Autodesk®
Scaleform®

Scaleform Best Practices Guide

本書では、Scaleform 4.0 以降のバージョンを使ったアセットの作成にベスト プラクティスを提供しています。

著者: Matthew Doyle
バージョン: 3.0
最終更新日: 2011 年 4 月 25 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

Autodesk Scaleform の連絡先:

ドキュメント	Scaleform　Best Practices (Scaleform のベスト プラクティス ガイド)
住所	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
ホームページ	www.scaleform.com
電子メール	info@scaleform.com
電話	(301) 446-3200
Fax	(301) 446-3199

目次

1 はじめに

この文書には、Autodesk® Scaleform® 4.0 以降での使用に Adobe Flash®でコンテンツを開発する時のためのベストプラクティスが集められています。紹介している実例は、特に、ゲーム内での Flash コンテンツのメモリとパフォーマンスの改善を対象としていますが、他の用途にも使用することができます。本書のコンテンツ作成の章は、アーティストやデザイナーを対象としていますが、ActionScript™ (AS) の章は、テクニカル アーティストやエンジニアを対象としています。ヘッズ アップ ディスプレイ (HUD) 開発の章は、HUD の作成に可能な開発シナリオの総括です。Flash と Scaleform 3.0 以降のバージョンを使って HUD システムを開発する前に、アーティストとエンジニアの両者がこの章を読むことをお勧めします。

本書は、カスタマー サポートへのリクエスト、開発者のフォーラムの投稿内容、Web 上の各種の Flash や AS リソースなど、さまざまな情報源からの豊富な情報を編集したものです。Scaleform は (携帯電話から最新の家庭用ゲーム機や PC に至るまで) 多くの異なるプラットフォームで使用され、各種のエンジンとの統合が可能です。従って、残念ながらすぐに何にでも使えるソリューションはありません。ゲームやプロジェクトごとに、最適なメモリ消費とパフォーマンスのために異なるソリューションが必要になります (例えば、ハイエンド PC からハンドヘルド プラットフォームに移植するときは、コードの調整が必要になるでしょう)。

ベスト プラクティスと共に、関連するパフォーマンス データの提供に尽力してきましたが、変数要素も多いので、これらの結果が自分のプロジェクトで同じとは限りません。すべてのユーザー インターフェイス (UI) を、徹底してテストされることをお勧めします。さらに UI の開発過程の早い段階で、選択可能なソリューションは、それぞれの使用条件下で負荷テストを行っておく必要もあります。

Flash のデザイナーや開発者は、自分も含め、同じプロジェクトで作業している他のメンバーにも直感的に分かり易く、役に立つようにコードを書き、アプリケーションを構築しなければなりません。多くのアセットを含んでいる FLA ファイルでは、これは特に重要です。なお、Flash Studio CS5 からは XML によるコンテンツ フォーマットが導入されました。このフォーマットはバイナリの.FLA フォーマットを置き換えるもので、バージョン管理やチェンジ リストの追跡に役立ちます。

複数のデザイナーや開発者が 1 つの Flash プロジェクトで作業することが一般的なので、全員が Flash の使用、FLA ファイルの構成、AS コードの記述に対する標準のガイドラインに従うと、チーム全体にプラスになります。本書の各章では、AS を記述する場合や、リッチ メディア コンテンツを作成する Flash オーサリング ツールを使用する場合のベスト プラクティスについて説明しています。デザイナー、開発者に関わらず、また単独で作業しているか、チームの一員として動いているかに関係なく、常にベスト プラクティスを選択してください。以下は、ベスト プラクティスを活用することの利点の一部です:

- Flash、または AS ドキュメントで作業する場合:
    - 一貫性があり効率的なプラクティスを採用すると、ワークフローをスピードアップすることができます。すでに確立されているコーディング規則を使って開発するほうが時間的に早く、後でドキュメントを編集するときに、その構造を理解しやすく覚えや

すくなります。また、さらに大規模なプロジェクトのフレームワーク内では、多くの場合コードの移植性がより高く再利用し易くなります。

- FLA、または AS ファイルを共有する場合:
    - そのドキュメントを他人が編集するときに、簡単に AS を検索して理解し、一貫性を保ってコードの変更を行い、アセットの検索、編集を行うことができます。
- アプリケーションで作業する場合:
    - 複数の作成者が、競合することなく効率的に、アプリケーションで作業することができます。プロジェクトやサイトの管理者は、複雑なプロジェクトまたはアプリケーションを、ほとんど競合や冗長なしに、管理および構築することができます。
- Flash と AS を学習、または教育する場合:
    - ベスト プラクティスクを使い、コーディング規則に従ってアプリケーションを構築する方法を学ぶと、特定の方法論を再度学習する必要性が減ります。Flash の学習者が、一貫性があり優れたコードの構築方法を学ぶと、その言語をより簡単に、フラストレーションもそれほど感じずに学習できるかもしれません。

開発者はこのようなベスト プラクティスを読んで、自分のより良い習慣を身に付けて行くと、必ずさらに多くのことを発見することになるでしょう。以下のトピックは、Flash で作業するときのガイドラインと考えてください。開発者は推奨事項の一部を実践することも、すべてに従うこともできます。また、自分の作業方法に合わせて、推奨事項を変更しても構いません。この章のガイドラインの多くは、Flash の作業と AS コードの記述について一貫性のある方法を、開発者が利用できるようにしています。

Flash で最適化が可能な分野は以下のとおりです:

- 高速の再描画処理のためにグラフィックを最適化することで、アニメーション パフォーマンスを向上します。
- 短時間で実行されるコードを書くことで、計算処理を向上します。

コードの実行が遅い場合、それは開発者がアプリケーションのスコープを削減する、またはもっと効率的な方法でその問題を解決する別の手段を探す必要がある、というサインである可能性が高くなります。開発者は、例えば、グラフィックの最適化や ActionScript で行う作業の削減によって、ボトルネックの識別と削除を行う必要があります。

多くの場合、パフォーマンスとは感じ方です。開発者が 1 つのフレームであまりに多くの作業を実行しようとすると、Flash はステージをレンダリングする時間が足りなくなり、ユーザーは遅いと感じます。作業量を細かく分けてやると、Flash は所定のフレーム レートでステージを更新できるので、遅いと感じることはありません。

1.1 一般的な使用法

- ステージ上のオブジェクトの数を減らすようにします。オブジェクトを 1 つずつ追加して、いつ、どの程度パフォーマンスが劣化するかを記録します。
- 標準の UI コンポーネント (Flash の [コンポーネント] パネルで使用できるもの) の使用は避けます。これらのコンポーネントは、デスクトップ コンピュータで実行するように設計されているもので、Scaleform 3.3 で実行するために最適化されているわけではありません。代わりに、Scaleform Common Lightweight Interface Kit (CLIK™) コンポーネントを使ってください。

1.2 セットアップの検討事項

- オブジェクト階層の深度を削減するようにします。例えば、オブジェクトの中に独立して移動したり回転しないものがある場合、これらのオブジェクトは自らの変形グループ内に存在する必要はありません。
- 大規模なユーザー インターフェイスは通常、個々の Flash ファイルに分けるのが最善の策です。例えば、MMORPG のトレード ルームは 1 つの Flash インターフェイスであり、ライフ メーターHUD はまた別のインターフェイスになります。AS を使用する場合は、AS2 と AS3 のいずれからでも、各種ファイルは複数の SWF ローディング API を介してロード可能です。これらの関数を使うと、アプリケーションは特定の時間に必要なインターフェイスだけをロードして、もう必要なくなるとコンテンツを解放することができます。別の方法として、各種の SWF ファイルを C++ API 経由で (異なる GFx::Movie インスタンスを作成して)、ロード/アンロードすることがあります。
- Scaleform 3.3 はマルチスレッド ロードとプレイバックをサポートします。複数の Flash ファイルのバックグラウンド読み込みは、追加で CPU コアが使用できる場合、パフォーマンスに悪影響を与えることはありません。ただし、CD や DVD から複数のファイルをロードする場合、開発者は注意しなければなりません。ディスク検索にかなり時間がかかり、ファイルのロードが大幅に遅くなる場合があるからです。
- シーンを使用すると SWF ファイルのサイズが大きくなる場合が多いので、使用しないようにします。
- ゲームに使用される UI メモリの現実的な量は、以下の範囲に収まると考えてよいでしょう:
  - シンプルなゲーム HUD のオーバーレイ: 400K–1.5M
  - アニメーションと複数の画面が含まれるスタートアップ/メニュー画面: 800K–4M
  - アセットを含む、完全に ActionScript で書かれた簡単なゲーム (パックマンなど): 700K–1.5M
  - アセットが常時変わる長尺のベクター アニメーション: 2M–10M+

2　コンテンツ作成時のメモリとパフォーマンスについての考察

Flash コンテンツの開発では、最適なパフォーマンスを得るために実装するべき検討事項や最適化事項はいくつもあります。

2.1 描画プリミティブ

描画プリミティブ（DP）とは、シェイプなどの Flash エレメントをレンダリングして表示するために、GFx によって作成されるメッシュ オブジェクトです。バッチ型（またはインスタンス型）レンダリングをサポートしているプラットフォームでは、GFx は、同一レイヤー上または隣接レイヤー上にある同一プロパティの複数のシェイプを単一の DP にグループ化しようと試みます。各 DP は個別にレンダリングされるため個数が多いと性能低下を招きます。一般的に、多くの DP がシーンに導入されるだけ、Display パフォーマンスは直線的に低下する傾向があります。従って、DP の数を可能な限り低く抑えることをお勧めします。描画プリミティブの数は、F2 キーを押して Scaleform Player HUD 経由で見ることができます。このキーを押すと HUD の Summary 画面が表示されます。この画面にはトライアングルの数、DP、メモリの使用量、そして他の最適化情報が表示されます。

以下は、描画プリミティブ数を低くしておくためのヒントです:

1. DP は同じプロパティのアイテムからしか構成されません（ソリッド カラーが異なるシェイプを除く）。たとえば、テクスチャあるいはブレンディング モードが異なるアイテムは、単一の DP として統合されません。
2. 異なるレイヤー上でオーバーラップするオブジェクトは、プロパティが同じでも、単一の DP として統合できません。
3. ひとつのシェイプに対して複数のグラディエント フィルが同時に用いられると DP の数が増えることがあります。
4. ソリッド カラー フィルがありストロークのないベクター グラフィクスは、性能低下はわずかです。
5. 空のムービー クリップは DP を必要としません。ただし、内部にオブジェクトを有するムービー クリップは、それらのオブジェクトで指定されるだけの DP を必要とします。

バッチ グループまたはインスタンス グループを Scaleform Player にて表示するには、AMP 内のオプションを有効にするか Ctrl+J を押下します。このモードでは完全に同色のアイテムは単一の DP としてレンダリングされます。

2.2 ムービー クリップ

1. ムービー クリップを非表示にするよりも、使用しないときは、タイムラインから完全に削除するのが最善の策です。削除しなければ、Advance 中に処理時間が取られる場合があります。

2. パフォーマンスに影響するので、ムービー クリップの過度なネスト化を避けます。

3. ムービー クリップを非表示にする必要がある場合、_alpha=0 ではなく_visible=false(in AS2) and visible=false rather than alpha=0 (in AS3)を使用します。"stop()" 関数を呼び出して、非表示のムービー クリップでアニメーションを停止したことを確認してください。そうしない場合は、非表示のアニメーションは依然実行され、パフォーマンスに影響します。

2.3 アートワーク

2.3.1 ビットマップ グラフィック vs ベクター グラフィック

Flash コンテンツは、イメージの他にもベクター アートで作成することができ、Scaleform はスムーズにベクター グラフィックとビットマップ グラフィックの両方をレンダリングすることができます。ただし、それぞれのグラフィックには、長所と短所があります。ベクター グラフィックを使うか、ビットマップ グラフィックを使うかという決断は、常に明確ではなく、多くの場合複数の要因に左右されます。この章では、ベクター グラフィックとビットマップ グラフィックの違いの一部を説明して、コンテンツのオーサリング決定の一助となればと思います。

ベクター グラフィックはサイズが変わったときに、その滑らかなシェイプを維持しますが、ビットマップ イメージは、サイズ変更されると箱型に、あるいはモザイク化されて表示されます。しかし、ビットマップとは異なり、ベクター グラフィックは生成するための処理能力がさらに必要になります。簡単な単色のシェイプは通常、ビットマップと同程度の速さになりますが、多くのトライアングル、シェイプ、塗りつぶしを含むベクター グラフィックは、レンダリングの負荷が高くなる場合があります。

ビットマップ グラフィックは、ベクター グラフィックほどレンダリングの処理時間を必要としないので、ビットマップ グラフィックのほうが一部のアプリケーションには向いている場合があります。しかし、ベクター グラフィックと比較すると、ビットマップ グラフィックでは必要なメモリ サイズが著しく増加します。

2.3.2 ベクター グラフィック

ベクター グラフィックは、他のイメージ フォーマットよりもさらにコンパクトです。というのは、ビットマップのような未加工のグラフィック (ピクセル) データではなく、実行時にイメージのレンダリングに必要な計算 (点、曲線、塗りつぶし) を定義するからです。しかし、ベクター データを最終的なイメージに変換するのは時間のかかる作業であり、グラフィックの外観やサイズに大きな変更があるときは常に行わなければなりません。ムービー クリップに、フレーム毎に変わる複雑なシェイプのアウトラインが含まれている場合、アニメーションの表示が遅くなる場合があります。

以下はベクター グラフィックを効率的にレンダリングするためのガイドラインです:

- 複雑なベクター グラフィックをビットマップに変換して、これがどのようにパフォーマンスに影響するかをテストする。
- アルファ ブレンドを使用するときは以下の点に注意する。
  - 単色実線のストロークは、はるかに効率の良いアルゴリズムを使用できるので、アルファブレンドのストロークよりも計算の負荷が低くなります。
  - 透明 (アルファ) の使用を避けます。Flash は透明のシェイプの下にあるピクセルをすべて確認しなければならないので、大幅にレンダリングが遅くなります。クリップを非表示にするには、_alpha プロパティを 0 に設定するのではなく、その _visible(AS2) or visible (AS3)プロパティを false に設定します。グラフィックは _alpha(AS2) or alpha (AS3) property が 100 に設定されているときに、最速でレンダリングされます。ムービー クリップのタイムラインを空白のキーフレームに設定すると (ムービー クリップには表示するコンテンツがなくなります)、通常、さらに速度が増します。場合によっては、Flash はそれでも非表示のクリップをレンダリングしようとします。_visible/visible プロパティを false に設定するほかに、そのクリップの_x と_y(in AS2) or x and y properties (in AS3)プロパティを表示されているステージ外の位置に設定して、ステージの外にそのクリップを移動します。そうすれば Flash はまったくそのクリップを描画しようとしなくなります。
- ベクター シェイプを最適化する。
  - ベクター グラフィックを使用するときに、余分なポイントを削除して、シェイプを可能な限り簡素化するようにします。これにより、Player がベクター シェイプ毎に算出しなければならない計算量が減少します。
  - 円、四角形、ラインなどのプリミティブなベクターを使用します。
  - Flash の描画パフォーマンスは、フレーム毎に点が何個描画されるかに関連しています。[修正] -> [シェイプ] を選択し、次に [滑らかに]、[まっすぐに]、または [最適化] を選択して (対象のグラフィックに応じて) シェイプを最適化し、そのシェイプの描画に必要な点の数を削減します。これにより、Scaleform ベクター テッセレーション コードが作成するメッシュ データを削減することができます。
- 角のほうが曲線よりも負荷が低くなる。
  - 曲線や点が多すぎる複雑なベクターを避けるようにします。
  - 角は曲線よりも数学的に簡単にレンダリングすることができます。可能な場合、特に非常に小さなベクター シェイプでは、エッジをフラットにします。曲線はこの方法でシミュレートすることができます。
- グラデーション塗りとグラデーション ストロークは、控えめに使用する。
- シェイプのアウトライン (ストローク)を避けます。
  - レンダリングされる線数が増えるので、できる限り、ベクター シェイプを持つストロークを使用しないようにします。
  - ベクター イメージの周囲のアウトラインは、パフォーマンス ヒットに影響します。

    - 塗りつぶしは、外側のシェイプしかレンダリングしませんが、アウトラインは内側と外側のシェイプをレンダリングします。これは塗りつぶしよりも、線の描画に 2 倍の作業が必要です。
- Flash の描画 API の使用を最小限にする。必要以上に使用すると、重大なパフォーマンス オーバーヘッドの原因となる場合があります。必要であれば、描画 API を使って、1 度だけムービー クリップ上で描画します。そのようなカスタムのムービー クリップのレンダリングには、パフォーマンス ペナルティがありません。
- マスクの使用を制限する。マスクが適用されたピクセルはレンダリング時間を使い果たし、描画されないとしても、パフォーマンスに悪影響があります。複数のマスクを使う場合は、使用されているマスク数に応じて、影響がさらに大きくなります。多くの場合、アーティストがマスクを使用する視覚効果は、どうしてもマスクが必要であるとも限らないことに注意してください。特に、ビットマップからシェイプを切り抜くにはマスクを使用するのが一般的ですが、同じ効果は、Flash Studio で直接シェイプにビットマップ塗りつぶしを適用することで、はるかに効率的に達成することができます。また、これにより、Scaleform の特許出願中の EdgeAA アンチエイリアスが使えるという追加のメリットもあります。
- オーバーラップしていない複数のオブジェクトは、前述のとおり余分な描画プリミティブの生成を回避するために、同じプロパティになるようできる限り変換してください。
- シェイプの作成後、メモリ消費量を増やさずにそのシェイプを変換、回転、ブレンドすることができる。ただし、新たに大きなシェイプを取り入れる、または大幅なスケーリングを行うと、テッセレーションのためにさらに多くのメモリを消費することになります。
- EdgeAA を有効にした状態で、複数の色で構成されているシェイプは、複数のグラデーション/ビットマップで作成されたものよりも速くレンダリングされる。1 つのシェイプの中でグラデーション/ビットマップを組み合わせるときは、注意が必要です。これは描画プリミティブが急増する原因となるからです。

2.3.3 ビットマップ

最適化と合理化されたアニメーションやグラフィックを作成するときに最初に行う手順は、作成前に自分のプロジェクトの輪郭を描き、プランを立てることです。作成したいアニメーションのファイル サイズ、メモリ使用量、長さの目標値を指定し、開発過程でテストを繰り返し、順調に進んでいることを確認します。

前述した描画プリミティブの他に、レンダリングのパフォーマンスに影響する大きな要因は、描画される総表面積です。目に見えるシェイプやビットマップがステージに置かれるたびに、他のオーバーラップするシェイプに隠されていても、隠れた部分をレンダリングする必要があり、ビデオ カードのフィルレートを消費します。現在のビデオ カードは、ソフトウェアの Flash よりも 10 倍高速ですが、画面上の大きなオーバーラップするアルファブレンド オブジェクトは、それでもパフォーマンスを大幅に低下させる場合があります。特によりローエンドなハードウェアや古いハードウェア上では顕著です。このため、オーバーラップするシェイプやビットマップをフラット化して、ぼやけた、あるいは切り取られたオブジェクトを明確に隠すことが重要です。

オブジェクトを隠すとき、SWF ファイルで_alpha(in AS2) or alpha (in AS3 レベルを 0 に変更するのではなく、ムービー クリップ インスタンスの_visible(in AS2) or visible (in AS3)プロパティを false に設定するのが最善の方法です。Scaleform は_alpha/alpha 値が 0 のオブジェクトは描画しませんが、それでも、アニメーションや ActionScript が原因で、その子であるオブジェクトに CPU 処理のコストがかかる場合があります。そのインスタンスの可視性が false に設定されている場合、CPU サイクルとメモリの節約の可能性があり、それによって、SWF ファイルに、より滑らかなアニメーションを提供し、アプリケーション全体のパフォーマンスを向上することができます。アセットのアンロードとリロードの代わりに、_visible/visible プロパティを false に設定します。これはそれほどプロセッサインテンシブではありません。以下は、ビットマップ グラフィックを効率的にレンダリングするためのガイドラインです:

- 幅と高さに 2 のべき乗を使って、すべてのテクスチャ/ビットマップを作成する。ビットマップ サイズの例には、16x32、256x128、1024x1024、512x32 などがあります。
- ターゲット ハードウェアがサポートしていない圧縮イメージは使用してはなりません（たとえば JPEG イメージは圧縮されていますが、すべてのプラットフォームのハードウェアでサポートされているわけではありません）。ファイルのロード中に解凍時間が必要になります。
- gfxexport ツールとテクスチャ圧縮スイッチを使ってターゲット プラットフォーム ハードウェアでサポートされる圧縮イメージを作成してください（たとえば、多くのプラットフォームが DXT 圧縮をサポートしています）。最終的な SWF を GFx フォーマットに変換し、必要なビットマップ メモリ容量をテクスチャ圧縮によって削減します。圧縮テクスチャのほうが非圧縮テクスチャに比べて必要なビットマップメモリ容量は大幅に少なくなります。ただし、圧縮テクスチャ フォーマットの多く（DXT を含む）は不可逆圧縮を使用していますので、得られたビットマップの画質が十分か確認してください。gfxexport のオプション–qp または–qh を使って、最高画質の DDS テクスチャを取得します (これらのオプションは、ビットマップ イメージの処理に長時間要する場合があるので注意してください)。
- ビットマップの数を減らし、さらにサイズの小さなビットマップを使用する、あるいはそのいずれかを行う。
- 大きなシンプルなシェイプの表示にビットマップを使用している場合、ベクター グラフィックを使ってそのビットマップを作り直す。こうすると、Edge AA でより高画質が得られ、メモリの節約になります。
- 必要に応じて、ActionScript を使って大きなビットマップのロード/アンロードを考える。
- SWF/GFX ファイルのサイズで、そのメモリ使用量を判断しない。. SWF ファイル サイズが小さくてもロード時に多くのメモリを使う可能性が残されています。たとえば、SWF ファイルに 1024×1024 の埋め込み JPEG イメージが格納されている場合、伸張後のランタイムにそのイメージは 4 MB を必要とします。
- UI で使用されているイメージ ファイルの数とサイズを記録しておくことが重要。すべてのイメージのサイズをカウントして合計し、その値に 4 を掛けます (通常はピクセル毎に 4 バイトです)。gfxexport ツールを–i DDS オプションで使用し、テクスチャ圧縮を使っ

て 4 という因数でイメージのメモリ使用量を削減する必要があります。AMP を使って、ビットマップによるメモリの消費量を確認します。

- 全体的に見て、ほとんどの場合、ビットマップは複雑なシェイプに対しては高速になりますが、ベクターのほうが見映えが良くなります。正確なトレードオフは、使用しているシステムのフィルレート、変換シェーダのパフォーマンス、そして CPU の速度に左右されます。つまり、確認できる唯一の現実的な方法は、ターゲット システム上でパフォーマンスをテストすることです。
- グラデーション テクスチャだけを含むマスターgradient.swf ファイルを作成して、必要に応じてそれを別の SWF ファイルにインポートする。gfxexport ツールを使って、-d0 スイッチでその gradients.swf をエクスポートします。このスイッチは圧縮を不可にして、その SWF ファイルのすべてのテクスチャに適用されます。このスイッチは、グラデーションを使用するこのファイルのすべてのテクスチャが、バンディングなしであることを確認します。
- 可能な場合は常に、アルファ チャンネルを持つビットマップを避ける。
- Flash ではなく、Adobe Photoshop® などの画像処理アプリケーションでビットマップを最適化しておく。
- ビットマップの透明化が必要な場合は PNG を使用して、描画される総表面積を削減する。
- フィルレート パフォーマンスに影響するので、大きなビットマップを重ねない。
- アプリケーションで使用されるサイズで、ビットマップ グラフィックをインポートする。大きなグラフィックをインポートして、それを Flash で縮小しないでください。ファイル サイズとランタイム メモリの浪費になります。

2.3.4 アニメーション

アニメーションをアプリケーションに追加するとき、FLA ファイルのフレーム レートを考慮します。フレーム レートは、最終的な SWF ファイルのパフォーマンスに影響する場合があります。特に、多くのアセットが使用されている場合、または AS を使って、ドキュメントのフレーム レートで動作するアニメーションを作成している場合、フレーム レートを高く設定しすぎるとパフォーマンス問題につながることがあります。

一方、フレーム レート設定は、アニメーションのスムーズな再生に影響します。例えば、12 フレーム/秒 (FPS) に設定されたアニメーションは、毎秒、タイムラインの 12 個のフレームを再生します。そのドキュメントのフレーム レートが 24 FPS に設定されている場合、アニメーションは 12 FPS に設定されているときよりも、より滑らかに動作するように見えます。ただし、24 FPS のアニメーションは 12 FPS のときよりも 2 倍の速さで再生されるので、合計持続時間 (秒単位) はその時間の半分になります。従って、高フレーム レートで 5 秒間のアニメーションを作成するには、低いフレーム レートのときよりもその 5 秒間を埋めるための追加フレームが必要になります。これにより合計ファイル サイズが増加してしまいます。

注意**:** onEnterFrame(in AS2) or Event.ENTER_FRAME (in AS3)イベント ハンドラを使って、スクリプティングされたアニメーションを作成する場合、アニメーションは、ドキュメントのフレーム レートで実行されます。これは、タイムラインでモーション トゥイーンを作成するのに似ています。onEnterFrame/Event.ENTER_FRAME イベント ハンドラの代わりに、setInterval を使用することもできます。フレーム レートに依存するのではなく、特定のミリ秒間隔で関数が呼び出されます。onEnterFrame/EVENT.ENTER_FRAME のように、setInterval を使って関数を呼び出す頻度が高いほど、アニメーションは多くのリソースを消費します。

実行時にアニメーションを滑らかにレンダリングできる、可能な限り低いフレーム レートを使用することです。これで、プロセッサのパフォーマンス ヒットを削減できます。30 – 40 FPS を超えるフレーム レートを使わないようにします。このポイントを超える高フレーム レートは、CPU コストを増やしますが、アニメーションの滑らかさをあまり向上することはありません。ほとんどの場合、Flash UI は、ベースとなるゲームのターゲット フレーム レートの半分に設定しておくと、安全です。

以下は、効率的なアニメーションの設計と作成に役立つガイドラインです:

- ステージ上のオブジェクト数と物が動く速度は、全体のパフォーマンスに影響する。
- ステージ上に大量のムービー クリップがあり、素早くオン/オフを切り替える必要がある場合、ムービー クリップの付加/削除を行う代わりに、_visible/visible = true/false を使ってその可視性をコントロールする。
- トゥイーンの使用は、慎重に。
    - 同時にあまりに多くのアイテムをトゥイーンしないようにする。トゥイーンと連続するアニメーションの数を減らす、あるいはそのいずれかでも数を減らして、1 つが終わったら別のものが始まるようにします。
    - 可能であれば、タイムライン モーション トゥイーンを使用する。標準の Flash Tween クラスより、パフォーマンス オーバーヘッドがはるかに少なくすみます。
    - Scaleform は、CLIK Tween クラス (gfx.motion.Tween in AS2 or scaleform.clik.motion.Tween in AS3) を使用することを勧めます。標準の Flash Tween クラスと比べて、このクラスのほうが小さく、高速で、クリーンだからです。
- フレームレートは低くしておく。高フレームレートと低フレームレートの差は、ほとんど分かりません。フレームレートが高ければ、アニメーションは滑らかになりますが、パフォーマンスへの影響は増加します。毎秒 60 フレームで実行しているゲームで、Flash ファイルを 60 FPS に設定する必要はありません。Flash のフレームレートは、必要な視覚効果を作成するための最小限の値にするべきです。
- 透明とグラデーションは、プロセッサ インテンシブ タスクなので、使用は控えめにする。
- フォーカスの領域をうまく設計して、アニメーション化し、画面の他の領域のアニメーションや効果を削減する。
- トランジション中は、受動的なバックグラウンド アニメーション (わずかなバックグラウンド効果など) を一時停止しておく。
- アニメーション化されたエレメントの追加と削除のテストをして、パフォーマンスへの影響を調べる。

- トゥイーンのイージングは、よく考えて使用する。速度の遅いハードウェアでは、遅延が「生じる」場合があります。
- 円を四角形に変形するなど、シェイプ モーフィング アニメーションは避ける。このようなアニメーションは非常に CPU インテンシブな操作だからです。シェイプが毎フレーム再計算されてしまうので、シェイプ トゥイーン (モーフィング) は非常に大きな CPU ヒットがあります。ヒットのコストは、シェイプの複雑さ (エッジ、曲線、交差の数) に左右されます。状況によってはそれでも使用できますが、そのコストが許容範囲であることを確認するようにシェイプを設定します。4 つの三角形トゥイーンのコストは許容できる場合があります。基本的に、認識しておくべきパフォーマンス/メモリ トレードオフがあります。標準のシェイプを表示すると、そのシェイプが今後のフレームで効率的に表示されるように、シェイプのテッセレーションとキャッシングが発生します。モーフィングの場合には、どのような変更でも、古いメッシュが解放されて新しいメッシュが生成される原因となるので、トレードオフが変わってしまうことになります。
- 最も効率的なアニメーションは変換と回転です。スケーリングを伴うアニメーションは避けるのが最善の策です。というのは、こういったアニメーションは再テッセレーション (目だったパフォーマンス ヒットになりえます) を引き起こす場合があり、その結果作成されたメッシュは、もっと多くのメモリを消費する可能性があるからです。

2.3.5 テキストとフォント

- テキスト グリフのフォント サイズは、フォント キャッシュ マネージャの SlotHeight や、gfxexport で使用する予定のサイズ (デフォルトは 48 ピクセル) よりも小さくしておきます。それよりもサイズの大きなフォントが使用された場合、ベクターが使用され、その結果多くの DP が発生するため、速度がはるかに遅くなります (各ベクター グリフが DP を作成します)。
- 可能な場合、テキスト フィールドの境界線と背景をオフにします。これにより描画プリミティブ（DP）を 1 つ節約できるからです。
- テキスト フィールドのコンテンツをフレーム毎に更新することは、最もパフォーマンスを低下させる行為の 1 つですが、簡単に避けられます。代わりに、テキスト フィールドのコンテンツが本当に変更されたときに限って、または可能な最も低いレートでテキスト フィールドの値を変更します。例えば、秒数を表示するタイマーを更新するとき、30 FPS のフレーム レートで更新する必要はありません。古い値を記録しておいて、前の値と新規の値が違う場合に限って、テキスト フィールドのその値を再割り当てします。
- テキスト フィールドにリンクしている変数 ("TextField.variable" プロパティ) を使用しないでください。テキスト フィールドは、変数をフレーム毎に取得して比較するので、パフォーマンスに影響するからです。
- “htmlText” プロパティの再割り当てによって、テキストの更新を最小限にします。HTML の解析は比較的負荷が高い処理です。
- gfxexport で、–fc、-fcl、-fcm オプションを使用して、グリフ字形のメモリが節約されるように、フォントを小型化します (特に、アジア言語のフォントが埋め込まれている場合)。

詳細は、“Font and Text Configuration Overview”（「フォントとテキスト設定の概要」）を参照してください。

- ローカライズが必要な場合、フォントに必要なシンボルだけを埋め込む、または fontlib メカニズムを使用します (再度、“Font and Text Configuration Overview”（「フォントとテキスト設定の概要」） を参照してください)。
- 必要最小限の TextField オブジェクトを使用し、出来れば常に、複数のアイテムを 1 つのアイテムに組み込みます。複数の色やフォント スタイルを使用していても、1 つのテキスト フィールドは通常は 1 つの DP でレンダリングできます。
- テキスト フィールドのスケーリング、または大きなフォント サイズの使用は避けます。特定のサイズを超えると、テキスト フィールドはベクター グリフに切り替わり、各ベクター グリフは、各 1 つの描画プリミティブになります。クリッピングが必要な場合 (ベクター グリフの一部だけが見える場合)、マスクが使えます。マスクは遅くなり、もう 1 つの描画プリミティブを追加します。ラスタライズされたグリフのクリッピングには、マスクは必要ありません。
- グリフのキャッシュ サイズが、使用されているすべての (またはほとんどの) グリフを格納するのに十分な大きさであることを確認します。このキャッシュ サイズが十分でなければ、一部のグリフが失われる可能性があります。あるいはグリフが頻繁に再ラスタライズされるために、深刻なパフォーマンス ヒットになります。
- ぼかし、ドロップ シャドウ、ノックアウト フィルタなどのテキスト効果を使うと、フォント キャッシュにさらにスペースが必要となり、パフォーマンスにも影響します。可能な場合は、テキスト フィルタの使用を最小限にします。
- 別のムービーを作成せずに、出来れば、DrawText API を使用します。DrawText API を使えば、開発者は、Scaleform で作成したユーザー インターフェイスで使用されているものと同じ Flash のフォントとテキストのシステムを使って、C++からテキストをプログラム的に描画することができます。画面上で動いているアバター上にネーム札を表示したり、レーダー上のアイテムの横にテキスト ラベルを表示することは、そのためにもう 1 つ Flash UI を持つ余裕のないゲームの場合、C++を使ったほうが効率が良い場合があります。詳細は、「DrawText API」を参照してください。

3 ActionScript の最適化

ActionScript はネイティブ マシン コードにはコンパイルされない代わりにバイトコードに変換されます。このバイトコードは解釈された言語よりも高速ですが、コンパイルされたネイティブ コードほど速くはありません。AS が遅い場合もありますが、ほとんどのマルチメディア 表現では、コードではなく、グラフィック、オーディオ、ビデオなどのアセットがパフォーマンスの制限要因です。

最適化する技術は多くありますが、AS 特有のものでなく、単に、最適化コンパイラなしに、何らかの言語でコードを書くための周知の技術です。例えば、ループ イテレーション毎に変化しないアイテムを、ループから削除してループ外に置けば、ループが速くなる、といったことです。

3.1 一般的な *ActionsScript* のガイドライン

以下の最適化で、AS の実行速度が速くなります。

- SWF を Flash バージョン 8 以降にパブリッシュする。
- AS の使用が少ないほど、ファイルのパフォーマンスは向上します。必要なタスクを実行するために書かれるコードの量は、常に最小限にします。AS はグラフィック エレメントの作成ではなく、主にインタラクティビティに使用します。コードが attachMovie 呼び出しを多用しているような場合、その FLA ファイルの構造を見直してください。
- AS は、出来るだけシンプルにしておく。
- スクリプティングされたアニメーションの使用は控えめにする。タイムライン アニメーションのパフォーマンスのほうが、通常は優れています。
- 過度の文字列操作は避けます。
- “if”ステートメントで終了を回避している、ループするムービー クリップは、むやみに使わない。
- on()または onClipEvent()イベント ハンドラの使用を避ける。代わりに、onEnterFrame、onPress などを使用する。
- フレーム上のプライマリ AS ロジックの配置は、最小限にする。その代わり、関数の内部に重要なコードをまとめて配置する。Flash AS コンパイラは、関数の本体内部に位置するソースに、フレームの内部に直接配置されたソースや、旧式のイベント ハンドラ (onClipEvent、on) と比較すると、はるかに高速のコードを生成することができます。ただし、タイムライン コントロール (gotoAndPlay、play、stop など) や他の重要ではないロジックなど、簡単なロジックをフレームに留めておくことは、許容範囲に入ります。
- いくつかのアニメーションを持つ長いタイムラインのムービークリップの中で、’ロングディスタンス’な gotoAndPlay/gotoAndStop を使うことを避けてください。

    - gotoAndPlay/gotoAndStop をフォーワードさせる場合、現在の場所から向こう側のターゲットフレームのうち、より高コストなものはタイムラインをコントロールする gotoAndPlay/gotoAndStop となります。従ってもっとも高価な gotoAndPlay/gotoAndStop のフォワードは最初のフレームから最後の フレームまで最大の高コストになります。
    - gotoAndPlay/gotoAndStop をバックさせる場合、タイムラインの開始からのターゲットフレームで より高いコストのものはタイムラインをコントロールします。従ってもっとも高価な gotoAndPlay/gotoAndStop のバックは、最後のフレームからその直前のものになります。
- 短いタイムラインでムービークリップを使用してください。gotoAndPlay/gotoAndStop のコストはキーフレームの数と タイムラインアニメーションの複雑性により依存しています。また、貴方が gotoAndPlay/gotoAndStop をコール することによってナビゲートすることを計画されているなら、長くて複雑なタイムラインを作成しないでください。その代わりに、短いタイムラインやより小さな gotoAndPlay/gotoAndStop のコールでそのいくつかの別々の ムービークリップに分けてください。
- 多くのオブジェクトを同時に更新する場合、これらのオブジェクトをグループとして更新できるシステムを開発することが不可欠です。C++側では、GFx::Movie::SetVariableArray 呼び出しを使って、C++から AS へ大量のデータを渡します。この呼び出しの後に、アップロードされた配列をベースにして、1 度に複数のオブジェクトを更新する 1 つの呼び出しが続きます。複数の呼び出しを 1 つの呼び出しにグループ化すると、通常、個別のオブジェクトごとにそれらを呼び出すよりも数倍速くなります。
- 1 つのフレームであまりに多くの作業を実行しない。Scaleform が、ステージをレンダリングする時間がなくなり、ユーザーが遅いと感じる場合があります。代わりに、実行する作業の量を細かく分けて、遅いと感じさせずに、Scaleform が所定のフレーム レートでステージを更新できるようにします。
- Object タイプを多用しない。
    - データ型の注釈は正確であるべきです。これはバグを識別するための、コンパイラの型チェックを許可するからです。Object タイプは、他の妥当な手段がない場合に限って使用してください。
- eval() 関数、または配列アクセス演算子の使用を避ける。多くの場合、ローカル参照は 1 度だけ設定するほうが好ましく、より効率的です。
- myArr.length 自体を使わずに、ループの前に Array.length を変数に割り当て、ループ条件として使用する。例:
- 以下のコードを使用します。

```
var fontArr:Array = TextField.getFontList();
var arrayLen:Number = fontArr.length;
for (var i:Number = 0; i < arrayLen; i++) {
    trace(fontArr[i]);
}
```

下記のコードは使用しません:

```
var fontArr:Array = TextField.getFontList();
for (var i:Number = 0; i < fontArr.length; i++) {
    trace(fontArr[i]);
}
```

- イベントを上手に、且つ具体的に管理する。イベント リスナーを呼び出す前に、そのリスナーが存在する (null ではない) かどうかをチェックする条件を使って、イベント リスナー配列をコンパクトにしておきます。
- オブジェクトへのリファレンスを解放する前に、removeListener() を呼び出して、オブジェクトからリスナーを明確に削除する。
- パッケージ名のレベル数を最小限にするとスタートアップ時間が短くなります。スタートアップ時に AS VM は、レベルあたりひとつのオブジェクトとなるオブジェクトのチェインを生成します。さらに、各レベルのオブジェクトを生成する前に、AS コンパイラはレベルがすでに生成されているかどうかを確認する「if」条件文を追加します。ゆえに、パッケージが「com.xxx.yyy.aaa.bbb」の場合、VM はオブジェクト「com」、「xxx」、「yyy」、「aaa」、「bbb」を生成するとともに、各生成前に、オブジェクトの存在を確認する「if」オペコードを追加します。このような深くネストされたオブジェクト／クラス／関数へのアクセスも性能的に遅くなるため、名前の解決には各レベルでのパース処理が必要です（「com」を解決し、次に「xxx」、次に「xxx」中の「yyy」、など）。このようなオーバーヘッドを避けるために、一部の Flash 開発者は、SWF のコンパイルの前にプリプロセッサ・ソフトウェアを用いて、c58923409876.functionName()のようにシングルレベルでユニークに識別できるパスを生成しています。
- アプリケーションが同じ AS クラスを使用する複数の SWF ファイルで構成されている場合、コンパイル中に選択する SWF ファイルからそのようなクラスを除外する。これによりランタイム メモリの必要サイズを削減できます。
- タイムラインのキーフレーム上の AS が完了までに時間を要する場合、そのコードを分割して、複数のキーフレーム上で実行するようにする。
- 最終的な SWF ファイルをパブリッシュするとき、コードから trace() ステートメントを削除する。これを行うには、[パブリッシュ設定] ダイアログの [Flash] タブで、[Trace アクションを省略] チェックボックスをオンにして、これらのステートメントをコメント アウトするか、削除します。これは、実行時にデバッグに使用されるすべての trace ステートメントを無効にする、効率的な方法です。
- 継承はメソッドの呼び出し数を増やし、さらにメモリを消費します。実行時には、必要とする機能をすべて含むクラスのほうが、スーパークラスからその機能の一部を継承するクラスよりも効率的です。従って、クラスの拡張性とコードの効率性の間で、設計トレードオフを行うことが必要な場合もあります。
- 1 つの SWF ファイルが、カスタムの AS クラス (例: foo.bar.CustomClass) を含む別の SWF ファイルをロードして、その SWF ファイルをその後アンロードすると、カスタムクラス定義がメモリに残ってしまいます。メモリを節約するため、アンロードされた SWF フ

ァイルのカスタム クラスは、明確に削除します。以下の例のように、delete ステートメントを使って、完全修飾クラス名を指定します:

```
delete foo.bar.CustomClass
```

- すべてのコードを毎フレーム実行する必要はありません。100%タイム クリティカルではないアイテムには、フリップ フロップ (コードが部分的にフレーム毎に切り替えられます) を使用します。
- onEnterFrames の使用はできるだけ控える。
- 数学関数を使わずに、データ テーブルを前もって計算しておく。
  - 大量に計算する場合、値を事前に計算し、その結果を変数の (擬似) 配列に保管することを検討してください。データ テーブルからそのような値を取るほうが、Scaleform にオンザフライでさせるよりもはるかに速くできます。

3.1.1 ループ

- ループの最適化と繰り返しの動作に集中する。
- 使用されるループの数と、各ループに含まれるコードの量を制限する。
- フレーム ベースのループは、必要がなくなり次第すぐに停止させる。
- ループ内から関数を何度も呼び出さない。
  - 小さな関数は、ループ内にその内容を含めるほうが効率的です。

3.1.2 関数

- 出来れば常に、関数を深くネストしない。
- 関数内で with ステートメントを使用しない。この演算子は最適化をオフにします。

3.1.3 変数 / プロパティ

- 存在しない変数、オブジェクト、または関数を参照しない。
- 出来れば常に、“var”キーワードを使用する。関数内で“var”キーワードを使用することは、特に重要です。ActionScript コンパイラは、ハッシュ テーブルに変数を置いて、そこに名前でアクセスするよりも、インデックスで直接アクセスできる内部のレジスタを使用して、ローカル変数へのアクセスを最適化するからです。
- ローカル変数で十分な場合は、クラス変数やグローバル変数を使用しない。

- グローバル変数の使用は抑える。グローバル変数は、定義していたムービー クリップが削除された場合、ガーベージ コレクトされません。
- 必要のなくなった変数は、削除する、または null に設定する。こうすると、ガーベージ コレクションのためにそのデータがマークされます。変数を削除すると、不要なアセットが SWF ファイルから削除されるので、ランタイム中のメモリ使用を最適化できます。変数を null に設定するよりは、削除するほうが適切です。
- 常に直接プロパティにアクセスするようにする。他のメソッド呼び出しよりもオーバーヘッドが多い、AS の getter メソッドや setter メソッドは、使わないようにする。

3.2 *ActionScript 3* の汎用ガイドライン

AS3 の実行性能を高めるには以下の最適化を行ってください。

3.2.1データ タイプの厳格な指定

- 変数またはクラス メンバーのデータ タイプは必ず宣言してください。
- Object タイプの変数またはクラス メンバーは宣言しないでください。Object は AS3 においてもっとも汎用的なデータ タイプです。Object タイプの変数宣言は、結局は何も宣言しないのと同じです。
- Array クラスは使わないようにしてください。代わりに Vector クラスを使用してください。Array は疎のデータ構造を持っています。インデックス 4294967295（またはクローズ）へのアクセスまたは値の設定を必要としない限り、Array を使う必要はありません。
- エレメントのタイプは Vector を使って指定します。Vector<*>タイプはこのベクターのインスタンスに含まれるいかなるタイプのデータも格納できることを意味します。Vector<*>のような汎用タイプは使用しないでください。変わりに Vector のエレメントとして特定タイプを使用してください。たとえば、Vector<int>、Vector<String>、あるいは Vector<YourFavoriteClass>は、Vector<*>に比べてメモリの使用量が少なく性能も良好です。
- Object をハッシュ テーブルとして使用してはなりません。代わりに flash.utils.Dictionary クラスを使用してください。
- ダイナミック クラス（およびダイナミック属性）の使用は避けてください。ダイナミック属性へのアクセスは、現時点で GFx からも Flash からも最適化されていません。
- オブジェクトのプロトタイプの使用および変更は避けてください。プロトタイプは AS2 の一部です。AS3 で互換性は維持されていますが、AS3 ではスタティック関数およびスタティック メンバーを使って同じ機能を実現できます。

3.3 *Advance*

Advance の実行に時間がかかり過ぎる場合、可能な最適化の手段が 6 つあります:

1. AS コードをフレーム毎に実行しない。また、onEnterFrame/Event.ENTER_FRAME ハンドラは各フレームでコードを呼び出すので、使わない。

2. イベントドリブンなプログラミング手法を使って、変更されたときだけ、明確な Invoke 通知によって、テキスト フィールドと UI のステート値を変更します。

3. 表示されないムービー クリップのアニメーションを停止する (_visible(AS2) or visible (AS3)プロパティを true に設定する必要があります。stop()関数を使ってアニメーションを停止します)。これにより、Advance リストから停止しているムービー クリップが除外されます。親ムービー クリップが停止していたとしても、子ムービー クリップを含め、階層内の 1 つ 1 つのムービー クリップを「**どれも**」停止する必要があるので注意してください。

4. _global.noInvisibleAdvance(AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)拡張機能の使用が関係する技術が他にもあります。この拡張機能は、表示されないムービー クリップをそれぞれ停止するのではなく、Advance リストからそのようなムービー クリップのグループを除外するのに使えます。この拡張機能のプロパティが "true" に設定されている場合、表示されていないムービー クリップは Advance リストに追加されないので (子ムービー クリップも含みます)、パフォーマンスが向上します。この技術は、完全に Flash と互換性があるわけではないので注意してください。Flash ファイルが、非表示のムービー内のフレーム単位の処理タイプに一切依存していないことを確認します。この拡張機能 (その他も含む) を使用するために、_global.gfxExtensions(AS2) or scaleform.gfx.Extensions.enabled (AS3)を true に設定して、Scaleform 拡張機能をオンにすることを忘れないでください。

5. ステージ上のムービー クリップの数を削減する。ムービー クリップの不要なネスト化を制限します。ネストされた各クリップは、Advance 中に若干のオーバーヘッドを課します。

6. タイムライン アニメーション、キーフレームの数、シェイプ トゥイーンを削減する。

3.4 *onEnterFrame and Event.ENTER_FRAME*

onEnterFrame(AS2) or Event.ENTER_FRAME (AS3)イベント ハンドラの使用は、最小限にします。または、常に実行させずに、必要なときだけインストールしてすぐ削除するようにします。あまりに多くの onEnterFrame/Event.ENTER_FRAME ハンドラを持つと、パフォーマンスが著しく低下する場合があります。別の方法として、setInterval と setTimeout 関数を使用することが考えられます。setInterval を使用するとき:

- このハンドラが必要なくなったら、clearInterval を忘れずに呼び出す。
- setInterval と setTimeout ハンドラが onEnterFrame/Event.ENTER_FRAME よりも頻繁に実行されると、これらのハンドラは onEnterFrame/Event.ENTER_FRAME よりも遅くなる場合があります。これを避けるには、インターバル時間に、控えめの値を使用してください。

onEnterFrame ハンドラを削除するには、delete 演算子を使用します:

```
delete this.onEnterFrame;
delete mc.onEnterFrame;
```

onEnterFrame に null や undefined を割り当てないでください (例、this.onEnterFrame = null;)。この操作は onEnterFrame ハンドラを完全に削除しないからです。onEnterFrame という名前を持つメンバーがまだ存在することになるので、Scaleform はこのハンドラを解消しようとします。

3.5 *ActionScript 2* の最適化

3.5.1onClipEvent イベントと on イベント

onClipEvent() や on() イベントの使用を避け、onEnterFrame、onPress などを使う。これには以下のような理由があります:

- 関数スタイルのイベント ハンドラは、実行時にインストールや削除が可能です。
- 関数内のバイトコードは、旧式の onClipEvent や on ハンドラ内よりも、最適化に優れています。主な最適化は、this、_global、arguments、super などをプリキャッシュするときや、ローカル変数に 256 の内部レジスタを使用するときです。 この最適化は関数のみに作用します。

唯一の問題は、最初のフレームの実行前に、onLoad 関数スタイル ハンドラをインストールする必要がある時です。この場合、記載されていないイベント ハンドラ onClipEvent(construct) を使って onEnterFrame をインストールすることができます:

```
onClipEvent(construct)
{
      this.onLoad = function()
      {
            //関数の本体
      }
}
```

または、onClipEvent(load) を使って、そこから正規関数を呼び出します。ただし、追加で関数を呼び出すためのオーバーヘッドが増えるので、この方法は効率が悪くなります。

3.5.2 Var キーワード

出来れば常に、var キーワードを使用します。関数内部で使うことが特に重要です。AS コンパイラは、ハッシュ テーブルに変数を置いて、その名前でアクセスするよりも、インデックスで直接アクセスできる内部のレジスタを使用することで、ローカル変数へのアクセスを最適化するからです。var キーワードを使用すると、AS 関数の実行速度を 2 倍にすることができます。

最適化されていないコード:

```
var i = 1000;
countIt = function()
{
      num = 0;
      for(j=0; j<i; j++)
      {
            j++;
            num += Math.random();
            }
      displayNumber.text = num;
}
```

最適化されたコード:

```
var i = 1000;
countIt = function()
{
      var num = 0;
      var ii = i;
      for(var j=0; j<ii; j++)
      {
            j++;
            num += Math.random();
            }
      displayNumber.text = num;
}
```

3.5.3 プリキャッシュ

頻繁にアクセスされる読み取り専用のオブジェクト メンバーを、ローカル変数に (var キーワードを使って) プリキャッシュします。

最適化されていないコード:

```
function foo(var obj:Object)
{
      for (var i = 0; i < obj.size; i++)
      {
            obj.value[i] = obj.num1 * obj.num2;
      }
}
```

最適化されたコード:

```
function foo(var obj:Object)
{
      var sz = obj.size;
      var n1 = obj.num1;
      var n2 = obj.num1;
      for (var i = 0; i < sz; i++)
      {
            obj.value[i] = n1*n2;
      }
}
```

プリキャッシュは、他のシナリオでも効率的に使用することができます。以下のような例があります:

```
var floor = Math.floor
var ceil = Math.ceil
num = floor(x) - ceil(y);
```

```
var keyDown = Key.isDown;
var keyLeft = Key.LEFT;
if (keyDown(keyLeft))
{
     //ここに何かを追加する;
}
```

3.5.4 長いパスのプリキャッシュ

以下のような長いパスを、繰り返し使用しないようにします:

```
mc.ch1.hc3.djf3.jd9._x = 233;
mc.ch1.hc3.djf3._x = 455;
```

このファイル パスを部分的にローカル変数にプリキャッシュします:

```
var djf3 = mc.ch1.hc3.djf3;
djf3._x = 455;

var jd9 = djf3.jd9;
jd9._x = 223;
```

3.5.5 複雑な式

以下のような、複雑な C スタイルの式は避けます:

```
this[_global.mynames[queue]][_global.slots[i]].gosplash.text =
_global.MyStrings[queue];
```

このような式は細かく分割して、ローカル変数に中間データを保管します:

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
var slot_i = _global.slots[i];
_splqueue[slot_i].gosplash.text = _splstring;
```

上記のような分割した部分に複数のリファレンスが存在している場合、以下の説明は、とても重要です。下のループの例を参照してください:

```
for(i=0; i<3; i++)
{
     this[_global.mynames[queue]][_global.slots[i]].gosplash.text =
                                                    _global.MyStrings[queue];
     this[_global.mynames[queue]][_global.slots[i]].gosplash2.text =
                                                    _global.MyStrings[queue];
     this[_global.mynames[queue]][_global.slots[i]].gosplash2.textColor =
```

```
                                                                0x000000;
}
```

上記のループを改善すると、以下のようになります:

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
for (var i=0; i<3; i++)
{
      var slot_i = _global.slots[i];
      _splqueue[slot_i].gosplash.text = _splstring;
      _splqueue[slot_i].gosplash2.text = splstring;
      _splqueue[slot_i].gosplash2.textColor = 0x000000;
}
```

このコードはさらに最適化できます。可能な場合、配列の同じエレメントへの複数のリファレンスを排除します。対応済みのオブジェクトをローカル変数にプリキャッシュします:

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
for (var i=0; i<3; i++)
{
      var slot_i = _global.slots[i];
      var elem = _splqueue[slot_i];
      elem.gosplash.text = _splstring;
      var gspl2 = elem.gosplash2;
      gspl2.text = splstring;
      gspl2.textColor = 0x000000;
}
```

4　HUD の開発

以下の章では、ヘッズ アップ ディスプレイ (HUD) を作成しイテレートする場合に推奨するベスト プラクティスを大まかに紹介しています。以下のインプリメンテーションは、決して必須というわけではありませんが、Scaleform で HUD を作成するときに、パフォーマンスの向上と最適なメモリ使用を達成するための考え方とガイダンスとして目を通してください。

推奨事項は、複雑さと実装にかかる時間の昇順でリストアップしました。HUD の複数のイテレーションを作成する場合、方法 4.1 から始め、最終版に向けてイテレーションが作成されるにつれて、このリストをさらに進むことをお勧めします。これは、ゲーム内で操作できる HUD エレメントをすぐにプロトタイプ化するための優れた方法です。HUD やリソースの必要条件を改善していく中で、弊社の推奨事例を使って最適化してください。

非常に複雑で多階層の HUD を、非常に高性能、かつ極めて低いメモリという条件で開発している場合、C++による開発がいまだに非常に効率が良いので、最適の選択かもしれません。

4.1　複数の *SWF* ムービー ビュー

全体的に見て、このタイプの HUD は、開発とイテレートの両方を極めて迅速に行うことができます。これは純粋にアーティスト主導であり、非常に短期間により多くの効果や良質のグラフィック表現を可能にします。最適化以前のプロトタイプ化と反復設計には、最適ですが、メモリの使用量は増える場合があります。個々のムービー ビューは新規のプレーヤー インスタンスを作成し、このインスタンスが約 80K のメモリ オーバーヘッドを追加します。ここでのトレードオフは、より高速の HUD の実現と個々のムービー制御、対　メモリ使用量の増加、ということになります。メモリとパフォーマンスの問題　対　ダイナミックで多階層なシステムの柔軟性、ということを認識しておいてください。

複数の SWF ムービー ビューを使用すると、以下のような利点があります:

1. 別々のスレッド上で *Advance* を呼び出す機能。マルチスレッド HUD インターフェイスは、別々のスレッド上で各 Flash ムービーの実行、つまり Advance を許可します (レンダリングではなく、タイムライン、アニメーション、AS の実行、Flash 処理などの処理です)。これにより個別の Advance コントロールが可能になります。Flash を複数のムービーに分割すると、開発者は特定のエレメントを停止して Advance しないことや、異なる時間に別々のスレッドに Advance を呼び出すことが可能になります。HUD の一部のエレメントはより高レートで Advance することができます。あるいは動作を必要とするイベントがゲーム内で起きるまで、静的 HUD エレメントで Advance の呼び出しを、実際に停止させておくことができます。

   注意: Scaleform では、異なる GFx::Movie オブジェクトのために別々のスレッド上で Advance を呼び出すことができますが、各ムービー インスタンスはスレッドセーフではない

ので、明確に同期しなければ、Display を別のスレッド上で呼び出すことはできません。Input 呼び出しと Invoke 呼び出しも Advance との同期が必要です。

2. *render-to-texture* キャッシュングを利用する。これには HUD エレメントをテクスチャにレンダリングしてキャッシュしておき、必要なときだけ更新するということです。この機能は描画プリミティブの数を最小限に抑えますが、メモリ使用量は増加します。なぜなら、HUD エレメントと同サイズのテクスチャ メモリ バッファが必要だからです。パフォーマンスへの影響はプラスですが、メモリにはマイナスになる場合があります。このオプションは、エレメントがめったに更新されず、あまり複雑ではない場合に限って使用するようにしてください。

4.2 複数の *SWF* を含む *1* つのムービー ビュー

この方法は、複数の Flash ファイルを AS loadMovie コマンドを使って、1 つのフルスクリーン ムービー ビューにロードするということです。この手法の利点は、メモリ使用がより効率的になることと、Display パフォーマンスがわずかに向上することです。

複数の Flash ファイルを 1 つのムービー ビューにロードするときに、以下のガイドラインを推奨します。

1. まとめて非表示にできるオブジェクトを慎重にグループ化して、_visible = false(AS2) or visible = false (AS3)でマークする。また、_global.noInvisibleAdvance(AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)を true に設定しておき、Advance 処理のオーバーヘッドを最小限にします。**HUD を作成するときに可能な最も重要な事項の 1 つは：**非表示にできるオブジェクトをグループ化して、そのオブジェクト群を管理するために親を追加することです。_visible/visible = false を使って、オブジェクトが非表示になった後の処理を止めます (注意: これは可視性をコントロールするのでは**なく**、すでに表示されていないオブジェクト上での Advance の呼び出しを止めることです)。オブジェクトの特定のグループを非表示にすると、その Flash ファイルの内部では、実行ロジックは、非表示のエレメント上では呼び出されなくなります。HUD の特定の部分が表示/非表示される箇所 (ポーズ メニュー、マップ、ヘルスバーなど) では特に便利です。 また、HUD 全体を表示/非表示にするときには、開発者が Advance の呼び出しを全部一括で停止することも推奨します。_global.gfxExtensions(AS2) or scaleform.gfx.Extensions.enabled (AS3)を true に設定して、Scaleform 拡張機能をオンにすることを忘れないでください。
2. 複数の変数アップデートを、SetVariableArray と 1 つの Invoke を使ってアプリケーションからグループ化する。これは、いくつかの異なるコンポーネントを持ち、高度な複雑性を持つエレメントで (移動するエレメントがあるマップなどがそうです) 、アップデートを 1 つの呼び出しにグループ化すること、一方、マップ上のアイコンごとに個別に呼び出すことなどに関係するようなものであるときに、便利に使えます。SetVariableArray を呼び出してデータの配列 (マップ上の各エレメントの位置の更新情報など) を渡し、その後 1 つの Invoke を呼び出して処理し、すべて 1 回の関数の実行で、アイテムを移動するように設定されたそのデータ配列を使用します。ただし、アップデートするデータが少ししかない場合には、パフォーマンスに悪影響を及ぼすことがあるので、この方法は使用しないでください。

3. HUD エレメントの作成には、慎重に考慮してから onEnterFrame(AS2) or Event.ENTER_FRAME (AS3)を使う。
   HUD 内に onEnterFrame/Event.ENTER_FRAME を持つ複数のエレメントがあると、開発者が Advance を呼び出すたびに、その特定のエレメントに変更がなくとも、これらのエレメントで、（onEnterFrame が）実行されてしまいます。
4. ムービーのアニメーション フレーム レートは、ゲームのフレーム レートの半分に保ち、AS は必要なときだけ呼び出す。一般的なゲーム プログラミング パラダイムは、ゲーム エンジンでフレーム毎に tick を呼び出すことです。これは明らかに、Flash の使用に適していません。フレーム毎に AS Invoke を呼び出すことを避けるようにすることが肝要です。これで、メモリの使用量を削減しパフォーマンスを向上することにつながります。例えば、ゲームを 30–40 FPS で実行している場合、アニメーションの更新は、15–20 FPS 毎だけで行います。ただし、非常な高フレームレートで作成されていて、呼び出しに時間がかかり過ぎるアニメーションがある場合、HUD アニメーションにラグやジッタが出ることや、滑らかに見えなくなってしまう可能性もあります。
5. タイムライン アニメーションは可能な限り短くする。長いタイムライン アニメーションは、より多くのメモリを使用する傾向があります。しかし、あまりに短くし過ぎると、アニメーションにジッタが出る場合があるので、このことは慎重に管理しなければなりません。

4.3 *1* つのムービー ビュー

この方法は、複雑なマルチエレメント インターフェイス (レーダー画面など) を持った Flash の効率性を上げるために使えます。Flash と C++がエレメントのレンダリングにどのように使用されるかを、慎重に検証することが重要です。複数の Flash レイヤーは、それぞれが描画プリミティブとなり、パフォーマンスが低下します。以下のガイドラインに加え、上述の 4.2 章の内容も含めて、使ってください:

1. インターフェイス内部のエレメントの描画にはゲーム エンジンを使います。Flash は、境界線やフレームの描画に、テキストに Scaleform を使用します。HUD 内に高速で変化するエレメントが複数ある場合、Flash を使って静的なエレメントを描画し、よく変化するアイテムのレンダリングに C++を使用することができます。
2. C++をエレメントの配置に使って、Flash は、それらエレメントを描画します。この良い例はレーダー画面です。Flash で点のセットを 1 つ作成しますが、その位置の管理は Render::Renderer 内部で行い、RenderString 識別子を使ってこのエレメントにタグを付けます。C++エンジンを使って、レンダリングする前にエレメントを正確に再配置します。これにより、AS 更新時のオーバーヘッドの一部を避けられますが、かなり複雑で、追加のプログラミングが必要になります。

4.4 *1* つのムービー ビュー *(*上級編*)*

この方法ははるかに高度で時間がかかりますが、メモリをさらに節約することができます。HUD のイテレーションと作成がほぼ終わりに近づくまで、この方法は使用しないことをお勧めします。4.3 章

で説明した方法に加えて、以下の技術を検討してください:

1. HUD のグラフィックの変更箇所だけで Advance を呼び出す。または、1 つの Invoke ですべての更新を行うことを試みる。これは、HUD アニメーションがアップデートされているバックグラウンド レイヤーを 1 枚持つ単一のムービーで使用することができます。ここでは、バックグラウンドの上に、さらに複雑な HUD インターフェイス (テキストやプログレス バーを含むものなど) があり、そのようなインターフェイスはアニメーション化されず、Advance を呼び出しません。この良い例は、通常はアニメーション化されていないヘルス バーです。変化があるまでそのバーで Advance を呼び出す理由はありません。(例えば、最初のフレームを再生し、フリーズして、display を呼び出し、キャラクターの健康状態に変化があったときだけ Advance/invoke を呼び出します。) これは CPU オーバーヘッドが少なくより効率的にレンダリングしますが、管理ははるかに複雑です。
2. 頂点データにはカスタムの静的バッファ管理を使い、Render::Renderer をオーバーライドする。この方法は C++集約的なので、高度なスキルを持った C++ グラフィック プログラマに、かなり助けて貰うことが必要になるでしょう。Render::Renderer をオーバーライドするときに、異なる (カスタムの) ビデオ メモリ ベクター データ ストレージを使い、Scaleform システムの他の部分は全てオーバーライドして、HUD エレメントの管理には、動的バッファではなく、静的バッファを使うようにします。
3. 複数スレッド レンダリングを使って、Render::Renderer をオーバーライドする。これはおそらく、最も複雑な方法です。レンダラの書き換えと、ゲーム エンジンがレンダリングする HUD で別の Advance の呼び出しを必要とします。このレベルの複雑さのトレードオフは、大幅なパフォーマンス向上の可能性を秘めています。

   注意: 複数スレッド レンダリングは Scaleform –Unreal® Engine 3 インテグレーションに存在しますが、それでもこの方法の実装には、プログラミングの多大な努力を必要とします。

4.5 *Flash* を使用しないカスタムの *HUD* 作成

この処理は、特に複雑で時間がかかるものです。つまり、この HUD は純粋に C++でビットマップをベースにします。Scaleform と Flash は、HUD の作成とイテレートのプロセスを通して使用することができますが、Flash インターフェイスをビットマップに変換することで、最終バージョンでは削除されます。この時点では、メモリに 1 度 Scaleform Player が発生する以外に、Scaleform は HUD エレメントに対して Advance を行う、またはメモリを使用することはありません。

開発者次第なのですが、パフォーマンス クリティカルな HUD アイテムに対するカスタム調整の C++ レンダリングと、その他すべてに対する Scaleform レンダリングを組み合わせることができます。外部のカスタム レンダリングから最も恩恵を得られる領域は、多くのアイテムを伴うミニマップや持ち物（装備）/ステータス画面です。これらは、DP（描画プリミティブ）バッチ処理と効率的なマルチアイテム アップデートから最大限に最適化できる領域で、Scaleform が自動的に行うのは困難なものです。境界線、パネル、スタッツなどのその他の HUD エレメントや、アニメーションのポップアップは、そのまま Scaleform を使い、ボトルネックになる場合に限って置き換えることができます。

とはいえ、開発者には Scaleform のフォント/テキスト エンジンをやはり使うことをお勧めします。特に、Scaleform が作成したユーザー インターフェイスで使用されているものと同じ Flash のフォントとテキストのシステムを使って、C++から開発者がプログラム的にテキストを描画できる DrawText API が、Scaleform には含まれているためです。これは、HUD に 1 つ、残りのメニュー システムに 1 つという具合に、2 つの異なるフォント システムを持つ必要がないので、メモリの節約になるはずです。Scaleform のフォント/テキスト エンジンの詳細と、フォントとテキストの使用法のベスト プラクティスについては、Scaleform FAQ の "Font & Text" セクションと「Font and Text Configuration Overview（フォントとテキスト設定の概要）」/「DrawText API」を参照してください。

総括すると、HUD 上で作成しイテレートするときは、以下の点に留意してください:

- ムービー クリップの数は最小限にとどめる。どうしても必要な場合だけ、アイテムをネスト化する。
- 可能な限り、マスクは使用しない。使っても、1 つか 2 つだけにとどめる。詳細は、FAQ の "Graphics Rendering & Special Effects" セクションを参照してください。
- PC と Wii™ではマウスとキーボードを無効にする(他の家庭用ゲーム機はデフォルトですでに無効になっています):
  - GFx::Movie::SetMouseCursorCount(0);
  - 無効にしたら、入力を入れない。
- 一緒に表示/非表示されるムービー クリップ アイテムをグループ化する。HUD パネルには、noInvisibleAdvance (AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)を _visible(AS2) or visible (AS3) = false で使用する。
- アイテムが変更になったときだけ Invoke を呼び出す。2 つ以上のアイテムが常に一緒に変更される場合 (同じフレーム)、アイテムの変更を 1 つの Invoke にバッチ処理する。あまり更新されないアイテムを (フレーム毎に) 処理しない。
- ビットマップとグラデーションの使用を、必ず最適化する。詳細は、第 2.1 章、または FAQ の "Art & Assets" セクションを参照してください。

5　一般的な最適化のヒント

5.1 *Flash* のタイムライン

タイムラインのフレームとレイヤーは、Flash オーサリング環境の 2 つの重要な部分です。これらの領域は、アセットがどこに置かれているかを示し、ドキュメントがどのように動作するかを決定します。タイムラインとライブラリをどのようにセットアップして使用するるかで、FLA ファイル全体と、その全般的な使い勝手とパフォーマンスに影響が出てきます。

- フレームベースのループの使用は控えめにする。.フレームベースのアニメーションは、FPS に拘束されない時間ベースのモデルとは対照的に、アプリケーションのフレームレートに依存します。
- フレーム ベースのループは、必要がなくなり次第すぐに止める。
- 可能なであれば、複雑なコード ブロックは複数のフレームに割り当てる。
- 数百行のコードを伴うスクリプトは、何百ものフレームベースのトゥイーンを持つタイムラインと同じほど、プロセッサ インテンシブです。
- コンテンツを評価して、アニメーション/操作がタイムラインで簡単に達成できるか、または AS を使って簡素化しモジュール化できるかどうかを判断する。
- デフォルトのレイヤー名 ([レイヤー 1]、[レイヤー 2] など) は使用しない。複雑なファイルで作業している場合、アセットの記憶や、検索で混乱する場合があります。

5.2 全般的なパフォーマンスの最適化

- パフォーマンスを向上するために、変形を組み合わせる方法はいくつもあります。例えば、3 つの変形をネストするのではなく、1 つのマトリックスを手動で計算します。
- 時間の経過でスローダウンが起きる場合、メモリ リークをチェックする。要らなくなったものは、必ず廃棄します。
- パフォーマンスが低下するので、オーサリングのときに、多くの trace() ステートメントや、テキスト フィールドのダイナミックな更新は避ける。できるだけ更新の頻度を低くする (つまり、絶えず更新するのではなく、何か変更があったときだけにします)。
- 出来れば、AS を含むレイヤーとフレーム ラベルのレイヤーを、タイムラインのレイヤースタックの最上部に置く。例えば、AS を含むレイヤーに「actions」という名前を付けるのは、良い習慣です。
- フレームの動作を異なるレイヤーに置くのではなく、すべての動作を 1 つのレイヤーに集約する。これにより、AS コードの管理が楽になり、複数回 AS を実行することで課されるオーバーヘッドがなくなるので、パフォーマンスが向上します。

6　その他の資料

ActionScript 2.0 Best Practices
http://www.adobe.com/devnet/flash/articles/as_bestpractices.html

Flash 8 Best Practices
http://www.adobe.com/devnet/flash/articles/flash8_bestpractices.html

Flash ActionScript 2.0 Learning Guide
http://www.adobe.com/devnet/flash/articles/actionscript_guide.html