します。次の図では、同じパスのスムージング後の状態を示しています。

最適化されたパス

パスを単純化し過ぎない限り、この最適化によって見た目が変更されることはありません。しかし、複雑なパスを単純化することで、最終的なアプリケーションの平均フレームレートを格段に向上することができます。

第 6 章：ネットワーク通信の最適化

ネットワーク通信の強化機能

Flash Player 10.1 および AIR 2.5 には、すべてのプラットフォームを対象に、ネットワークを最適化する一連の新機能（循環バッファー、スマートシークなど）が導入されています。

循環バッファー

モバイルデバイスにメディアコンテンツをロードする際、デスクトップコンピューターではほとんど発生しない問題に遭遇する可能性があります。例えば、ディスク容量不足やメモリ不足が発生しやすくなります。デスクトップ用の Flash Player 10.1 および AIR 2.5 では、ビデオのロード時に FLV ファイル（または MP4 ファイル）全体をハードドライブにダウンロードしてキャッシュします。その後、ランタイムはこのキャッシュファイルからビデオを再生します。ディスク容量が不足することは滅多にありません。容量不足が発生すると、デスクトップランタイムはビデオの再生を停止します。

モバイルデバイスでは、ディスク容量が容易に不足する可能性があります。モバイルデバイスのディスク容量が不足しても、デスクトップランタイムとは異なり、モバイルランタイムが再生を停止することはありません。その代わり、モバイルランタイムはキャッシュファイルの再利用を開始して、ファイルの先頭から再書き込みを行います。これにより、ユーザーはビデオを視聴し続けることができます。再書き込みが行われたビデオの領域はシークできません。ただし、ファイルの先頭へのシークは可能です。循環バッファーはデフォルトでは開始されません。再生中、または、ムービーがディスク容量または RAM のサイズを超える場合は再生の最初から開始することができます。ランタイムで循環バッファーを使用するには、4 MB 以上の RAM または 20 MB 以上のディスク容量が必要です。

注意：デバイスに十分なディスク容量がある場合、モバイル用のランタイムはデスクトップと同じように動作します。デバイスにディスクが実装されていないかディスクが一杯になっている場合は、RAM のバッファーが代替利用されることに注意してください。キャッシュファイルと RAM バッファーのサイズ制限は、コンパイル時に設定できます。一部の MP4 ファイルは、構造上、再生を開始する前にファイル全体がダウンロードされている必要があります。ランタイムはこのようなファイルを検出し、十分なディスク容量がない場合はダウンロードを行いません。したがって、MP4 ファイルは再生できません。最善策は、このようなファイルのダウンロードを要求しないことだと言えます。

開発者は、キャッシュされたストリームの境界内でのみシークが機能することに注意する必要があります。オフセットが範囲外の場合、NetStream.seek() は失敗することがあります。このとき、NetStream.Seek.InvalidTime イベントが送出されます。

スマートシーク

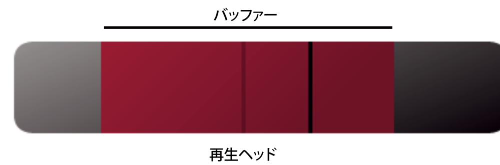
注意：スマートシーク機能には、Adobe® Flash® Media Server 3.5.3 が必要です。

Flash Player 10.1 および AIR 2.5 には、スマートシークと呼ばれる新機能が導入されています。この機能により、ストリーミング配信のビデオを再生する際のユーザーエクスペリエンスが向上します。ユーザーがバッファー境界内をシークする場合、ランタイムはバッファーを再利用して高速なシーク機能を提供します。以前のバージョンのランタイムでは、バッファーは再利用されませんでした。例えば、バッファー時間（NetStream.bufferTime）が 20 秒に設定されている状態でストリーミングサーバーからビデオを再生中に、ユーザーが 10 秒先をシークすると、ランタイムはロード済みの 10 秒間を再利用せずに、すべてのバッファーデータを破棄していました。この動作では、ランタイムがサーバーに新しいデータを要求する頻度が増加し、低速接続環境では再生パフォーマンスが低下します。

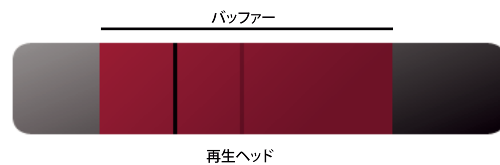
次の図は、前のリリースのランタイムでのバッファーの動作を示しています。bufferTime プロパティでは、バッファーにプリロードする秒数を指定します。接続が切断された場合、この期間はビデオを停止せずにバッファーを使用できます。

スマートシーク機能導入前のバッファーの動作

スマートシーク機能の導入により、ユーザーがビデオをスクラブしたときに、バッファーを使った瞬時の後方シークまたは前方シークが可能になりました。次の図は、この新しい動作を示しています。



スマートシーク機能を使用した前方シーク



スマートシーク機能を使用した後方シーク

スマートシーク機能では、ユーザーが前方または後方シークを実行したときにバッファーを再利用するので、より高速でスムーズな再生エクスペリエンスを実現できます。この新機能のメリットは、ビデオ配信者側の帯域幅を節約できることです。ただし、バッファー境界外でシークを実行すると、通常の動作が行われ、ランタイムはサーバーに新しいデータを要求します。

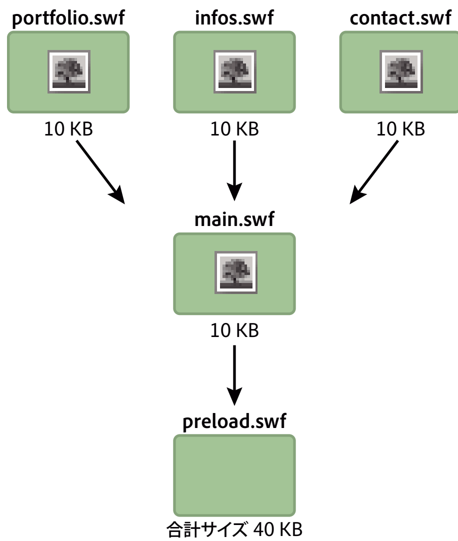
注意：この動作は、プログレッシブビデオのダウンロードには適用されません。

スマートシークを使用するには、NetStream.inBufferSeek を true に設定します。

外部コンテンツ

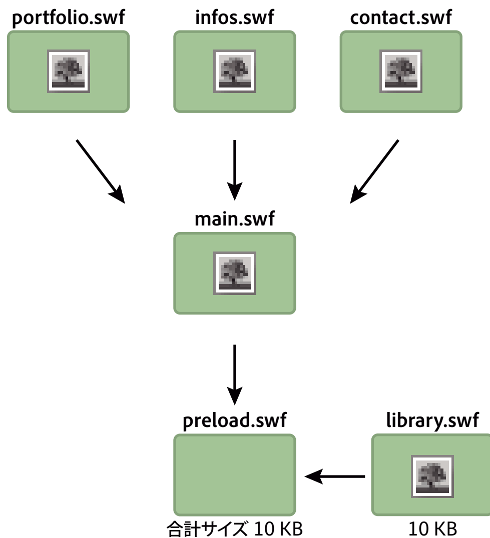
アプリケーションを複数の SWF ファイルに分割してください。

モバイルデバイスは、ネットワークへのアクセスが制限されている場合があります。コンテンツのロードを高速化するには、アプリケーションを複数の SWF ファイルに分割します。コードのロジックやアセットは、アプリケーション全体で再利用するようにします。例えば、複数の SWF ファイルに分割されているアプリケーションについて考えてみましょう。次の図を参照してください。



複数の SWF ファイルに分割されたアプリケーション

この例では、各 SWF ファイルに同じビットマップのコピーが含まれています。ランタイム共有ライブラリを使用すれば、このようなファイルの重複を防ぐことができます。次の図を参照してください。



ランタイム共有ライブラリの使用

この方法を使用すれば、ランタイム共有ライブラリがロードされ、ビットマップを他の SWF ファイルでも使用できるようになります。ApplicationDomain クラスには、ロードされたすべてのクラス定義が格納され、実行時に getDefinition() メソッドを通して使用できるようになります。

ランタイム共有ライブラリにはすべてのコードロジックも含まれています。実行時には、再コンパイル不要でアプリケーション全体を更新できます。次のコードは、実行時にランタイム共有ライブラリをロードし、SWF ファイルに含まれている定義を抽出します。この方法は、フォント、ビットマップ、サウンド、またはすべての ActionScript クラスで使用できます。

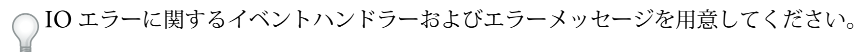
```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load(new URLRequest("library.swf") );
var classDefinition:String = "Logo";

function loadingComplete(e:Event ):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the loaded SWF file application domain
    var appDomain:ApplicationDomain = objectLoaderInfo.applicationDomain;

    // Check whether the definition is available
    if ( appDomain.hasDefinition(classDefinition) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate logo
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " +  classDefinition + " is not available.");
}
```

ロードする SWF ファイルのアプリケーションドメイン内にクラス定義をロードすることにより、より簡単に定義を取得できるようになります。

```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load ( new URLRequest ("rsl.swf"), new LoaderContext ( false, ApplicationDomain.currentDomain) );
var classDefinition:String = "Logo";

function loadingComplete ( e:Event ):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the current SWF file application domain
    var appDomain:ApplicationDomain = ApplicationDomain.currentDomain;

    // Check whether the definition is available
    if (appDomain.hasDefinition( classDefinition ) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate it
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " +  classDefinition + " is not available.");
}
```

これで、ロード済みの SWF ファイルに含まれているクラスは、現在のアプリケーションドメインで getDefinition() メソッドを呼び出すことにより使用できます。getDefinitionByName() メソッドを呼び出しても、これらのクラスにアクセスできます。この方法では、フォントおよびサイズの大きいアセットを一度ロードするだけで済むので、帯域幅を節約できます。アセットが他の SWF ファイルに書き出されることはありません。唯一の制約は、loader.swf ファイルを使用してアプリケーションをテストおよび実行する必要があるという点です。このファイルは、最初にアセットをロードしてから、アプリケーションを構成している別の SWF ファイルをロードします。

入出力エラー


IO エラーに関するイベントハンドラーおよびエラーメッセージを用意してください。

モバイルデバイスは、高速インターネットに接続されているデスクトップコンピューターと比較して、ネットワークの信頼性に劣ります。モバイルデバイスで外部コンテンツへアクセスする場合、利用可能性と速度という 2 つの制約が伴います。したがって、アセットを軽量化し、すべての IO_ERROR イベントにハンドラーを追加して、ユーザーにフィードバックを提供するようにしてください。

例えば、モバイルデバイスで Web サイトを閲覧しているユーザーのネットワーク接続が、地下鉄の駅と駅の間で突然切断されたとします。接続が切断されたときに、動的アセットをロード中でした。デスクトップでは、このシナリオはほとんど発生しないので、空のイベントリスナーを使用して、ランタイムエラーが表示されないようにします。一方、モバイルデバイスでは、単純な空のリスナーよりも複雑な方法で、この状況に対応する必要があります。

次のコードでは、IO エラーに応答しません。このコードをそのまま使用しないでください。

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener( Event.COMPLETE, onComplete );
addChild( loader );
loader.load( new URLRequest ("asset.swf" ) );

function onComplete( e:Event ):void
{
    var loader:Loader = e.currentTarget.loader;
    loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
    loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}
```

より適切に対応するには、このようなエラーを処理して、ユーザーにエラーメッセージを表示します。次のコードでは、エラーを適切に処理しています。

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );
loader.contentLoaderInfo.addEventListener ( IOErrorEvent.IO_ERROR, onIOError );
addChild ( loader );
loader.load ( new URLRequest ("asset.swf" ) );

function onComplete ( e:Event ):void
{
      var loader:Loader = e.currentTarget.loader;
      loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
      loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}

function onIOError ( e:IOErrorEvent ):void
{
      // Show a message explaining the situation and try to reload the asset.
      // If it fails again, ask the user to retry when the connection will be restored
}
```

ベストプラクティスは、ユーザーにコンテンツを再ロードする方法を提供することです。この動作は、onIOError() ハンドラーで実装できます。

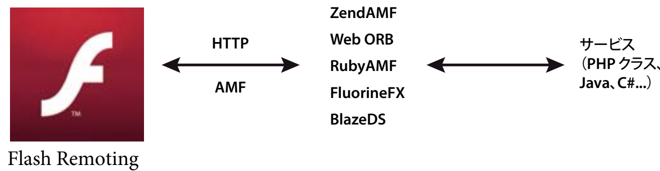
Flash Remoting

クライアントとサーバーの最適化されたデータ通信のためには、Flash Remoting および AMF を使用します。

リモートコンテンツを SWF ファイルにロードするのに XML を使用することができます。ただし、XML はランタイムがロードおよび解析するプレーンテキストです。XML は限られた量のコンテンツをロードするアプリケーションに適しています。大容量のコンテンツをロードするアプリケーションを開発している場合は、Flash Remoting テクノロジーと Action Message Format（AMF）の使用を検討してください。

AMF は、サーバーとランタイム間でのデータ共有に使用されるバイナリ形式です。AMF を使用すると、データのサイズを縮小して、転送速度を改善できます。AMF はランタイムのネイティブフォーマットなので、AMF データをランタイムに送信すると、メモリを集中的に使用するシリアル化および非シリアル化をクライアント側で避けることができます。これらのタスクはリモートゲートウェイが処理します。ActionScript データタイプをサーバーに送信すると、リモートゲートウェイがシリアライゼーションをサーバーサイドで処理します。また、ゲートウェイは対応するデータタイプをユーザーに送信します。このデータタイプはサーバー上で作成されたクラスです。ランタイムから呼び出すことのできる一連のメソッドが公開されています。Flash Remoting ゲートウェイには、ZendAMF、FluorineFX、WebORB および BlazeDS（アドビ システムズ社が提供している公式オープンソースの Java Flash Remoting ゲートウェイ）があります。

次の図は、Flash Remoting の概念図です。



Flash Remoting

次の例では、NetConnection クラスを使用して Flash Remoting ゲートウェイに接続します。

```
// Create the NetConnection object
var connection:NetConnection = new NetConnection ();

// Connect to a Flash Remoting gateway
connection.connect ("http://www.yourserver.com/remotingservice/gateway.php");

// Asynchronous handlers for incoming data and errors
function success ( incomingData:* ):void
{
    trace( incomingData );
}

function error ( error:* ):void
{
    trace( "Error occured" );
}

// Create an object that handles the mapping to success and error handlers
var serverResult:Responder = new Responder (success, error);

// Call the remote method
connection.call ("org.yourserver.HelloWorld.sayHello", serverResult, "Hello there ?");
```

リモートゲートウェイへの接続は簡単です。Adobe® Flex® SDK に含まれている RemoteObject クラスを使用すれば、Flash Remoting をさらに簡単にできます。

注意：Adobe® Flash® Professional プロジェクト内では、外部の SWC ファイル（Flex フレームワークのファイルなど）を使用できます。SWC ファイルを使用すれば、RemoteObject クラスとその依存クラスを使用できます。他の Flex SDK は必要ありません。上級開発者は、必要に応じて、未加工型の Socket クラス経由でリモートゲートウェイと直接通信できます。

不要なネットワーク操作

アセットが必要になるたびにネットワークからロードするのではなく、ロードしたアセットをローカルにキャッシュしておきます。

アプリケーションでメディアまたはデータなどのアセットをロードする場合、ローカルデバイスに保存することでアセットをキャッシュします。変更頻度が低いアセットの場合、変更の間にキャッシュを更新する方法があります。例えば、1 日に 1 度、イメージファイルの新しいバージョンをチェックする場合や、2 時間に 1 度、更新データをチェックする場合です。

アセットのキャッシュ方法はいくつかあり、アセットの種類と特性によって異なります。

- イメージやビデオなどのメディアアセット：File クラスおよび FileStream クラスを使用してファイルシステムにファイルを保存します。

- 個々のデータ値または小さなデータセット：SharedObject クラスを使用してローカルの共有オブジェクトとして値を保存します。
- 大きなデータセット：ローカルデータベースにデータを保存するか、データをシリアル化してファイルに保存します。

データ値のキャッシュについては、オープンソースの AS3CoreLib プロジェクトを参照してください。ロードとキャッシュを行う ResourceCache クラスが含まれています。

第 7 章：メディアの操作

ビデオ

モバイルデバイス上のビデオパフォーマンスの最適化について詳しくは、Adobe Developer Connection Web サイトの「Optimize web content for mobile delivery」を参照してください。

特に、次の節を参照してください。

- **Playing video on mobile devices**
- **Code samples**

これらの節には、モバイルデバイス用のビデオプレーヤーの開発に関して、次のような情報が記載されています。

- ビデオエンコーディングのガイドライン
- ベストプラクティス
- ビデオプレーヤーのパフォーマンスのプロファイル設定方法
- ビデオプレーヤー実装のリファレンス

StageVideo

StageVideo クラスを使用し、ハードウェアアクセラレーションを活用してビデオを表示します。

StageVideo オブジェクトの使用について詳しくは、『ActionScript 3.0 開発ガイド』の「ハードウェアアクセラレーションによる表示のための StageVideo クラスの使用」を参照してください。

オーディオ

Flash Player 9.0.115.0 および AIR 1.0 以降では、AAC ファイル（AAC Main、AAC LC、SBR）を再生できます。mp3 ファイルの代わりに AAC ファイルを使用すると、簡単な最適化を実行できます。AAC フォーマットは、mp3 フォーマットと同等のビットレートにおいて mp3 よりも高品質であり、ファイルサイズも小さくなります。ファイル サイズを減らすと、帯域幅が節約されます。これは、高速インターネット接続を行うことができないモバイルデバイスでは重要な要素です。

オーディオのハードウェアデコード

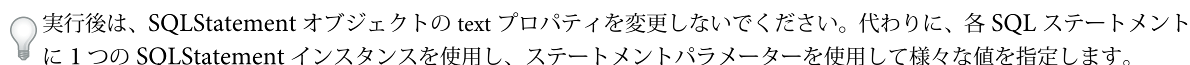
オーディオデコードは、ビデオデコードと同様に高い CPU サイクルを必要とし、デバイスで利用可能なハードウェアを活用することで最適化できます。Flash Player 10.1 および AIR 2.5 はオーディオのハードウェアドライバーを検出および使用して、AAC ファイル（LC、HE/SBR プロファイル）または mp3 ファイル（PCM はサポートされません）のデコード時にパフォーマンスを向上させることができます。CPU 使用率が大幅に減り、これによりバッテリー使用率も減ります。また、CPU を他の操作に利用できるようになります。

注意：AAC フォーマットを使用する場合、AAC Main プロファイルはデバイスでサポートされません。これは、ほとんどのデバイスでハードウェアサポートが行われていないためです。

オーディオのハードウェアデコードはユーザーおよび開発者にとって透過的です。ランタイムがオーディオストリームの再生を開始するときは、ビデオの場合と同様に、まずハードウェアをチェックします。ハードウェアドライバーが利用でき、オーディオ形式がサポートされている場合は、オーディオのハードウェアデコードが実行されます。ただし、着信 AAC または mp3 ストリームでハードウェアを使用してコードを処理できる場合でも、ハードウェアですべての効果を処理できないことがあります。例えば、ハードウェアの制限によっては、オーディオミキシングと再サンプリングを処理できないことがあります。

第 8 章：SQL データベースのパフォーマンス

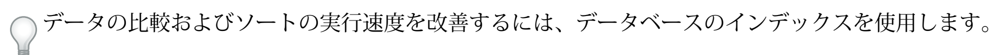
データベースのパフォーマンスためのアプリケーションデザイン


実行後は、SQLStatement オブジェクトの text プロパティを変更しないでください。代わりに、各 SQL ステートメントに 1 つの SQLStatement インスタンスを使用し、ステートメントパラメーターを使用して様々な値を指定します。

SQL ステートメントは、実行前に準備（コンパイル）されて、そのステートメントを実行するために内部で実行されるステップが特定されます。以前に実行されていない SQLStatement インスタンスに対して SQLStatement.execute() を呼び出すと、ステートメントが実行される前に自動的に準備されます。その後の execute() メソッドの呼び出しでは、SQLStatement.text プロパティが変更されていない限り、ステートメントは準備された状態のままです。したがって、より高速に実行されます。

ステートメントの再利用のメリットを最大限に活用するために、実行のたびに値を変更する場合は、ステートメントパラメーターを使用してステートメントをカスタマイズします（ステートメントパラメーターを指定するには SQLStatement.parameters 連想配列プロパティを使用します）。ステートメントパラメーターの値を変更しても、SQLStatement インスタンスの text プロパティを変更した場合とは違って、ステートメントを準備し直す必要はありません。

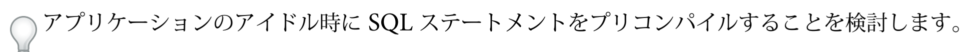
SQLStatement インスタンスを再利用する場合は、準備した SQLStatement インスタンスへの参照をアプリケーションで保存する必要があります。インスタンスへの参照を保持するには、その変数を関数スコープの変数ではなくクラススコープの変数として宣言します。SQLStatement をクラススコープの変数にするためには、SQL ステートメントが 1 つのクラスにラップされるようにアプリケーションを構成することをお勧めします。一緒に実行される一連のステートメントを 1 つのクラスにラップすることもできます（この方法は、コマンドデザインパターンの使用と呼ばれます）。インスタンスをクラスのメンバー変数として定義すると、そのインスタンスは、そのラッパークラスのインスタンスがアプリケーション内に存在する限り保持されます。最低でも、SQLStatement インスタンスを含む変数を関数の外部で定義すれば、そのインスタンスがメモリ内に保持されるようになります。例えば、SQLStatement インスタンスを ActionScript クラスのメンバ変数または JavaScript ファイルの関数以外の変数として宣言し、実際にクエリを実行する必要が生じたら、ステートメントのパラメーター値を設定して execute() メソッドを呼び出します。


データの比較およびソートの実行速度を改善するには、データベースのインデックスを使用します。

列にインデックスを作成すると、その列データのコピーがデータベースに格納されます。このコピーは、数字順またはアルファベット順で常にソートされます。これを使用して、値の照合処理（等号演算子を使用する場合など）および ORDER BY 句による結果データのソート処理が高速に実行されます。

データベースインデックスが最新の状態に維持されるので、テーブルでの変更操作（INSERT または UPDATE）が少し遅くなります。ただし、データ取得速度については大幅な向上効果が見込まれます。このパフォーマンスのトレードオフがあるので、すべてのテーブルのすべての列に無条件にインデックスを設定することは避け、インデックスの定義に関する基準を定めてください。次のガイドラインを使用して、インデックスの定義方法を計画してください。

- 結合テーブル、WHERE 句、または ORDER BY 句で使用されるインデックス列。
- 複数の列が同時に、また頻繁に使用される場合、単一のインデックスでそれらの列にインデックスを定義します。
- アルファベット順でソートされているテキストデータを含む列を取得する場合、インデックスには COLLATE NOCASE 照合を指定します。


アプリケーションのアイドル時に SQL ステートメントをプリコンパイルすることを検討します。

SQL ステートメントを初めて実行するときは時間がかかります。これは、データベースエンジンによって SQL テキストが準備（コンパイル）されるからです。ステートメントを準備して実行する操作は負荷が高くなる可能性があるので、初期データを事前に読み込んでおき、他のステートメントはバックグラウンドで実行するという方法が考えられます。

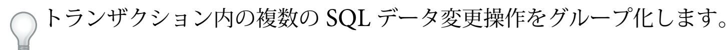
1 まず、アプリケーションで最初に必要になるデータをロードします。

2 アプリケーションの最初の起動操作が完了したら（またはアプリケーションのその他の「アイドル」時に）、他のステートメントを実行します。

例えば、初期画面を表示するためにアプリケーションからデータベースへ一切アクセスしないとします。この場合、画面の表示を待ってから、データベース接続を開きます。最後に、SQLStatement インスタンスを作成し、その他の必要な処理を実行します。

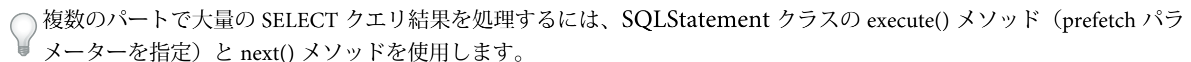
または、アプリケーションで起動後すぐに何らかのデータ（特定のクエリの結果など）を表示する場合があるとします。そのクエリの SQLStatement インスタンスをすぐに実行します。初期データが読み込まれて表示されたら、他のデータベース操作のための SQLStatement インスタンスを作成し、可能であれば、後に必要になる他のステートメントを実行します。

実際には、SQLStatement インスタンスを再使用している場合、ステートメントの準備に必要な追加の時間にかかるコストは、1 回分のコストだけです。そのため、全体的なパフォーマンスに与える影響は大きくないと考えられます。


トランザクション内の複数の SQL データ変更操作をグループ化します。

例えば、データの追加や変更を伴う SQL ステートメント（INSERT ステートメントや UPDATE ステートメント）を多数実行するとします。すべてのステートメントを明示的なトランザクションの中で実行するとパフォーマンスが大幅に向上します。トランザクションを明示的に開始しない場合は、各ステートメントが、自動的に作成される固有のトランザクションで実行されます。この場合、トランザクション（ステートメント）の実行が完了するたびに結果のデータがディスク上のデータベースファイルに書き込まれます。

一方、トランザクションを明示的に作成してそのトランザクションのコンテキストでステートメントを実行する場合は、すべての変更がメモリ内で行われ、トランザクションがコミットされるときに一度にデータベースファイルに書き込まれます。一般に、ディスクへのデータの書き込みは操作の中で最も時間のかかる部分です。したがって、ディスクへの書き込みを SQL ステートメントごとに行う代わりに一度に行うようにすると、パフォーマンスが大幅に向上します。


複数のパートで大量の SELECT クエリ結果を処理するには、SQLStatement クラスの execute() メソッド（prefetch パラメーターを指定）と next() メソッドを使用します。

例えば、大量の結果セットを取得する SQL ステートメントを実行するとします。この場合、各行のデータはループで処理されます。例えば、データをフォーマットするか、そこからオブジェクトを作成します。そのデータの処理には長時間かかる可能性があります。また、それによって画面がフリーズしたり応答しないなどのレンダリングの問題が発生する可能性があります。68 ページの「非同期操作」で説明したように、解決方法の 1 つは作業を複数のチャンクに分割することです。SQL データベース API で、データ処理の分割は簡単に実行できます。

SQLStatement クラスの execute() メソッドには、オプションの prefetch パラメーター（第 1 パラメーター）があります。値を指定する場合、実行が完了したときにデータベースが返す結果行の最大数を指定します。

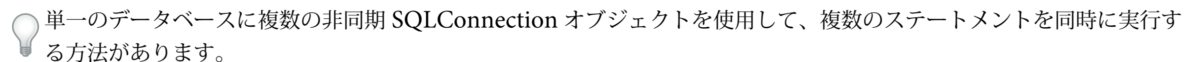
```
dbStatement.addEventListener(SQLEvent.RESULT, resultHandler);
dbStatement.execute(100); // 100 rows maximum returned in the first set
```

結果データの最初のセットが返されると、next() メソッドを呼び出してステートメントの実行を続行し、別の結果行を取得できます。execute() メソッドと同様に、next() メソッドでは prefetch パラメーターを使用して、返す行の最大数を指定できます。

```
// This method is called when the execute() or next() method completes
function resultHandler(event:SQLEvent):void
{
    var result:SQLResult = dbStatement.getResult();
    if (result != null)
    {
        var numRows:int = result.data.length;
        for (var i:int = 0; i < numRows; i++)
        {
            // Process the result data
        }

        if (!result.complete)
        {
            dbStatement.next(100);
        }
    }
}
```

すべてのデータがロードされるまで、next() メソッドを呼び出し続けることができます。前の一覧で示したように、データがすべてロードされたときを判断できます。execute() メソッドまたは next() メソッドが完了するたびに作成される SQLResult オブジェクトの complete プロパティを確認します。

注意：prefetch パラメーターおよび next() メソッドを使用して、結果データの処理を分割します。このパラメーターとメソッドは、クエリの結果を結果セットの一部のみに限定する目的では使用しないでください。ステートメントの結果セットから行のサブセットを取得する場合は、SELECT ステートメントの LIMIT 句を使用します。その結果セットが大きい場合にも、prefetch パラメーターおよび next() メソッドを使用して、結果の処理を分割できます。


単一のデータベースに複数の非同期 SQLConnection オブジェクトを使用して、複数のステートメントを同時に実行する方法があります。

SQLConnection オブジェクトが openAsync() メソッドを使用してデータベースに接続されている場合、メインのランタイム実行スレッドではなく、バックグラウンドで実行されます。さらに、各 SQLConnection は独自のバックグラウンドスレッドで実行されます。複数の SQLConnection オブジェクトを使用すると、複数の SQL ステートメントを実質的に並列して実行できます。

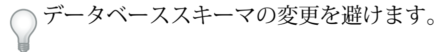
この方法には潜在的な弱点もあります。最も重要な点は、追加の各 SQLStatement オブジェクトに追加のメモリが必要なことです。さらに、同時実行によってプロセッサーの処理が増えます。CPU または CPU コアが 1 つしかないコンピューターの場合は特に顕著です。このような問題点があるため、この方法をモバイルデバイスで使用することは推奨されません。

もう 1 つの問題点は、SQLStatement オブジェクトが単一の SQLConnection オブジェクトにリンクしていることで、SQLStatement オブジェクトの再利用による利点が失われる可能性があることです。そのため、関連する SQLConnection が使用中の場合、SQLStatement オブジェクトは再利用できません。

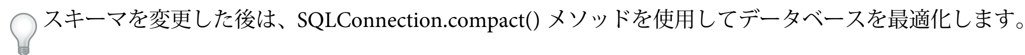
単一のデータベースに接続されている複数の SQLConnection オブジェクトを使用する場合、各オブジェクトが個々のトランザクションでステートメントを実行するように注意してください。データの追加、変更、削除を実行するコードによって別々のトランザクションが同時に実行される可能性を考慮してください。

Paul Robertson が作成したオープンソースコードライブラリを使用すると、複数の SQLConnection オブジェクトを使用する利点を取り入れながら、潜在的な弱点を最小限に抑えることができます。このライブラリでは、SQLConnection オブジェクトのプールを使用し、関連する SQLStatement オブジェクトを管理します。この方法で、SQLStatement を再利用し、複数の SQLConnection オブジェクトを使用して複数のステートメントを同時に実行できます。ライブラリについて詳しくは、またライブラリをダウンロードするには、http://probertson.com/projects/air-sqlite/ を参照してください。

データベースファイルの最適化

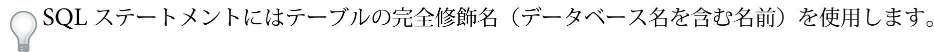

データベーススキーマの変更を避けます。

いったんデータベースのテーブルにデータを追加したら、なるべくデータベースのスキーマ（テーブル構造）を変更しないようにします。データベースファイルでは通常、ファイルの先頭にテーブル定義が配置されています。データベースへの接続を開くと、それらの定義が読み込まれます。データベースのテーブルに追加したデータはファイルのテーブル定義データの後に追加されますが、スキーマを変更すると、新しいテーブル定義データと、データベースファイルのテーブルデータが混合されます。例えば、列をテーブルに追加するか、新しいテーブルを追加すると、様々な種類のデータが混在するようになります。テーブル定義データの一部が、データベースファイルの先頭に配置されていない場合、データベースへの接続を開く時間が長くなります。ファイル内の様々な場所からテーブル定義データを読み取るのに時間がかかるので、接続を開く速度が遅くなるのです。


スキーマを変更した後は、SQLConnection.compact() メソッドを使用してデータベースを最適化します。

スキーマを変更する必要がある場合は、変更が完了した後に SQLConnection.compact() メソッドを呼び出します。これにより、データベースファイルが再構築されて、テーブル定義データがファイルの先頭にまとめて配置されます。ただし、compact() 操作には時間がかる場合があります（データベースファイルが大きくなるにつれてその傾向が強くなります）。

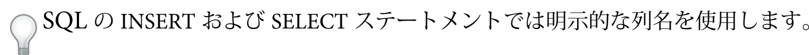
不要なデータベースランタイム処理


SQL ステートメントにはテーブルの完全修飾名（データベース名を含む名前）を使用します。

ステートメントの各テーブル名には常にデータベース名を明示的に指定します（メインデータベースの場合は「main」を使用します）。例えば、次のコードには、明示的なデータベース名 main が含まれます。

```
SELECT employeeId
FROM main.employees
```

データベース名を明示的に指定すると、一致するテーブルを見つけるためにランタイムが各接続データベースをチェックする必要がなくなります。間違ったデータベースが選択される可能性もなくなります。SQLConnection が接続されているデータベースが 1 つしかない場合でもこのルールに従ってください。SQLConnection は、SQL ステートメントからアクセスできる一時データベースにも背後で接続されています。


SQL の INSERT および SELECT ステートメントでは明示的な列名を使用します。

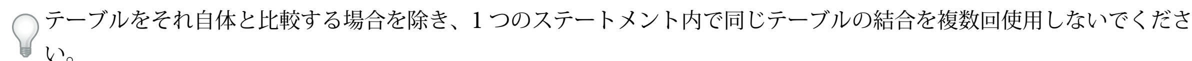
明示的な列名の使用例を次に示します。

```
INSERT INTO main.employees (firstName, lastName, salary)
VALUES ("Bob", "Jones", 2000)

SELECT employeeId, lastName, firstName, salary
FROM main.employees
```

前の例と次の例を比較してください。この形式のコードは使用しないでください。

```
-- bad because column names aren't specified
INSERT INTO main.employees
VALUES ("Bob", "Jones", 2000)

-- bad because it uses a wildcard
SELECT *
FROM main.employees
```

明示的な列名を指定しないと、列名を特定するために追加の処理が実行されます。SELECT ステートメントで明示的な列ではなくワイルドカードを使用すると、余分なデータを取得する処理が発生します。この余計なデータには追加の処理が必要であり、必要ではない余計なオブジェクトインスタンスが作成されます。

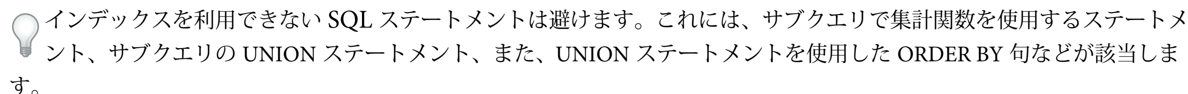

テーブルをそれ自体と比較する場合を除き、1 つのステートメント内で同じテーブルの結合を複数回使用しないでください。

SQL ステートメントが長くなると、意図せずに 1 つのデータベーステーブルを複数回クエリに結合してしまうことがあります。そのようなステートメントを見直すと、当該テーブルを 1 回使用するだけで同じ結果を得られることが少なくありません。同じテーブルを複数回結合する操作は、ビューを 1 つまたは複数使用しているクエリで発生しがちです。例えば、テーブルをクエリに結合し、さらに、そのテーブルのデータが含まれるビューにも結合してしまうことがあります。この 2 つの操作により、複数の結合が発生することになります。

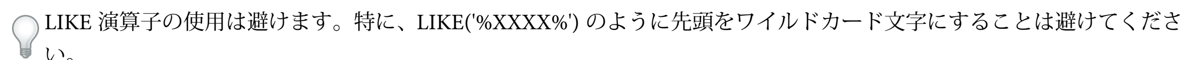
効率的な SQL 構文

WHERE 句内のサブクエリではなく、（FROM 句で）JOIN を使用して、クエリにテーブルを含めます。結果セットではなくフィルター処理のためにのみテーブルのデータが必要な場合でも、このヒントは適用されます。

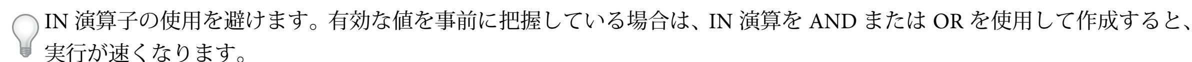
FROM 句で複数のテーブルを結合すると、WHERE 句でサブクエリを使用するよりもパフォーマンスが向上します。


インデックスを利用できない SQL ステートメントは避けます。これには、サブクエリで集計関数を使用するステートメント、サブクエリの UNION ステートメント、また、UNION ステートメントを使用した ORDER BY 句などが該当します。

インデックスによって SELECT クエリの処理速度は大幅に向上する可能性があります。ただし、一部の SQL 構文では、データベースにインデックスを使用できず、検索操作またはソート操作に実際のデータを使用する必要があります。

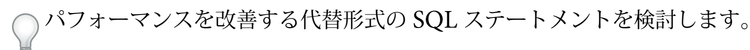

LIKE 演算子の使用は避けます。特に、LIKE('%XXXX%') のように先頭をワイルドカード文字にすることは避けてください。

LIKE 演算はワイルドカード検索の使用をサポートしているので、完全一致の比較を使用する演算子に比べると低速です。特に、検索ストリングの先頭をワイルドカード文字で開始すると、検索時にインデックスを使用できません。その場合、データベースは、テーブルの各行を全文検索する必要があります。


IN 演算子の使用を避けます。有効な値を事前に把握している場合は、IN 演算を AND または OR を使用して作成すると、実行が速くなります。

次の 2 つのステートメントでは、2 番目のステートメントのほうが高速に実行されます。高速な理由は、IN() または NOT IN() ステートメントを使用するのではなく、単純な等価式と OR を組み合わせて使用しているからです。

```
-- Slower
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary IN (2000, 2500)

-- Faster
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary = 2000
    OR salary = 2500
```


パフォーマンスを改善する代替形式の SQL ステートメントを検討します。

前の例で示したように、SQL ステートメントの記述方法もデータベースのパフォーマンスに影響を与えます。多くの場合、SQL SELECT ステートメントを記述して特定の結果セットを取得するには複数の方法があります。場合によっては、ある方法が別の方法よりも大幅に実行が高速になることもあります。上記の説明だけでなく、SQL 言語専用のリソースから、様々な SQL ステートメントとパフォーマンスについて詳しく学習できます。

SQL ステートメントのパフォーマンス

高速なステートメントを判断するには、代替の SQL ステートメントを直接比較します。

複数バージョンの SQL ステートメントのパフォーマンスを比較する最適な方法は、データベースおよびデータを直接使用してテストすることです。

次の開発ツールは、SQL ステートメントを実行するときの実行時間を提示します。その時間を使用して、代替バージョンのステートメントの速度を比較します。

- 「Run!」(Paul Robertson による AIR SQL クエリのオーサリングおよびテストツール)
- 「Lita」(David Deraedt による SQLite Administration Tool)

第 9 章：ベンチマークおよびデプロイ

ベンチマーク

ベンチマークアプリケーションには多数のツールを利用できます。Flash コミュニティメンバーによって開発された Stats クラスおよび PerformanceTest クラスを使用できます。また、Adobe® Flash® Builder™ のプロファイラー、および FlexPMD ツールを使用することもできます。

Stats クラス

外部ツールを使用することなく、リリースバージョンのランタイムを使用して実行時にコードをプロファイルするには、Flash コミュニティの Mr.doob によって開発された Stats クラスを使用できます。Stats クラスは、https://github.com/mrdoob/Hi-ReS-Stats からダウンロードできます。

Stats クラスを使用すると、次のものを追跡できます。

- 1 秒ごとにレンダリングされるフレーム数（この数値が高いほど良い）
- フレームのレンダリングにかかったミリ秒（この数値が低いほど良い）
- コードで使用されているメモリの量これが各フレームで増える場合は、アプリケーションでメモリリークが発生している可能性があります。メモリリークの可能性がある場合は、その原因を調査することが重要です。
- アプリケーションが使用した最大メモリ量。

Stats クラスは、ダウンロード後に、次のコンパクトコードと共に使用できます。

```
import net.hires.debug.*;
addChild( new Stats() );
```

Adobe® Flash® Professional または Flash Builder の条件付きコンパイルを使用して、Stats オブジェクトを有効にすることができます。

```
CONFIG::DEBUG
{
    import net.hires.debug.*;
    addChild( new Stats() );
}
```

DEBUG 定数の値を切り替えることで、Stats オブジェクトのコンパイルを有効または無効にすることができます。これと同じ手法を使用して、アプリケーションでコンパイルしないコードロジックを置き換えることができます。

PerformanceTest クラス

Grant Skinner 氏は、ActionScript コードの実行をプロファイルするため、ユニットテストワークフローに統合できるツールを開発しました。カスタムクラスを PerformanceTest クラスに渡します。これにより、一連のテストがコードで実行されます。PerformanceTest クラスを使用すると、様々な手法のベンチマークを簡単に行うことができます。PerformanceTest クラスは http://www.gskinner.com/blog/archives/2009/04/as3_performance.html からダウンロードできます。

Flash Builder プロファイラー

Flash Builder には、高レベルの詳細でコードのベンチマークを行うことができるプロファイラーが用意されています。

注意：このプロファイラーにアクセスするには、Flash Player のデバッガーバージョンを使用します。このようにしないと、エラーメッセージが表示されます。

このプロファイラーは、Adobe Flash Professional で作成されたコンテンツにも使用できます。そのためには、ActionScript から、コンパイルされた SWF ファイルをロードするか、Flex プロジェクトを Flash Builder にロードし、そこでプロファイラーを実行できます。プロファイラーについて詳しくは、Flash Builder 4 ユーザーガイドの「Flex アプリケーションのプロファイリング」を参照してください。

FlexPMD

アドビテクニカルサービスは、FlexPMD と呼ばれるツールをリリースしました。このツールを使用すると、ActionScript 3.0 コードの品質を監査することができます。FlexPMD は、JavaPMD に類似した ActionScript ツールです。FlexPMD は、ActionScript 3.0 または Flex ソースディレクトリを監査して、コードの品質を向上させます。未使用のコード、過度に複雑なコード、過度に長いコード、Flex コンポーネントライフサイクルの誤った使用など、好ましくないコーディング手法を検出します。

Adobe オープンソースプロジェクトの FlexPMD は、http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD で入手できます。Eclipse プラグインは、http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD+Eclipse+plugin で入手できます。

FlexPMD を使用すると、コードを監査し、コードがクリーンで最適化されていることを確認できます。FlexPMD の本当の機能は、その拡張性にあります。開発者は、独自のルールセットを作成して任意のコードを監査できます。例えば、フィルターの過度の使用や、把握する必要がある好ましくないコーディング手法を検出するルールセットを作成できます。

デプロイ

アプリケーションの最終バージョンを Flash Builder で書き出すときは、そのリリースバージョンを書き出すことが重要です。リリースバージョンの書き出しにより、SWF ファイルに含まれているデバッグ情報が削除されます。デバッグ情報を削除すると、SWF ファイルサイズが小さくなり、アプリケーションの実行速度が高まります。

プロジェクトのリリースバージョンを書き出すには、Flash Builder のプロジェクトパネルを使用し、「リリースビルドの書き出し」オプションを使用します。

注意：Flash Professional でプロジェクトをコンパイルするときは、リリースバージョンとデバッグバージョンを選択するオプションはありません。コンパイルされた SWF ファイルは、デフォルトでリリースバージョンになります。