

# Autodesk® Scaleform®

## Font and Text Configuration Overview

本書では、Scaleform 4.4 で使用するフォントおよびテキスト レンダリング システムについて記述します。多国語化 (国際化) で必要となる、アートアセットと Scaleform、 C++ API の設定方法を詳しく説明しています。

著者: Maxim Shemanarev, Michael Antonov  
バージョン: 2.2  
最終更新日: 2012 年 6 月 21 日

# Copyright Notice

## Autodesk® Scaleform® 4.4

© 2014 Autodesk, Inc. All rights reserved. Except as otherwise permitted by Autodesk, Inc., this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

The following are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and other countries: 123D, 3ds Max, Algor, Alias, AliasStudio, ATC, AutoCAD LT, AutoCAD, Autodesk, the Autodesk logo, Autodesk 123D, Autodesk CAM 360, Autodesk Homestyler, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSketch, AutoSnap, AutoTrack, Backburner, Backdraft, Beast, BIM 360, Burn, Buzzsaw, CADmep, CAiCE, CAMduct, CFdesign, Civil 3D, Cleaner, Combustion, Communication Specification, Configurator 360™, Constructware, Content Explorer, Creative Bridge, Dancing Baby (image), DesignCenter, DesignKids, DesignStudio, Discreet, DWF, DWG, DWG (design/logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DWGX, DXF, Ecotect, ESTmep, Evolver, FABmep, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, ForceEffect, FormIt, Freewheel, Fusion 360, Glue, Green Building Studio, Heidi, Homestyler, HumanIK, i-drop, ImageModeler, Incinerator, Inferno, InfraWorks, InfraWorks 360, Instructables, Instructables (stylized robot design/logo), Inventor, Inventor HSM, Inventor LT, Kynapse, Kynogon, LandXplorer, Lustre, MatchMover, Maya, Maya LT, Mechanical Desktop, MIMI, Mockup 360, Moldflow Plastics Advisers, Moldflow Plastics Insight, Moldflow, Moondust, MotionBuilder, Movimento, MPA (design/logo), MPA, MPI (design/logo), MPX (design/logo), MPX, Mudbox, Navisworks, ObjectARX, ObjectDBX, Opticore, Pipeplus, Pixlr, Pixlr-o-matic, Productstream, Publisher 360, RasterDWG, RealDWG, ReCap, ReCap 360, Remote, Revit LT, Revit, RiverCAD, Robot, Scaleform, Showcase, Showcase 360 ShowMotion, Sim 360, SketchBook, Smoke, Socialcam, Softimage, Sparks, SteeringWheels, Stitcher, Stone, StormNET, TinkerBox, ToolClip, Topobase, Toxik, TrustedDWG, T-Splines, ViewCube, Visual LISP, Visual, VRED, Wire, Wiretap, WiretapCentral, XSI.

All other brand names, product names or trademarks belong to their respective holders.

### Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Autodesk Scaleform の連絡先:

---

ドキュメント	フォントおよびテキスト設定の概要(Font and Text Configuration Overview)
住所	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
ホームページ	<a href="http://www.scaleform.com">www.scaleform.com</a>
電子メール	<a href="mailto:info@scaleform.com">info@scaleform.com</a>
電話	(301) 446-3200
FAX	(301) 446-3199

# 目次

<b>1</b>	<b>はじめに: Scaleform のフォント</b>	1
1.1	Flash のテキストとフォントの基礎	2
1.1.1	テキストフィールド	2
1.1.2	文字の埋め込み	3
1.1.3	埋め込みフォントによるメモリの使用	4
1.1.4	フォントで使用するメモリの調整	5
1.2	ゲーム UI のフォントに関する決定	6
<b>2</b>	<b>パート 1:ゲームフォントライブラリの作成</b>	8
2.1	フォントシンボル	8
2.1.1	フォントシンボルの書き出しと読み込み	9
2.1.2	書き出されたフォントと文字の埋め込み	11
2.2	gfxfontlib.swf を作成する手順	11
<b>3</b>	<b>パート 2:多国語化のアプローチの選択</b>	13
3.1	読み込みフォントの置き換え	13
3.1.1	多国語のテキストの設定	15
3.1.2	Scaleform Player 上での多国語化	17
3.1.3	デバイスフォントエミュレーション	18
3.1.4	フォントライブラリ作成の詳細手順	19
3.2	カスタムアセットの生成	21
<b>4</b>	<b>パート 3:フォントソースの設定</b>	23
4.1	フォントの参照順	23
4.2	Gfx::FontMap	25
4.3	Gfx::FontLib	26
4.4	Gfx::FontProviderWin32	26
4.4.1	ネイティブにヒントの付いたテキストを使う	27
4.4.2	ネイティブヒンティングの設定	28
4.5	Gfx::FontProviderFT2	29
4.5.1	FreeType フォントのファイルへのマッピング	30
4.5.2	フォントのメモリへのマッピング	30
<b>5</b>	<b>パート 4:フォント レンダリングの設定</b>	33
5.1	フォントキャッシュマネージャ	34

5.2	ダイナミックフォントキャッシュの使用 .....	35
5.3	フォントのコンパクター、gfxexport の使用 .....	38
5.4	フォント テクスチャの前処理 - gfxexport .....	38
5.5	フォント グリフ パッカーの設定 .....	40
5.6	ベクトル化の制御 .....	42
<b>6</b>	<b>パート 5:テキストのフィルタエフェクトと ActionScript 拡張 .....</b>	<b>44</b>
6.1	フィルタの種類、使用可能なオプションと制約 .....	44
6.2	フィルタ品質 .....	46
6.3	フィルタのアニメーション .....	47
6.4	ActionScript からのフィルターの使用 .....	47

# 1 はじめに: Scaleform のフォント

Scaleform には、新しい柔軟なフォント対応システムが組み込まれ、高品位で変形も自在な HTML 形式のテキストを表示することも可能になりました。この新しいシステムでは、ローカライズ用のフォントへの置き換え、およびローカルに埋め込まれたテキスト、共有 GFX/SWF ファイル、オペレーティングシステムのフォントデータ、FreeType 2 ライブラリなど、多様なソースからフォントを取り込むことができます。フォントレンダリング品質も大幅に向上しており、開発者は、アニメーションのパフォーマンス / メモリ使用量 / テキストの読みやすさという要素間の調整を取ることが出来ます。

Scaleform フォント機能の多くは、Flash® Studio で提供される機能に依存しています。例えば、フォント文字セットを SWF ファイルに埋め込む機能やシステムフォントを利用する機能などがあります。

しかし、Flash Studio は、主として個別のファイルを開発するために構成されているので、残念ながらフォントのローカライズ機能は限られています。特に、Flash では埋め込みフォントや変換テーブルを、ファイル間で共有することは容易ではありません。こうした機能は、効率的なメモリの利用とゲームアセットの開発には不可欠です。さらに、Flash はファイルに埋め込まれていない多国語文字のフォントの置き換えをオペレーティングシステムによって操作します。これでは、システムフォントを利用できないゲームコンソールには対応できません。

この問題に対し、Scaleform では、翻訳可能なすべてのテキストの集中的なコールバックとして Gfx::Translator クラスを使用しています。

また、ダイナミックフォントマッピングには Gfx::FontLib クラスを使用することによって対処しています。フォント用キャッシングメカニズムの制御、ローカライズ作業用のフォント名置き換え、必要であればオペレーティングシステムと FreeType 2 フォントの参照、こういった各種追加インターフェイスも用意されています。

Scaleform フォントシステムを使いこなすには、Flash Studio と Scaleform ランタイムの両方のフォント機能を理解しておく必要があります。アーティストは、Flash の機能を理解することにより、統一感のある体裁を持ったコンテンツを創ることができます。望ましいクオリティで効率的なレンダリングを実現しながらも、メモリ使用量を最小限に保つこともできるのです。一方、Scaleform フォントランタイム オプションを理解していただければ、ターゲットのプラットフォームで品質、パフォーマンス、メモリ使用量の特性を最適化する設定を選択することができるようになります。

本書は、Flash と Scaleform にあまりなじみのない開発者が、ゲーム UI フォント設定の詳細について学習できるように構成されています。このセクションは、次のように構成されています：

## 「Flash のテキストとフォントの基礎」

Flash のテキストとフォントの基本的な使い方を説明します。Flash についてよくご存知の方は、「テキストフィールド (Text Field)」と「文字の埋め込み」という初めの二つの章を読む必要はないかもしれません。

## 「ゲーム UI のフォントに関する決定」

Flash でゲーム UI を作成するときに、開発者が決定しなければならないフォント関連の基本事項について説明しています。このセクションでは本書のその後の構成についても説明しているので、必ずお読みください。

## 1.1 Flash のテキストとフォントの基礎

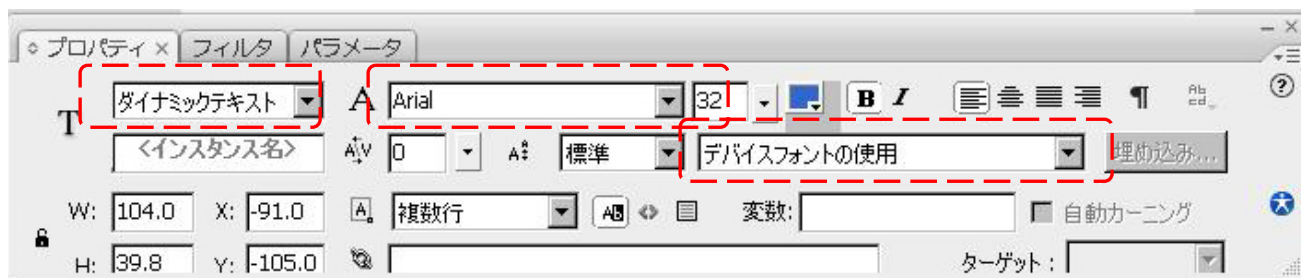
Flash Studio では、ステージ上にマウスでテキストフィールドを描いて、その中に対応するテキストを入力することによって、視覚的にテキストフィールドを作成します。テキストフィールドで使われるフォントは、アーティストのシステムにインストールされているフォントリストから選択されます。使いたいフォントを持たないシステムでコンテンツの再生をするには、埋め込みオプションを使います。

フォント名を直接指定するほか、ライブラリ内にフォントシンボルを作成し、シンボル名に基づいて間接的に使用することもできます。文字の埋め込みとフォントシンボルを併せて使えば、マルチプラットフォームで統一感を持つゲーム UI アセットを開発するための、強力な概念的フレームワークが構築できます。このセクションの以下の部分では、テキストフィールドの作成と文字の埋め込みの基礎について簡単に説明します。詳細は Flash Studio のオンラインマニュアルやヘルプを参照してください。

Flash のフォントシンボルシステムは便利で設定も容易です。しかし、制限事項も多いため、多国語化する場合のフォントの共有や置き換えは難しくなることに注意してください。こうした制限事項への GfX での対処法詳細については、「パート 1 - ゲームフォントライブラリの作成」で後述します。

### 1.1.1 テキストフィールド

Flash では、まずテキストツールを選択し、次にステージ上で相応の長方形の領域を描くことによってテキストフィールドを作成します。テキストフィールドのタイプ、フォント、サイズ、スタイルを含むテキストの様々な属性は、通常は Flash Studio の下部にあるテキストフィールドの [プロパティ] パネルで設定します。このパネルを下図に示します。



テキストフィールドには多くのオプションがありますが、上の図では本書の説明に必須の属性を赤い破線で囲んであります。ここで、最も重要なオプションは、左上にあるテキストのタイプです。これは次のいずれかの値に設定できます。

静置テキスト

ダイナミックテキスト

テキスト入力

ゲーム開発には、「ダイナミックテキスト」プロパティが最もよく使われます。これは、ActionScript によるコンテンツの変更ができること、ユーザがインストールした GfX::Translator クラスを介して、フォントの置き換えと多国語化をサポートしているからです。「静置テキスト」は、上記の機能を提供

していません。ベクトルアートと、機能的には非常に近いということになります。「テキスト入力」は、編集可能なテキストボックスが必要な場合に使用できます。

赤い破線で囲まれた他の2つのフィールドは、フォント名(プロパティシートの上の行)とフォントレンダリング方法です。フォント名は、a) 開発者のシステムにインストールされたフォント名、b) ムービーライブラリで作成したフォントシンボル名、の何れかを使うように選択できます。上の例では、フォント名として「Arial」が選択されています。[ボールド]と[イタリック]のトグルボタンを使用してフォントスタイルを設定することもできます。

多国語化を考慮する場合、フォントレンダリング方法が最も重要な設定事項となります。というのは、SWF ファイルへの文字の埋め込みが、これによって制御されることになるからです。Flash 8 では、次のいずれかの値に設定できます。

デバイスフォントの使用

ビットマップテキスト (アンチエイリアスなし)

アンチエイリアス (アニメーション優先)

アンチエイリアス (読みやすさ優先)

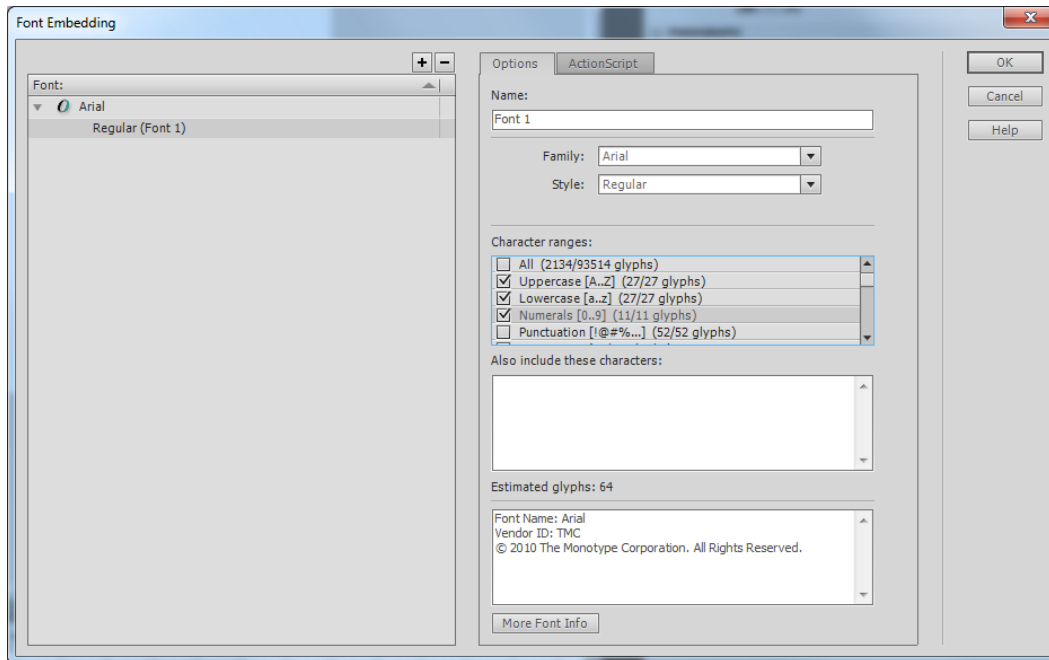
「デバイスフォントの使用」を選択すると、システムに固有のフォントレンダリング方法とオペレーティングシステムから取得したフォントデータが使用されます。デバイスフォントを使用する利点の1つは、SWF ファイルが小さくなるので再生時にシステムメモリの使用量が少ないことです。しかし、残念ながら、ターゲットシステムに必要なフォントがないと問題が発生することがあります。このような場合は、Flash Player が代替フォントを選択します。この代替フォントは、オリジナルのフォントと外観が異なっていたり、必要なグリフが不足したりすることがあります。例えば、オペレーティングシステムのフォントライブラリがないゲームコンソールでは、テキストが表示されません。Scaleform では、グリフがない場合は、四角形がレンダリングされます。

「アンチエイリアス (アニメーション優先)」と「アンチエイリアス(読みやすさ優先)」の設定では、システムフォントを使用せずに、埋め込みフォント文字に依存して、アニメーションのパフォーマンスを優先して、あるいは品質の高さを最適化して、文字をレンダリングします。このどちらかのオプションを選択すると、右にある[埋め込み...] ボタンが有効になり、フォントとして埋め込む文字の範囲を選択することができるようになります。

### 1.1.2 文字の埋め込み

組み込みフォントが正常に動作するには、ファイル内でキャラクターが指定のフォントに対して少なくとも1つの TextField で組み込まれている必要があります(フォントがエクスポートされている場合はtextfieldの必要はありません)。必要な文字が埋め込まれていない場合は、デバイスフォントが使用され、前に述べたようなあらゆる制限が適用されてしまいます。(Scaleform 上で、GFX::FontLib に依存していない場合。これは後述します。)フォント文字を埋め込む場合は[埋め込み...] ボタンを押してください。以下に示す[文字の埋め込み] ダイアログが表示されます。





この埋め込みダイアログでは、選択済みのフォントとスタイルから、テキストフィールドに埋め込みたい必要な文字群の範囲を決定することができます。例えば、「かな」だけ、とか「JIS 第一水準漢字」だけといった具合です。「追加する文字を指定する」のウィンドウに、個別の文字を入力して追加してゆくことも可能です。同じフォントスタイルを使用するすべてのテキストフィールドで同一の埋め込み文字を共有できますので、通常はいずれか一つのテキストフィールドに埋め込みを指定すれば十分です。

組み込みの観点からは、太字と斜体のフォント属性は別個に扱われることに注意してください（上のスクリーンショットの「Style」のコンボボックスを参照してください）。例えば、Arial と Arial Bold の両方のスタイルを使用すると、両方を別々に埋め込む必要があるので、ファイルのメモリ使用量が増大してしまいます。一方、フォントサイズの変更は、ファイルサイズやメモリのオーバーヘッドを増大させることはありません。フォントと文字の埋め込みに関する詳細は、Flash Studio のマニュアルの「Using Fonts (フォントの使用)」のセクションを参照してください。

マルチプラットフォームで同一のフォントを使用するには、「文字の埋め込み」の使用が唯一の方法となります。フォント文字を埋め込むと、そのベクトル化されたものが SWF/GFX ファイルに保存され、後に使用するときメモリにロードされることとなります。グリフデータは SWF/GFX ファイルの一部を構成していますので、使われるシステムにインストールされているフォントに関わらず、グリフは常に正しくレンダリングされます。Scaleform では、埋め込みフォントはゲームコンソール (Xbox 360、PS3、Wii など) でもデスクトップ PC (Windows、Mac、Linux など) でも同等に機能します。埋め込みフォントの使用による避けられない副作用として、ファイルサイズとメモリ使用量の増大があります。サイズはかなり増大する可能性があるので (特にアジア文字の場合)、ゲームで使用するフォントについて事前に計画し、可能な限りフォントを共有する必要があります。

### 1.1.3 埋め込みフォントによるメモリの使用

Scaleform では文字の埋め込みがメモリにどう影響するかを理解するために、次の表に埋め込み文字の数が増えた場合に使用されるメモリの概要を示し、これについて考察します。

埋め込み文字の数	圧縮されていない SWF の サイズ	Scaleform ランタイ ムサイズ
----------	-----------------------	------------------------

1 文字	1 KB	450 K
114 文字 – ラテン文字 + 句読点記号	12 KB	480 K
596 文字 – ラテン文字とキリル文字	70 KB	630 K
7,583 文字 – ラテン文字と日本語	2,089 KB	3,500 K
18,437 文字 – ラテン文字と繁体中国語	5,131 KB	8,000 K

このサンプルの表は、“Arial Unicode MS” フォントを埋め込んだ場合です。ここで見られるように、欧州文字セット組み込みの場合は他と比べてメモリを消費しません。500 文字追加してもメモリ使用量は約 150K 増えるだけです。従って、いくつものフォントスタイルを欧州言語のローカライズに使うことは可能です。しかしながら、アジア文字の場合は非常に多くのグリフが必要になるため、メモリ使用量は大きく増えてしまいます。

### 1.1.4 フォントで使用するメモリの調整

大規模な文字セットの埋め込みは数メガバイトのメモリを消費するため、フォントの使い方について事前に計画し、メモリ使用量を削減する必要があります。フォントで使用するメモリを抑えるには、次のようないくつかの方法があります：

1. UI アートアセット開発開始前に、使えるフォントとフォントスタイルを事前に指定したセットだけに限定しておく。ボールドとイタリックは Flash では別の文字セットとして埋め込まれてしまいます。そこで、必要な場合だけ使うようにしてください。一方、フォントサイズの変更はオーバーヘッドを生じないので、メモリを増大させずに使用することができます。将来は、ボールドとイタリックのフォント文字を個別に格納する必要のない見かけ上の(フェイク) ボールドとイタリックのオプションも提供する予定です。
2. GfX::FontLib の使用または読み込み（インポート）により、ファイル間でフォントを共有する。 ファイルの一つ一つに同じ文字群を埋め込んでゆくと、アセットのサイズ、メモリ使用量（後続のファイルも同時にロードする場合）、ロード時間が非常に増大することがあります。できれば、埋め込みフォントを個別の SWF/GFX ファイルに格納して共有することによって、メモリ内での重複や再ロードを回避するのが最善です。GfX::FontLib の使い方については本書で後述します。
3. アジア文字セットの場合は、使用する文字だけを埋め込む。 アジア向けにローカライズするゲームには、言語のすべての文字を埋め込まず、ゲームテキストで使用する文字だけを埋め込むようにします。翻訳後、ゲーム中のすべての文字列を走査して必要となる独自の文字セットを生成し、これを埋め込みます。IME による任意のダイナミックテキスト入力が必要としないゲームであれば、文字セットを限定することによってメモリが大幅に節約されます。
4. GfXEport フォント圧縮（-fc オプション）の使用を考慮してください。一般的に目立った質の劣化を招かずにサイズを 10～30%削減できます（セクション 5.3 参照）。

出来栄の良い多国語対応ゲームを開発するには；上述したように、使用するフォントの制限をすること、GfX::FontLib を利用してフォントの共有をすること等のテクニックを上手く組み合わせるようしてください。アジア言語の場合はゲーム中で使用される文字だけを埋め込み、IME が必要な場合のみフォント全体を埋め込みます。そうすることによって、メモリがかなり節約できます。

## 1.2 ゲーム UI のフォントに関する決定

Scaleform でゲーム UI を作成する場合は、フォントの使用、設定、多国語化のアプローチに関していくつもの決断を行なわなければなりません。これ以後、これらの決定事項を以下の 4 つのカテゴリに分けて説明していきます：

### パート 1: ゲームフォントライブラリの作成

フォントライブラリを作成する場合に、使用するフォントの数やスタイルを決定するために役立つ説明です。

### パート 2: 多国語化アプローチの選択

多国語対応のゲームアセットを作成するための様々なアプローチについて説明します。

### パート 3: フォントソースの設定

フォントの参照順、及び Scaleform で使用可能な様々なフォントソースの設定方法について説明します。

### パート 4: フォントレンダリングの設定

Scaleform テキストのレンダリングアプローチとその様々な設定オプションについて説明します。

ゲームのフォントライブラリを作成するということは、ゲームインターフェイスの基準となるフォントスタイルを先ず選択することになります。これは、通常ゲーム UI アセットを開発開始する前のステップです。アートの点では、すべての UI ファイル内で種類の異なるフォントが、どの場所でも同じ目的で使われるように整合性を持たせなければなりません。このために、標準となる一つのフォントライブラリを持つことが重要になってきます。技術的には、埋め込むフォントの数を抑えてアプリケーションのメモリの割り当て内に収まるようにすることが肝要です。

ゲームフォントが決まったら、次に多国語化する方法を決定することになります。パート 2 では、次に示す 3 つの多国語化のモデルについて説明します。

**読み込みフォントの置き換え** このモデルは、読み込みフォントと、Player にロードされ Gfx::FontLib を介して共有される各言語毎の個別フォントファイルに依存しています。

**デバイスフォントエミュレーション** 上の方法と似ていますが、読み込みフォントでなくデバイスフォントに依存します。適切なシステムフォントのサポートが基になっていなければなりません。(現在のゲームコンソールでは使用できません。)

**カスタムアセットの生成** Flash 組み込みの翻訳機能を使用して各ターゲット市場向けの SWF/GFX ファイルのカスタムバージョンを生成します。

開発者は、これらのアプローチのいずれかを選択するか、または上手く組み合わせて選択し、開発全体を通じて使用することができます。

フォントは、Scaleform 内の多くのソースから取得できるので、正しく設定することが非常に大切です。フォント設定に役立つように、「パート 3: フォントソースの設定」では、Scaleform のフォント参照プロセスについて、Gfx::FontLib およびフォントプロバイダを設定する方法の例を含めて詳細に説明してあります。フォントプロバイダは、システムフォントと FreeType 2 フォントのサポートに使用するユーザが設定可能なクラスです。Gfx::FontLib を使わずに、埋め込みを経ずにフォントにアクセスできます。

パート 4 の「フォントレンダリングの設定」では、Scaleform で使用するフォントレンダリングアプローチについて説明します。3 つのフォントテキストチャの初期化方法；すなわち gfxexport によるプリプロセス中のプリレンダリングテキストチャ；ロード時に生成されるスタティックテキストチャのパッキン

グ；およびオンデマンドで更新されるダイナミックテキストキャッシュの使用、について集中的に述べています。そこでは、各アプローチで使用可能な設定オプションについて説明し、開発者がアプリケーションとターゲットプラットフォームに適したアプローチを正しく選択できるようにしています。

## 2 パート 1:ゲームフォントライブラリの作成

アーティストが、ゲーム UI 制作を開始するにあたって最初の仕事の一つは、ゲーム全体を通じて使用する可能性のあるフォントスタイルのセットを、何れにするか決めなければなりません。

例えば、ゲーム中のタイトル部分には、一種類のフォントタイプだけを使用し、タイトル以外のテキストには別のフォントタイプ一つを使用するように指定できます。一度このように決めたなら、他のフォントタイプは、特別な理由のない限りゲームの UI 画面に使用すべきではありません。こうした制限をお薦めする主な理由として、下記の三点があげられます：

同じフォントを使用することにより、ゲーム内のあらゆる場面で、ユーザ向けの表示に一貫性が保証されます。ゲームの画面が変わるたびに異なるフォントを使用すると、読みにくく、わかりにくい無秩序な UI が作成されます。

多国語化すると、多くの場合、初期開発時のフォントをターゲット言語の文字を含む別のフォントに置き換える必要があります。固定のフォントセットが事前に指定されていれば、この作業は容易に行なえます。

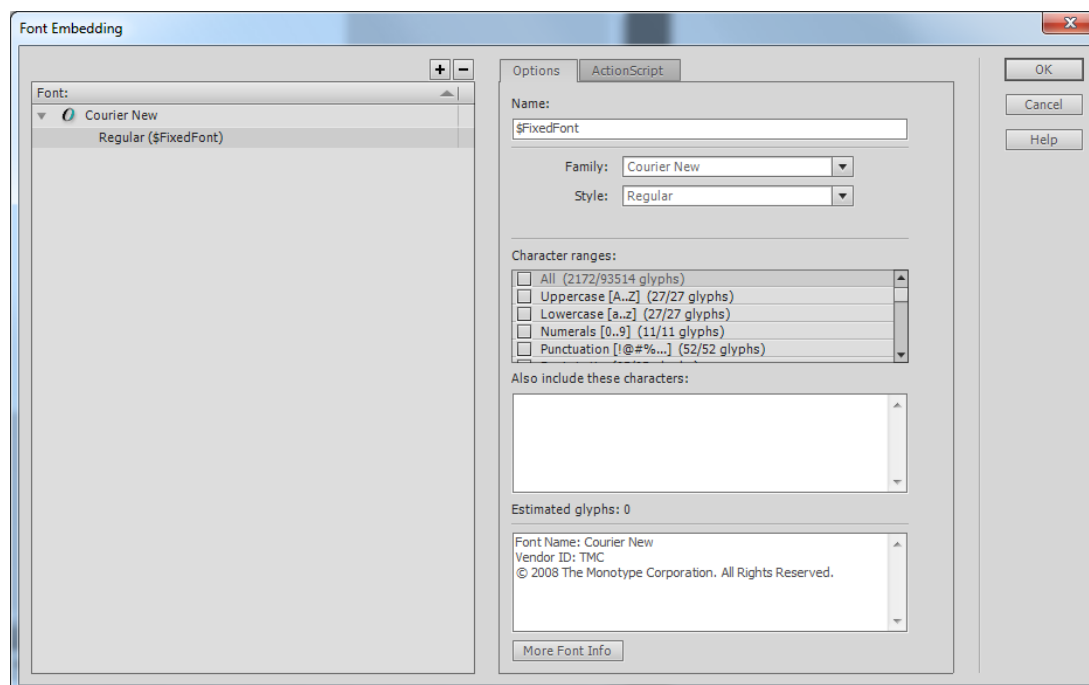
多くの UI 画面で固定のフォントセットを共有すると、メモリ内でフォントデータを共有できるので、メモリ使用量を大幅に削減し、画面のロード時間を大幅に短縮することができます。フォントデータは多くのメモリを占有する可能性があるので、これは非常に重要な技術的検討対象です。

次に述べる Scaleform でのインポートされたフォント置換のアプローチでは、ゲームに対するフォント選択プロセスは、言語ごとにファイル群から成るライブラリの生成となります

(`fonts_en.swf`、`fonts_kr.swf` など)。アーティストは、Flash ファイルライブラリ内に名前付きのフォントシンボルを作成し、それに対応する SWF に書き出すことによって、このファイルを作成できます。ライブラリファイルが作成されれば、そのフォントシンボルを別の Flash ファイルに読み込むことができるようになり、開発期間中を通して使うことができます。次のセクションでは、フォントシンボルの作成と、それを使ってゲームフォントライブラリを作成する方法について詳しく説明します。

### 2.1 フォントシンボル

「はじめに」で、テキストフィールドにフォントを適用する方法と、その文字を埋め込んで他プラットフォームでも再生を可能にする方法について説明しました。アーティストは、テキストフィールドのプロパティで、システムフォントを使用するだけでなく、Flash Studio を使用して、新たなフォントシンボルを定義することができます。ウィンドウ (W) のプルダウンメニューで、ライブラリ (L) を選択して、ライブラリウィンドウを開きます。ライブラリの空欄で右クリックして、[新しいフォント...] を選択すると、次のダイアログが表示されます。



こうして作成されたフォントシンボルが FLA ファイルのライブラリに追加され、後で標準のシステムフォントと同様にテキストフィールドに適用できます。

例えば、作成したゲームフォントに“\$FixedFont”という名前を付けておけば、あなたの作ったテキストフィールドに使用可能なフォントとして、フォントリストにこの名前が表示され、選択できるようになります。

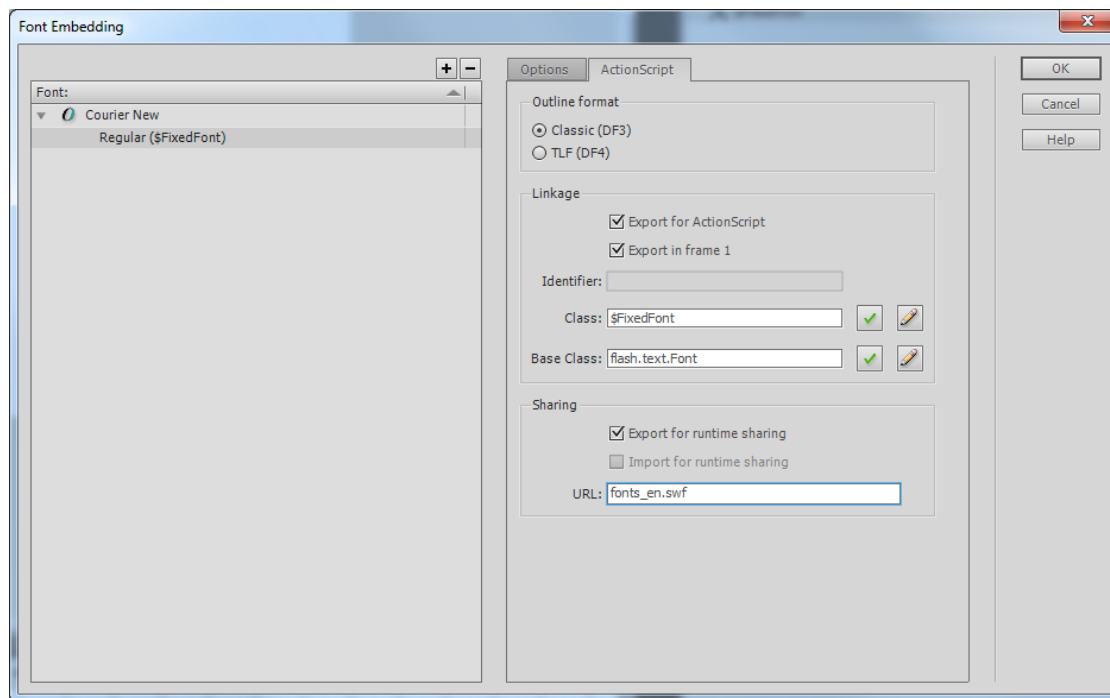
ライブラリにフォント名を追加すると、独自に使用するフォントを特定できるという利点があります。ライブラリフォントが、フォントリストに表示される場合は、“\$ArialGame\*”のように横にアスタリスクが付いています。アーティストがフォント名だけでライブラリを作成したような場合に、UI ファイルに誤って他のフォントを使ったりしないようにするのに役立ちます。さらに、すべてのフォントシンボルにドル記号 '\$' で始まる名前を付けておけば、テキストフィールドのプロパティでは、フォントリストの常に一番上に表示され、非常に便利です。

多くの Web サイトでは、フォントシンボル名のプレフィックスとしてアンダースコア '\_' の使用を推奨しています。アンダースコアはすっきり見えますが、テキストフィールド-プロパティの[フォントレンダリング方法]の選択ボックスが使えなくなってしまうます。これは、Flash Studio が、アンダースコアで始まるフォント名を組み込みフォントとして扱っているためです。そこで、この問題を回避するために、私どもの説明では、代わりに\$記号を使用しています。

### 2.1.1 フォントシンボルの書き出しと読み込み

FLA ファイル内のフォントシンボルは、他のライブラリエントリとほぼ同じく、そのコンテンツをコピーしたり、または読み込み/書き出しメカニズムを使えば、他のファイル内でも使用できます。ライブラリシンボルをコピーするには、ソース FLA ライブラリ内で右クリックして [コピー] を選択した後、ターゲット FLA ファイルに切り替えて右クリックし、[ペースト] を選択します。これで、シンボルのコピーが作成され、そのデータ コンテンツと一緒に、すべてターゲットの FLA ファイルに複製されます。

データの重複を回避するには、Flash では読み込み/書き出しメカニズムが使用できます。ライブラリのシンボルをエクスポートするには、その上で右クリックして、"Properties"を選択してから "ActionScript"タブをクリックします。そうすると次のプロパティシートが表示されます。



"OK"ボタンをクリックすると Flash はクラスの定義が無いことを警告しますが、この警告は無視してください。

シンボルをエクスポートするには、これにエクスポート識別子を付けて「ランタイム シェアリングのためにエクスポート」、「ActionScript 用にエクスポート」、「最初のフレームでエクスポート」のフラグを立てます。読み込み識別子はシンボル名と同一にしておくことをお勧めします。URL は、このシンボルがインポートされる時に使用される SWF ファイルへのパスを指定する必要があります。このフォントシンボルが GfX::FontLib オブジェクトで替えられる場合には、この URL はユーザーのデフォルトのフォントライブラリを指定する必要があります。このマニュアルではデフォルトの名前として "fonts\_en.swf"を使用しますが、他のどのような名前でも構いません。**GfX のデフォルトでのフォントライブラリがフォントの置換を使用するように設定する必要があります。**

たとえば、次のようにしてください。

```
Loader.SetDefaultFontLibName("fonts_en.swf");
```

GfXPlayer のデフォルトのフォントライブラリは、次のように fonconfig.txt で設定できます。

```
fontlib "fonts_en.swf"
```

最初に現れる fontlib がデフォルトライブラリとして使用されます。

シンボルが書き出されると、ライブラリの「リンケージ」列に“書き出し: \$識別子”として記載されます。読み込まれたライブラリシンボルにコピー/ペーストまたは、ドラッグアンドドロップ操作が適用されると、そのコピーは作成されなくなりますが、ターゲットファイル内に読み込みリンクが作成されます。つまり、ターゲットファイルは小さくなり、メモリにロードされたときにソース SWF から読み込みデータを取得します。Scaleform では、読み込まれた SWF データは複数のファイルから読み込ま

れた場合にも 一度だけしかロードされませんので、システムメモリが基本的に大幅に節約されることになります。

フォントシンボル名は、TextField.htmlText 文字列の一部としても、TextFormat 内のフォント名としても返されず、オリジナルのシステムフォント名 (“Arial” など) が返されます。同様に、シンボル名は読み込み名と合致していない限り、ActionScript を使用して、書き出し「フォント」に割り当てることはできません (この手法は、お勧めできません)。

### 2.1.2 書き出されたフォントと文字の埋め込み

Flash Studio ではバージョン CS4 までは、エクスポートされるフォントにどの文字群を組み込むかを指定できませんでした。その代わり、組み込む文字セットはシステムのロケール設定で決定されていました。Windows では、これは、コントロールパネルの [地域と言語のオプション] > [詳細設定] > [Unicode 対応でないプログラムの言語] で制御されます。この言語が “English” に設定されている場合は、フォントごとに 243 個のグリフしか書き出されません。韓国語を選択すると、11,920 個の文字が書き出されます。これはゲームの開発には便利ではなかったため、“gfxfontlib.swf”を使用するアプローチを新たに作り直しました。

幸い Flash の CS5 以降はどのグリフをエクスポートするかを指定できるようになったので、“gfxfontlib.swf”はもう必要ありません。これに代わって、今は言語固有のフォントライブラリ (fonts\_en.swf、fonts\_jp.swf など) を作成して、その各々でどのグリフを組み込むべきかを直接指定できます。

しかし、Flash Studio の CS4 (またはそれ以前のバージョン) がまだ使用されている場合は、“gfxfontlib.swf”を使用するアプローチを使用する必要があります (まだサポートされていますが、現在のバージョンでは推奨しません)。このアプローチに関して詳細は GfX 4.0 の古いバージョンのマニュアルや、以下の詳細情報を参照してください。

## 2.2 gfxfontlib.swf を作成する手順

フォントライブラリに関する理解が出来たところで、フォントライブラリのファイルを作成できます。繰り返しますが、これらのファイルは言語固有のソースフォントを提供するために作成されます。ライブラリフォントがどのようににマッピングされ、代替されるかの詳細は、次のセクションで取り上げます。

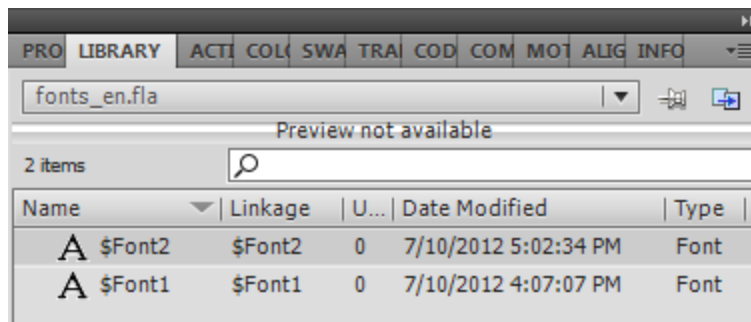
特定の言語に対するライブラリファイルを作成するには、Flash Studio で新規 FLA を作成し、そのライブラリをフォントシンボルで満たします。具体的には、次の手順を実行してください。

1. ゲームのすべての画面で使用するフォントを決定し、それぞれの目的に基づいて固有の名前を付けてください。ゲームのタイトルに使用するフォントを ‘\$TitleFont’、標準サイズのフォントを ‘\$NormalFont’ などとするのがよいでしょう。
2. ライブラリ用の新しい FLA ファイルを作成します。
3. 手順 1 で決定した各フォント毎に、新しいフォントシンボルを作成します。



4. 読み込みフォントの置き換えを行いたい場合には、フォントごとにフォントシンボル名と同じ識別子で書き出されるように [リンケージプロパティ] を設定します。
5. どのグリフを組み込むかを指定します。
6. その結果出来るファイルを fonts\_<lang>.swf として保存します。ここで lang は言語を表します（例えば 'en' は英語、'ru' はロシア語、'jp' は日本語、'cn' は中国語、'kr' は韓国語などです）。名前は何でも構いませんが、この名前は後ほど FontMap の設定で必要になります。

文書作成のために他のテキストフィールドの使用が必要でなければ、このフォントライブラリのステージにテキストフィールドを加える必要はありません。。フォント以外、フォント ライブラリファイルには、如何なるシンボルタイプも追加してはいけません。FLA が出来上がると、ライブラリは、次のように表示されているはずです。



フォントライブラリが完成しましたので、これから制作するアプリケーションのユーザインターフェイス画面で、シンボルを使用できるようになっています。読み込みファイル上でライブラリフォントを使うようにするには、ターゲットムービーのステージにフォントシンボルをドロップするか、または前述のコピー/ペースト手法を使うことができます。

## 3 パート 2:多国語化のアプローチの選択

UI を多国語化するときには、テキストフィールドの文字列とフォントという、二つの主要なエンティティを置き換える必要があります。テキストフィールドの文字列の置き換えは言語変換を目的としており、開発時のテキストがターゲット言語の対応するフレーズに置き換えられます。フォントの置き換えは、ターゲット言語に適した文字セットの提供を目的としています。オリジナルの開発フォントに、ターゲット言語で必要となる文字の全てが含まれているとは限らないからです。

ここでは、アートアセットの多国語化に利用できる、次の三つのアプローチについて説明します。

7. 読み込みフォントの置き換え
8. デバイスフォントエミュレーション
9. カスタムアセットの生成

「読み込みフォントの置き換え」は、フォントを多国語化する場合に推奨されるアプローチです。GfX::FontLib オブジェクトと GfX::FontMap オブジェクトに依って、基は'fonts\_en.swf'ライブラリで提供されていたフォントシンボルを置き換えます。

「デバイスフォントエミュレーション」は、読み込みフォントの置き換えと似てはいますが、読み込まれたシンボルでなくフォントマッピングに依って実際のフォントを提供します。設定は若干容易ですが、いくつかの制限があります。

「読み込みフォントの置き換え」と「デバイスフォントエミュレーション」は、いずれもテキスト文字列を翻訳するためにユーザが作成した GfX::Translator オブジェクトに依存します。

「カスタムアセットの生成」は、前述のアプローチとは異なります。フォントマッピングや GfX::Translator オブジェクトを使用せず、Flash Studio の機能によってターゲット言語ごとにカスタム UI アセットファイルを生成します。このアプローチは、メモリが極端に制限されているプラットフォームで、数少ないスタティックな UI アセットしかないような場合に使えるでしょう。

### 3.1 読み込みフォントの置き換え

読み込みフォントの置き換えは、テキストフィールドを置き換え可能なフォントに結合する Flash のフォントシンボルの読み込み/書き出しメカニズムに依存しています。この方法を使うには、例えばパート 1 で説明した fonts\_en.swf などの名前を付けたデフォルトのフォントライブラリファイルをまず作成し、このファイルからエクスポートされたフォントシンボルを全てのゲーム UI ファイルで使用します。次に、このファイルから書き出されたフォントシンボルを、すべてのゲーム UI ファイルで使用します。UI ファイルを Adobe Flash Player でテストすると、“fonts\_en.swf” からフォントが読み込まれ、初期開発言語でコンテンツを正しくレンダリングすることができるはずです。これらのアセットを Scaleform で実行すれば、多国語設定がロードされれば、翻訳とフォントの置き換えが可能になることになります。

Loader::SetDefaultFontLibName (const char\* filename)を呼び出す C++の方法をファイル名（パスなしでファイル名のみ）を'filename'パラメーターとしてパスしてください。例えば、デフォルトの開発言語が韓国語の場合、次のように呼び出しをします。

```
Loader.SetDefaultFontLibName( "fonts_kr.swf" );
```

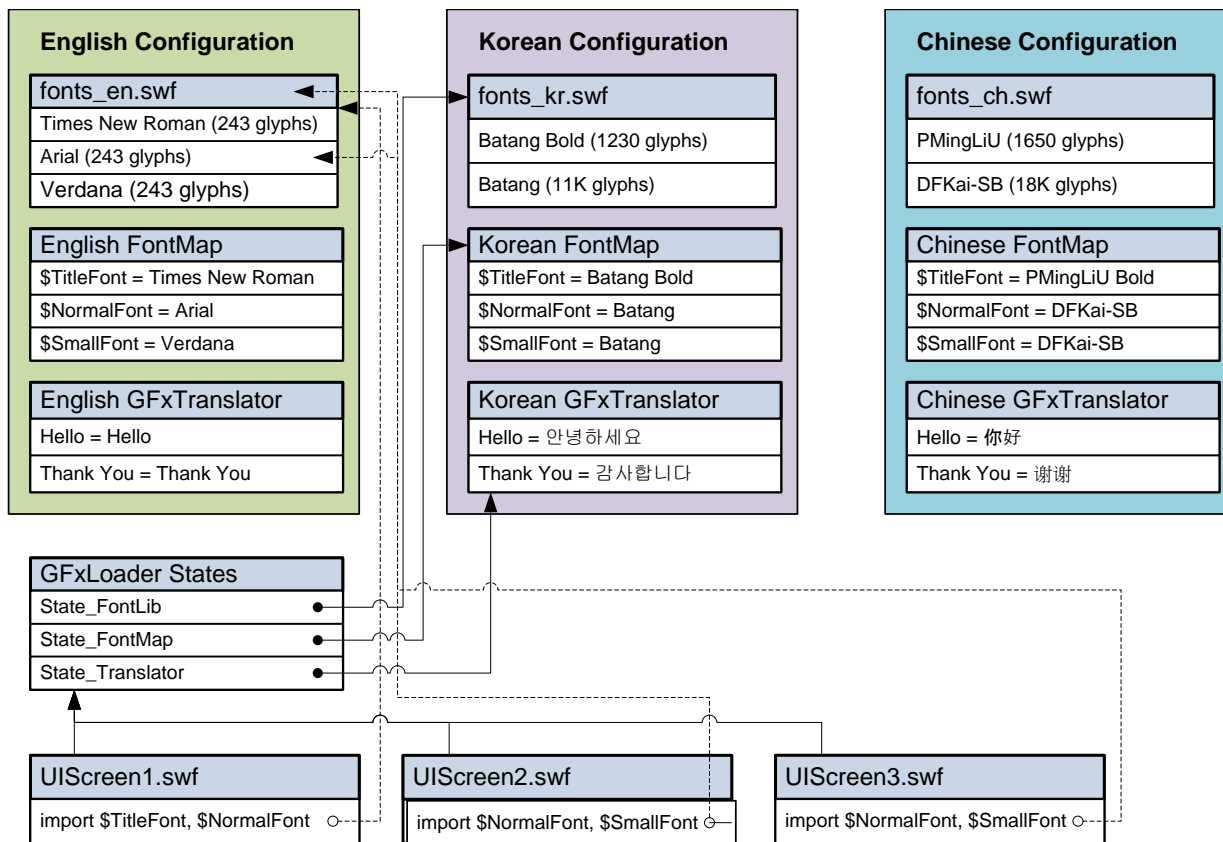
この方法はメインの SWF ファイルの CreateInstance を呼び出すよりも必ず前に呼び出す必要があります。

基本的に、ゲームの多国語設定は次のデータセットで構成されます。

1. 文字列変換テーブル: ゲームの各フレーズを翻訳されたフレーズにマップするものです。
2. 一連のフォントのセット: 翻訳で使用する文字のグリフを提供するものです。
3. フォントの置き換えテーブル: フォント シンボル名 ("NormalFont" など) を多国語設定内で使用可能となっているフォントにマップします。

Scaleform では、これらのコンポーネントは Gfx::Loader 上にインストール可能なステートオブジェクト Gfx::Translator、Gfx::FontLib、および Gfx::FontMap に相当します。

Gfx::Loader::CreateMovie の呼び出しによってムービーファイルがロードされると、ローダが提供しているフォント設定ステートに結合し、そのフォント設定ステートに在るフォントによって、テキストがレンダリングされます。異なる別のフォント結合を作成するには、ローダ上で新しいステートの組み合わせを設定し、もう一度 CreateMovie を呼び出さなければなりません。



上の図は、多国語対応のフォント設定として考えられる例を示しています。この設定では、三つのユーザインターフェイスファイル “UIScreen1.swf” ~ “UIScreen3.swf” が “fonts\_en.swf” ライブラリから一連のフォントのセットを読み込みます。読み込みファイルのリンクは、破線の矢印で示されています。Scaleform では、読み込みファイルからフォントを取得するのではなく、GfX::Loader 上で設定された GfX::FontLib ステートと GfX::FontMap ステートに基づいてフォントを置き換えることができます。この代替は次の 2 条件：（1）インポートされるファイルが Loader::SetDefaultFontLibName() の方法の指定するファイルとマッチする、（2）ヌルではない GfX::FontLib ステートが、UI スクリーンがロードされる前にローダーファイルにセットされている、が成立している場合には自動的に成り立ちます。

多国語変換をサポートするために、上の図では三つ設定が使用されています。つまり、英語、韓国語、中国語です。各設定には、それぞれに固有のフォントファイルが用意されています。例えば、韓国語には “fonts\_kr.swf” が使われています。また、読み込みフォントシンボル名をフォントファイルに埋め込まれた実際のフォントにマップするフォントマップが含まれています。この例では、ローダのステートが韓国語の設定になっているので、韓国語のフォントと変換テーブルが使用されます。ただし、ローダのステートは英語または中国語の設定に容易に変更できます。その場合は、ユーザインターフェイスがそれぞれの言語に変換されます。UI 画面の ファイルを変更せずに、翻訳が実現できるのです。

### 3.1.1 多国語のテキストの設定

読み込みフォントの置き換えを使用してゲームを多国語化する場合は、まず多国語のアートアセットと、必要な言語の設定を作成する必要があります。

アートとフォントのアセットは、通常は SWF 形式のファイルに格納されます。これは、ゲームがマスター化される前に GFX 形式に変換することができます。一方、フォントマップと変換テーブルのデータは GfX::FontMap オブジェクトと GfX::Translator オブジェクトの作成をサポートしている、ゲームに固有のデータ形式で格納できます。

Scaleform のサンプルでは “fontconfig.txt” 設定ファイルに基づいてこの情報を提供していますが、ゲーム開発者はプロダクション用に、より高度なスキーマを作成されたほうが良いでしょう。

多国語設定の作成プロセスを形式的に表せば、次のようなステップに細分化されます：

1. パート 1 で説明したようにデフォルトのライブラリファイル (“fonts\_en.swf” など) を作成して、これを使用してゲームアセットを作成してください。
2. 各言語用の文字列変換テーブルを作成します。このテーブルを使用して、対応する各 GfX::Translator オブジェクトを作成できます。
3. 各言語に使用するフォントマッピングを決定し、このマップを GfX::FontMap の作成に使用できる形式で格納します。
4. 各ターゲット言語ごとに “fonts\_kr.swf” などと言語固有のフォントファイルを作成します。

多国語設定が出来あがれば、多国語の UI 画面をゲームにロードできるようになります。ターゲット言語が選択されると、次の手順を実行する必要があります。

1. その言語のトランスレータオブジェクトを作成し、GfX::Loader 上に、これを設定します。翻訳は、GfX::Translator から独自のクラスを抽出し、Translate 仮想関数を上書きすることによって実装されます。開発者は、このクラスを上書きすることによって、選択された任意の形式で翻訳データを表現できます。
2. GfX::FontMap オブジェクトを作成し、GfX::Loader 上に、これを設定します。MapFont 関数を呼び出して、必要なフォントマッピングを GfX::FontMap オブジェクトに追加します。GfX::FontMap の使用例はパート 3 にあります。
3. 他にゲームに必要なフォント関連のステートがあれば、(例えば、GfX::FontCacheManager, GfX::FontProvider や GfX::FontPackParams など)、ロード上に設定します。
4. GfX::FontLib オブジェクトを作成し、GfX::Loader 上に、これを設定します。
5. デフォルトのフォントライブラリに次のように SetDefaultFontLibName(filename) の方法 (swf の内容に使用したもの) を呼び出します。

```
Loader.SetDefaultFontLibName("fonts_en.swf");
```

GfX::FontLib が作成されたら、ターゲット言語で使用するフォントソースの SWF/GFX ファイルをロードし、これを GfX::FontLib::AddFromFile の呼び出しによってライブラリに追加します。この呼び出しにより、読み込みフォントまたはデバイスフォントとして使用可能な引数ファイルにフォントが埋め込まれます。

6. GfX::Loader::CreateMovie を呼び出してユーザインターフェイスファイルをロードします。フォントマップ、フォントリブ、トランスレータの各ステートがユーザインターフェイスに自動的に適用されます。

### 3.1.2 Scaleform Player 上での多国語化

多国語化を実現するために、Scaleform Player は多国語のプロファイルファイル (通常は "fontconfig.txt" と呼ばれる) の使用をサポートします。SWF/GFX ファイルを Scaleform Player にドラッグアンドドロップすると、ファイルローカルディレクトリに 'fontconfig.txt' が在れば、設定が自動的にロードされます。ここで、コマンドラインから /fc オプションを指定することもできます。多国語のプロファイルがロードされると、ユーザは Ctrl+N キーを押すことによって、言語設定を切り替えることができます。F2 キーを押すと、現在の設定が Scaleform Player HUD の下部に表示されます。

多国語のプロファイルファイルは、8 バイトの ASCII 形式または UTF-16 形式で保存でき、フォント設定とその属性のリストによる線形構造を持ちます。次の多国語のプロファイルを使用して、前のセクションで示したゲームの設定を説明します。

```
[FontConfig "English"]
fontlib "fonts_en.swf"
map "$TitleFont" = "Times New Roman" Normal
map "$NormalFont" = "Arial " Normal
map "$SmallFont" = "Verdana" Normal

[FontConfig "Korean"]
fontlib "fonts_kr.swf"
map "$TitleFont" = "Batang" Bold
map "$NormalFont" = "Batang" Normal
map "$SmallFont" = "Batang" Normal
tr  "Hello" = "안녕하세요"
tr  "Thank You" = "감사합니다"

[FontConfig "Chinese"]
fontlib "fonts_ch.swf"
map "$TitleFont" = "PMingLiU" Bold
map "$NormalFont" = "DFKai-SB" Normal
map "$SmallFont" = "DFKai-SB" Normal
tr  "Hello" = "你好"
tr  "Thank You" = "谢谢"
```

サンプルからわかるように、独立した三つの設定セクションがあり、各セクションは、[FontConfig "name"] ヘッダで始まっています。設定が選択されると、Scaleform Player HUD にその設定の名前が表示されます。各設定内で、その設定に適用するステートメントのリストが使用されます。使用可能なステートメントの概要を、次の表に示します。

設定ステートメント	意味
fontlib "fontfile"	指定された SWF/GFX フォントファイルを Gfx::FontLib にロードします。最初に現れる fontlib がデフォルトライブラリとして使用されます。
map "\$UIFont" = "PMingLiU"	Gfx::FontMap に、エントリを追加します。Gfx::FontMap は、ゲーム UI 画面で使用するフォントをフォント ライブラリが提供するターゲットフォントにマップします。読み込みフォントの置き換えでは、\$UIFont は、初めに "fonts_en.swf" から UI ファイルにインストールしたフォント シンボルの名前でない限りなりません。

<pre>tr  "Hello" = "你好"</pre>	<p>ソース文字列からの翻訳を、ターゲット言語の相応する文字列に追加します。開発者は、変換テーブルを格納するためのより高度なソリューションを使用する必要があります。</p>
-------------------------------	--

設定ファイルの機能は、Scaleform を使えば多国語化が如何に実現できるのかという方法の例とすることを主眼として提供されていることを御理解ください。設定ファイルの構造は、将来、通知なしに変更されることがあります。さらに、私どもでは、Scaleform 多国語化の機能の拡張に伴い、XML 形式への移行も計画しています。開発者は、フォントステートを設定する方法の一例として、FontConfigParser.h/.cpp ファイルのソース コードを参照してください。

読み込みフォントの置き換えのより完全に近いサンプルは、Scaleform の Bin\Data\AS2\Samples\FontConfig and Bin¥Data¥AS3¥Samples¥FontConfig ディレクトリにあります。このディレクトリには、Scaleform Player にドロップできる “sample.swf” ファイルと、これを様々な言語に変換する “fontconfig.txt” ファイルが含まれています。開発者は、このディレクトリ内のファイルを調べて、多国語化のプロセスについて理解を深めてください。

### 3.1.3 デバイスフォントエミュレーション

デバイスフォントエミュレーションは、読み込みフォントの置き換えとは異なり、読み込みフォントシンボルを利用する代わりに Gfx::FontMap によってシステム フォント名を置き換えます。アーティストは、UI 作成時に開発システムのフォントセット (例えば、“Arial” や “Verdana”) を選択し、これをすべての UI アセットのデバイスフォントとして直接使用します。Scaleform Player で実行する場合に、フォント設定ファイルを使用して、システムフォント名をターゲット言語の該当する代替フォントにマップすることができます。こうしたマッピングの例として、“Arial” は韓国語のユーザ インターフェイスの “Batang” にマップできます。次に、“Batang” フォントは、さらに Gfx::FontLib を使用して “font\_kr.swf” からロードすることができます。

デバイスフォントエミュレーションは、システムフォントを Gfx::FontProviderWin32 から直接使用する時や、または Gfx::FontProviderFT2 からフォントファイルを使用する場合に、効果的です。デバイスフォントエミュレーションでは、Gfx::FontLib に依存することはありません。このアプローチは設定が簡単に思われるかもしれませんが、デバイスフォントには以下のように制限が多く、読み込みフォントの置き換えに比べて柔軟性が低くなります。

デバイスフォントは Flash では変形できません。Adobe Flash Player では、デバイスフォントテキストフィールドに対して回転を伴う変形が行われた場合、これを表示できません。また、デバイスフォント テキストフィールドは正しくスケーリングされず、これに適用されたマスクは無視されます。こうした機能のすべては Scaleform では正しく動作しますが、Flash のサポートが限定されているため、UI アートアセットのテストは困難になります。

デバイスフォントは、「アンチエイリアス (アニメーション優先)」設定を適用する視覚的な設定はできません。Scaleform Player は、デバイス フォントのテキストに対して読みやすいように常にアンチエイリアス処理を行っています (ActionScript の TextField.antiAliasType プロパティで、処理しないように設定することも出来ます)。

デバイスフォントを使用すると、UI アセットのすべてを再編集しない限り、初期開発フォントを変更できなくなります。Scaleform 側ではフォントマップの使用はもちろん可能ですが、Adobe Flash Player でファイルをテストする場合は使えません。代わりにインポートされたフォントの置換を使用する場合は、デフォルトの lib ファイル (“fonts\_en.swf”) を編集、再生成するだけで異なるフォントを選択できます。

ActionScript で TextFormat クラスを使っても、Flash には、読み込まれていないフォントシンボル名は認識されません。つまり、開発者はシンボリック名 (“\$TitleFont” など) でなく、フォント名 (“Arial” など) を直接使わなければなりません。

デバイスフォントエミュレーションでは、“fonts\_en.swf” ファイルは厳密には必要ないのですが、開発時のフォントシンボルリポジトリとしては使用可能です。アーティストが、ここで “fonts\_en.swf” ファイルの使いたいときは、そこにあるフォントシンボルを書き出してはいけません。代わりに、“fonts\_en.swf”にあるシンボルは、ターゲット UI ファイルにコピーしてください。こうすれば、シンボルは、代表するマップされたデバイス フォントのエイリアスとして機能します。

### 3.1.4 フォントライブラリ作成の詳細手順

ここでは 2 つの言語を持つフォントライブラリを使用し、fontconfig.txt を使用する簡単な SWF の作成方法を説明します。

1. 新規 FLA を作成し、ActionScript 2.0 または 3.0 を選択します。これを ‘main-app.fl’a と名付けます。これがフォントライブラリを使用するメインのアプリケーション SWF となります。
2. 別の FLA を作成し、前のステップで選択したのと同じ ActionScript を選択します。これを ‘fonts\_en.swf’ と名付けます。これがデフォルトのフォントライブラリとなります。
3. ‘Library’ウィンドウを参照し、マウスの右ボタンをクリックして ‘New Font…’ を選択します。名前を \$Font1 と指定して、“Family”には “Arial”、“Style”には “Regular” を選択します。
4. “Character ranges”には組み込みたい文字群を選択します。英語には ‘Basic Latin’ を選択すれば十分です。
5. ‘ActionScript’タブを選択します。‘Export for ActionScript’、‘Export for frame 1’、‘Export for run-time sharing’にチェックマークを入れます。‘URL’入力フィールドには ‘fonts\_en.swf’ とタイプします。OK ボタンをクリックします。ActionScript 3.0 Flash Studio は ‘A definition for this class could not be found…’ という警告を出しますが、これは無視して再度 ‘OK’ をクリックします。
6. 別のフォントに関してステップ 3～5 を繰り返し、‘\$Font2’ と名付けて、異なる ‘Family’ および/または ‘Style’ を選択します。‘Family’ と ‘Style’ が同じ場合、Flash Studio は 2 つめのフォントは最初のフォントの複製だとして破棄します。ここでは ‘Times New Roman’ を選択するとします。
7. もう 1 つの FLA を作成し、同じ ActionScript の設定を使用します。これを使用する言語に応じて ‘fonts\_kr.fl’a、‘fonts\_ru.fl’a などと名付けます。韓国語の場合 ‘fonts\_kr.fl’a とします。



8. 'Library'ウィンドウを参照してマウスの右ボタンをクリックして'New Font...'を選択します。名前を\$Font1 と指定して、今回は"Family"には何らかの韓国語フォント、例えば"□□"、"Style"には"Regular"を選択します。
9. "Character ranges"には組み込みたい文字群を選択します。韓国語では例えば'Korean Hangul (All)'とします。
10. 'ActionScript'タブを選択します。'Export for ActionScript'、'Export for frame 1'、'Export for run-time sharing'にチェックマークを入れます。'URL'入力フィールドには'fonts\_kr.swf'とタイプします。OK ボタンをクリックします。ActionScript 3.0 Flash Studio は'A definition for this class could not be found...'という警告を出しますが、これは無視して再度'OK'をクリックします。
11. 別のフォントに関してステップ 8~10 を繰り返し、'\$Font2'と名付けて、'Family'には例えば'□□□'を選択します。
12. 'fonts\_en.swf'と'fonts\_kr.swf'の両方を発行します。
13. ここで main-app.flc.に戻ります。しかし先ずデフォルトのライブラリファイル ('fonts\_en.swf') を参照し、そのライブラリの'\$Font1'と'\$Font2'両方を選択してクリップボードにコピーします (Ctrl-C または右クリックから'Copy'を選択) 。
14. 'main-app.flc.'に切り替えてライブラリを参照し、フォントシンボル j を貼り付けます (Ctrl-V または右クリックから'Paste'を選択) 。選択したフォントに'Import:'のプリフィックスが付いたものが確認できます。
15. 'main-app.flc.'のステージにテキストフィールドを作成します。'Classic Text'、'Dynamic Text'とします。'Family'のドロップダウンリストから'\$Font1\*'を選択します。テキストフィールドの内容として'\$TEXT1'とタイプします (これが翻訳 ID として使用されます) 。
16. 第 2 のテキストフィールドを作成して、'\$Font2\*'を選択し、その中に'\$TEXT2'とタイプします。
17. 'main-app.swf'を発行して、この段階は終了です。
18. ここで、同じディレクトリで全ての swf を保存する fontconfig.txt を作成します。Unicode または UTF-8 を扱えるノートパッドなどのテキストエディタを起動します。次をタイプします。

```
[FontConfig "English"]
fontlib "fonts_en.swf"
map "$Font1" = "Arial"
map "$Font2" = "Times New Roman"
tr "$TEXT1" = "This is"
tr "$TEXT2" = "ENGLISH!"
```

```
[FontConfig "Korean"]
fontlib "fonts_kr.swf"
map "$Font1" = "Batang"
map "$Font2" = "BatangChe"
tr "$TEXT1" = "이것"
tr "$TEXT2" = "은 한국이다!"
```

**重要：**ここでは韓国語のフォント名（“□□”と“□□□”）を使用しましたが、Flash は SWF で英語名を使用することがあります（“Batang”と“BatangChe”）。そのために fontconfig.txt では英語フォント名を使用しています。SWF でどのフォント名が生成されているかを確認するには、`gfxexport -fntlst <swfname>` コマンドを使用して出力される \*.lst ファイルを解析してください。

19. ファイルを保存します（必ず Unicode または UTF-8 として保存してください）。これで終了です。GFXPlayer で 'main-app.swf' をオープンするとデフォルトテキストとして 'This is' と 'English!' が現れます。Ctrl-N で韓国語に切り替えると、fonts\_kr.swf からのフォントで英語のフォントが韓国語に置き換わっているのが確認できます（韓国語のグリフは fonts\_kr.swf でのみ組み込んだため）。

## 3.2 カスタムアセットの生成

カスタムアセットの生成は、言葉通り、各ターゲット言語のマーケット向けにカスタム SWF/GFX ファイルを作成するということです。ゲームに "UIScreen.fla" ファイルを使うとしましょう、各言語マーケット向けのゲームごとに異なるバージョンの SWF を手動で生成することができます。例えば、韓国語の "UIScreen.swf" と、英語の "UIScreen.swf" とは中味が異なることになります。異なるバージョンのファイルは、別のファイルシステムディレクトリに置いておくことができます。そうしておきませんと、ターゲット言語のマーケットでゲームを販売することができなくなってしまいます。

カスタムアセットを使用する最大の利点は、必要最少限のグリフを各ファイルに埋め込めることです。このアプローチは一般的にはお勧めできませんが、メモリが極端に制限されており (UI 用に 600K 未満の空き領域)、アートアセットが以下の基準を満たす場合は、カスタムアセットの生成を検討してください。

UI アセットファイルの数が比較的少なく、テキストコンテンツがシンプルである。

複数の UI ファイルが同時にロードされることが、ほとんどない (同時にロードされるような場合は、他のアプローチによるフォント共有機能を利用することを推奨します)。

ダイナミックテキストフィールドの更新は限定的で、使用する埋め込み文字の数が少ない。

UI には、アジア言語の IME サポートが不要である。

開発者がゲームタイトルにカスタムアセットの生成を使用する場合は、Flash Studio のマニュアルで「[ストリングパネルを使って多言語テキストをオーサリング]」の項を参照し、多国語テキストの [ストリング] パネル (Ctrl + F11) の使用を検討することをお勧めします。アーティストは、現バージョンの Scaleform で Flash テキストの置き換えが正しく機能するように、[ストリングの置き換え] を「ステージ言語を使用して手動で」に設定する必要があります。

## 4 パート 3:フォントソースの設定

Flash 内のテキストフィールドは、それぞれにフォント名が関連付けられており、そのまま、または HTML タグ形式で格納されます。テキストフィールドが表示されるときには、レンダリングに使用するフォントが以下のフォント ソースの検索によって取得されます。

1. ローカルに埋め込まれたフォント
2. 別の SWF/GFX ファイルから読み込んだフォントシンボル
3. GfX::FontLib からインストールされた SWF/GFX ファイル。フォント名または読み込みフォントの置き換えで検索されます。
4. システムフォントプロバイダ (例えば GfX::FontProviderWin32、ユーザがインストールした場合)

埋め込みフォントと読み込みフォントの使い方については、本書の冒頭で説明しました。ほとんどの場合、埋め込みフォントは Flash と同様に機能するので、カスタム設定は必要ありません。フォント参照の目的では、読み込みフォントシンボルは埋め込みフォントと同様に機能します。

埋め込まれていないフォントの参照を設定するインストール可能な三つのステートは、GfX::FontLib、GfX::FontMap、GfX::FontProvider です。前述したように、GfX::FontLib と GfX::FontMap は、読み込みフォントの置き換え、あるいはデバイスフォントエミュレーションのために、SWF/GFX でロードされたフォントの参照に使用されます。システムフォントプロバイダの使い方について、これから詳しく説明します。

システムフォントプロバイダは GfX::Loader::SetFontProvider 呼び出しによってインストールされます。これによって、SWF 以外のファイルソースからフォントデータを取得できるようになります。フォントプロバイダは、ダイナミックキャッシュのみで使用でき、フォントデータが SWF またはフォント ライブラリに埋め込まれていない場合にだけ検索されます。現在のところ、Scaleform には次の二つのフォントプロバイダがあります。

GfX::FontProviderWin32 – Win32 API に基づいてフォントグリフデータを取得します。

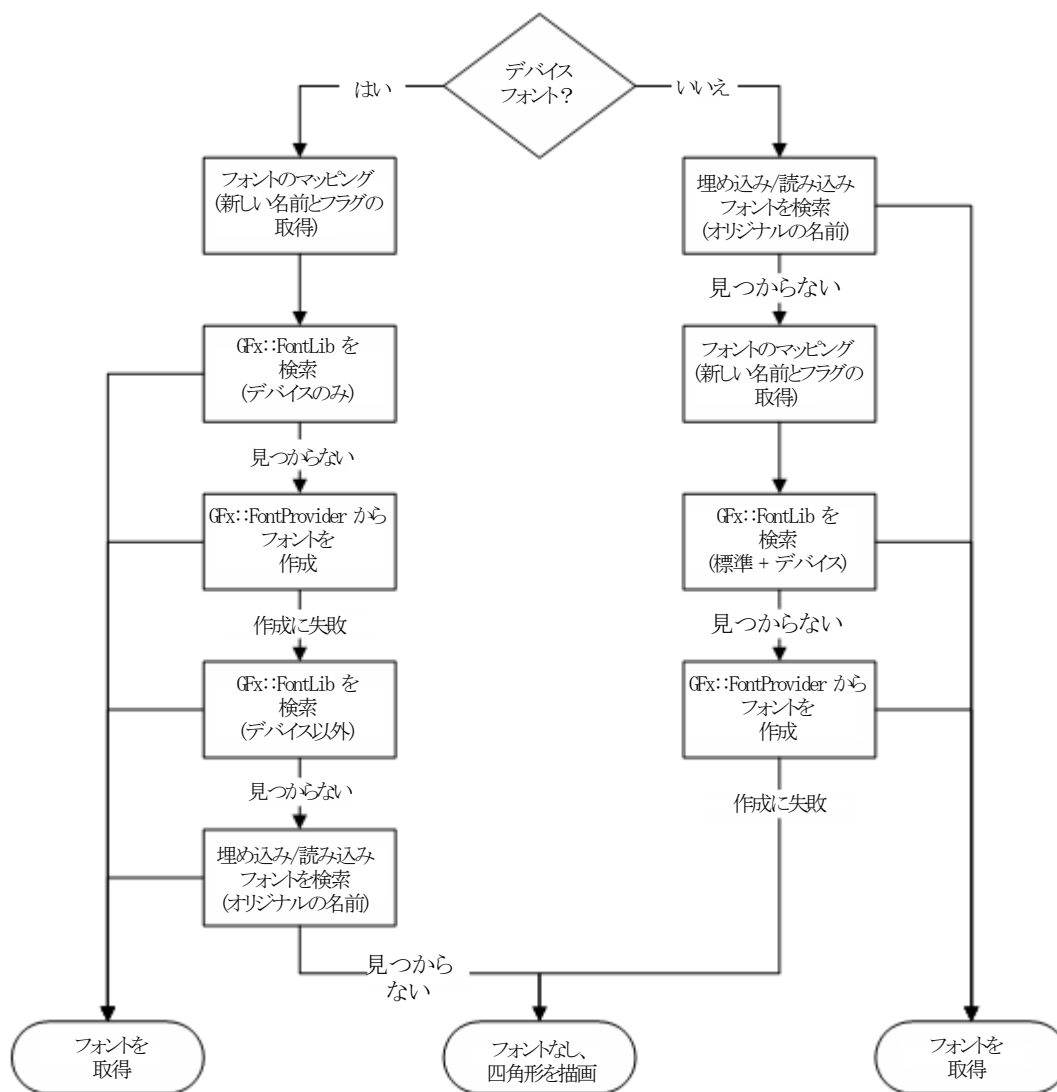
GfX::FontProviderFT2 – David Turner が開発した FreeType-2 ライブラリを使用して、独立したフォントファイルを読み込んで解釈します。

このパートでは、フォント参照順について詳しく説明し、様々なフォントソースの設定方法のコード例を示します。

### 4.1 フォントの参照順

次のページのフローチャートに Scaleform によるフォントの参照順を示します。チャートからわかるように、参照の動作はテキストフィールドのプロパティのレンダリング方法で [デバイスフォントの使用] が選択された場合に設定されるデバイスフォントフラグによって変わります。

Adobe Flash では、デバイスフォントフラグを設定すると、対応するフォントが、同じ名前の埋め込みフォントより先に、システムから取得されます (この場合では、後者-埋め込みフォントは、フォールバックとして使用されます)。Scaleform はこの動作を再現しますが、インストールされたフォントライブラリがシステムフォント プロバイダより先に検索されます。こうした設定によって競合が発生するようなことはありません。というのは、移植対象のほとんどの家庭用ゲーム機は、共有フォントライブラリに依存しており、一方では、システムプロバイダを使用するゲームはフォント ライブラリを初期化されていない (設定されていない) 状態に維持しているからです。



上図からわかるように、テキストフィールドにデバイスフォントが使用されて「いない」場合には、埋め込みフォントが、最初に検索され、それ以外の場合には最後に検索されます。さらに、埋め込みフォントは常にテキストフィールドで使われているオリジナルのフォント名に基づいて参照されますが、フォントマップが使われると、Gfx::FontLib と Gfx::FontProvider から参照されたフォント名に置き換えられます。

## 4.2 Gfx::FontMap

Gfx::FontMap は、フォント名を置き換えるためのステートであり、必要な文字が初期開発フォント内にない場合は、代替フォントを多国語化のために使用できるようにします。フォントマップは、読み込みフォントの置き換えとデバイスフォントエミュレーションの両方で使用されます。前者の場合は、フォントシンボル識別子をフォント名に変換します。後者の場合は、オリジナルのフォント名を翻訳先の代替名にマップします。

パート 3 では、フォント設定ファイル内の以下の行でフォントマップが作成されていました。

```
[FontConfig "Korean"]
fontlib "fonts_kr.swf"
map "$TitleFont" = "Batang" Bold
map "$NormalFont" = "Batang" Normal
map "$SmallFont" = "Batang" Normal
```

この例では、マッピングステートメントを使用してフォント読み込み識別子 (“\$TitleFont” など) を実際のフォント名 (この場合はフォントライブラリファイルに埋め込まれている) にマップしています。

同等のフォントマップ設定は、以下の C++ ステートメントによっても行なえます。

```
#include "Gfx/Gfx_FontLib.h"
. . .
Ptr<FontMap> pfontMap = *new FontMap;
Loader.SetFontMap(pfontMap);

pfontMap->MapFont("$TitleFont", "Batang", FontMap::MFF_Bold, scaleFactor = 1.0f);
pfontMap->MapFont("$NormalFont", "Batang", FontMap::MFF_Normal, scaleFactor = 1.0f);
pfontMap->MapFont("$SmallFont", "Batang", FontMap::MFF_Normal, scaleFactor = 1.0f);
```

この場合は、三つのフォントがすべて同じフォント名にマップされます。特に、“\$NormalFont” と “\$SmallFont” は同じフォントスタイルを共有しています。これで、フォントライブラリファイルで使用するメモリを節約できます。第 3 の引数に MapFont を指定すると、特定の埋め込みを使用するように強制的にマッピングすることができます。この引数を指定しないと、MFF\_Original の値が使用され、フォント参照はテキストフィールドに最初に指定されたスタイルに維持されることを示しています。フォントのサイズは scaleFactor パラメータで設定される値で変更することができます。デフォルトでは、scaleFactor は 1.0f に設定されています。フォントがうまく表示されず少し大きくしたい場合に、このパラメータは便利です。

開発者は、Gfx::FontMap が結合のステートであることを認識する必要があります。つまり、これで作成されるムービーインスタンスは、後にローダ内で変更されてもこのステートを使用することになります。別のフォント マップが設定されると、同じファイル名の Gfx::Loader::CreateMovie によって別の Gfx::MovieDef が返されます。これは、他のすべてのフォント設定ステートについても言えることです。

## 4.3 Gfx::FontLib

パート 3 で述べたように、Gfx::FontLib ステートは、(1) デフォルトの“fonts\_en.swf”からインポートされたフォントの代わりを提供し、(2) デバイスフォントのエミュレーション用のフォントを提供する、インストール可能なフォントライブラリを表します。フォントライブラリはシステムフォントプロバイダより先に検索されます。その使い方は、次の例から明らかになります。

```
#include "Gfx/Gfx_FontLib.h"
. . .
Ptr<FontLib> fontLib = *new FontLib;
Loader.SetFontLib(fontLib);
fontLib->SetSubstitute("<default_font_lib_swf_or_gfx_file>");

Ptr<MovieDef> m1 = *Loader.CreateMovie("<swf_or_gfx_file1>");
Ptr<MovieDef> m2 = *Loader.CreateMovie("<swf_or_gfx_file2>");
. . .
fontLib->AddFontsFrom(m1, true);
fontLib->AddFontsFrom(m2, true);
```

ムービーは、作成されて通常どおりにロードされるのですが、再生されるのではなく、フォントストレージとして使用される、と考えてください。ムービーは、必要な数だけロードすることができます。別々のムービーで同一のフォントが定義されている場合には、最初の ムービーだけが使用されます。

AddFontsFrom の最初のパラメータは、フォントソースとなるムービー定義です。AddFontsFrom の第 2 パラメータは、ピンフラグです。ローダがメモリ内のムービーに AddRef する必要がある場合にセットされます。このフラグは、ロードされたムービー (例の m1 と m2) へのスマートポインタをユーザが保持しない場合にのみ必要です。ピンフラグが「認識しない」(false)場合は、書き出された、またはバックされたテキストチャがすぐに解放され、フォントを使用するときに再ロード/再生成する必要があります。

フォントマップと同様に、Gfx::FontLib は作成されたムービーが削除されるまで参照する結合ステートです。

## 4.4 Gfx::FontProviderWin32

Gfx::FontProviderWin32 フォントプロバイダは Win32 API で使用でき、GetGlyphOutline() 関数によってベクトルデータを取得できます。以下のように使用できます。

```
#include "Gfx/Gfx_FontProviderWin32.h"
. . .
Ptr<FontProviderWin32> fontProvider = *new FontProviderWin32(::GetDC(0));
Loader.SetFontProvider(fontProvider);
```

Gfx::FontProviderWin32 のコンストラクタは、引数として、Windows ディスプレイコンテキスト (DC) のハンドラをとります。ほとんどの場合、画面 DC を使うのが適切です (::GetDC(0))。

フォントは、リクエストされれば必要に応じて生成されます。

#### 4.4.1 ネイティブにヒントの付いたテキストを使う

多くの場合、フォントヒンティングは決して見過すことの出来ない問題です。Scaleform はオートヒンティングメカニズムを持っているとはいえ、中国語、日本語、韓国語の、所謂「CJK」(Chinese / Japanese / Korean)文字に対しては十分に機能するとは言えません。加えて、多くの巧みにデザインされた「CJK」フォントは特定のサイズのグリフにはラスタイメージを持っています。というのは小さなサイズの「CJK」文字に適切なヒンティングを行なうのは極めて複雑なことになってしまうからです。ベクトルのグリフをラスタイメージに複製するのは、適切で実用的な解決方法でしょう。フォント API に基づくフォントプロバイダー (GFX::FontProviderWin32 や GFX::FontProviderFT2) は、両方の形、つまりベクトルとラスタの表示体で、グリフを生成することが出来ます。これらのフォントプロバイダーには、ネイティブなヒンティングをコントロールできるインターフェイスがあります。インターフェイスには、やや異なる点もありますが、原理は同じです。以下の 4 個のパラメータがあります。

```
Font::NativeHintingRange vectorRange;  
Font::NativeHintingRange rasterRange;  
unsigned maxVectorHintedSize;  
unsigned maxRasterHintedSize;
```

vectorRange と rasterRange パラメータは、ヒンティングが適用される範囲をコントロールしています。ここでの値は以下ようになります：

```
Font::DontHint - ネイティブヒンティングを使わない  
Font::HintCJK - 中国語、日本語、韓国語文字にネイティブヒンティングを使う  
Font::HintAll - 全ての文字にネイティブヒンティングを使う
```

rasterRange は、vectorRange より優先されます。maxVectorHintedSize と maxRasterHintedSize とのパラメータは、ネイティブヒンティングが適用される最大フォントサイズをピクセルで定義します。次の画像では、SimSun フォントを例にして、いくつかのオプションをお見せしています。

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ	vectorRange=GFXFont::DontHint rasterRange=GFXFont::DontHint
𐄂𐄃𐄄𐄅𐄆𐄇𐄈𐄉𐄊𐄋𐄌𐄍𐄎𐄏𐄐𐄑𐄒𐄓𐄔𐄕𐄖𐄗𐄘𐄙𐄚𐄛 𐄜𐄝𐄞𐄟𐄠𐄡𐄢𐄣𐄤𐄥𐄦𐄧𐄨𐄩𐄪𐄫𐄬𐄭𐄮𐄯𐄰𐄱𐄲𐄳𐄴𐄵𐄶𐄷𐄸𐄹𐄺𐄻𐄼𐄽𐄾𐄿𐅀𐅁𐅂𐅃𐅄𐅅𐅆𐅇𐅈𐅉𐅊𐅋𐅌𐅍𐅎𐅏𐅐𐅑𐅒𐅓𐅔𐅕𐅖𐅗𐅘𐅙𐅚𐅛𐅜𐅝𐅞𐅟𐅠𐅡𐅢𐅣𐅤𐅥𐅦𐅧𐅨𐅩𐅪𐅫𐅬𐅭𐅮𐅯𐅰𐅱𐅲𐅳𐅴𐅵𐅶𐅷𐅸𐅹𐅺𐅻𐅼𐅽𐅾𐅿𐆀𐆁𐆂𐆃𐆄𐆅𐆆𐆇𐆈𐆉𐆊𐆋𐆌𐆍𐆎𐆏𐆐𐆑𐆒𐆓𐆔𐆕𐆖𐆗𐆘𐆙𐆚𐆛𐆜𐆝𐆞𐆟𐆠𐆡𐆢𐆣𐆤𐆥𐆦𐆧𐆨𐆩𐆪𐆫𐆬𐆭𐆮𐆯𐆰𐆱𐆲𐆳𐆴𐆵𐆶𐆷𐆸𐆹𐆺𐆻𐆼𐆽𐆾𐆿𐇀𐇁𐇂𐇃𐇄𐇅𐇆𐇇𐇈𐇉𐇊𐇋𐇌𐇍𐇎𐇏𐇐𐇑𐇒𐇓𐇔𐇕𐇖𐇗𐇘𐇙𐇚𐇛𐇜𐇝𐇞𐇟𐇠𐇡𐇢𐇣𐇤𐇥𐇦𐇧𐇨𐇩𐇪𐇫𐇬𐇭𐇮𐇯𐇰𐇱𐇲𐇳𐇴𐇵𐇶𐇷𐇸𐇹𐇺𐇻𐇼𐇽𐇾𐇿𐈀𐈁𐈂𐈃𐈄𐈅𐈆𐈇𐈈𐈉𐈊𐈋𐈌𐈍𐈎𐈏𐈐𐈑𐈒𐈓𐈔𐈕𐈖𐈗𐈘𐈙𐈚𐈛𐈜𐈝𐈞𐈟𐈠𐈡𐈢𐈣𐈤𐈥𐈦𐈧𐈨𐈩𐈪𐈫𐈬𐈭𐈮𐈯𐈰𐈱𐈲𐈳𐈴𐈵𐈶𐈷𐈸𐈹𐈺𐈻𐈼𐈽𐈾𐈿𐉀𐉁𐉂𐉃𐉄𐉅𐉆𐉇𐉈𐉉𐉊𐉋𐉌𐉍𐉎𐉏𐉐𐉑𐉒𐉓𐉔𐉕𐉖𐉗𐉘𐉙𐉚𐉛𐉜𐉝𐉞𐉟𐉠𐉡𐉢𐉣𐉤𐉥𐉦𐉧𐉨𐉩𐉪𐉫𐉬𐉭𐉮𐉯𐉰𐉱𐉲𐉳𐉴𐉵𐉶𐉷𐉸𐉹𐉺𐉻𐉼𐉽𐉾𐉿𐊀𐊁𐊂𐊃𐊄𐊅𐊆𐊇𐊈𐊉𐊊𐊋𐊌𐊍𐊎𐊏𐊐𐊑𐊒𐊓𐊔𐊕𐊖𐊗𐊘𐊙𐊚𐊛𐊜𐊝𐊞𐊟𐊠𐊡𐊢𐊣𐊤𐊥𐊦𐊧𐊨𐊩𐊪𐊫𐊬𐊭𐊮𐊯𐊰𐊱𐊲𐊳𐊴𐊵𐊶𐊷𐊸𐊹𐊺𐊻𐊼𐊽𐊾𐊿𐋀𐋁𐋂𐋃𐋄𐋅𐋆𐋇𐋈𐋉𐋊𐋋𐋌𐋍𐋎𐋏𐋐𐋑𐋒𐋓𐋔𐋕𐋖𐋗𐋘𐋙𐋚𐋛𐋜𐋝𐋞𐋟𐋠𐋡𐋢𐋣𐋤𐋥𐋦𐋧𐋨𐋩𐋪𐋫𐋬𐋭𐋮𐋯𐋰𐋱𐋲𐋳𐋴𐋵𐋶𐋷𐋸𐋹𐋺𐋻𐋼𐋽𐋾𐋿𐌀𐌁𐌂𐌃𐌄𐌅𐌆𐌇𐌈𐌉𐌊𐌋𐌌𐌍𐌎𐌏𐌐𐌑𐌒𐌓𐌔𐌕𐌖𐌗𐌘𐌙𐌚𐌛𐌜𐌝𐌞𐌟𐌠𐌡𐌢𐌣𐌤𐌥𐌦𐌧𐌨𐌩𐌪𐌫𐌬𐌭𐌮𐌯𐌰𐌱𐌲𐌳𐌴𐌵𐌶𐌷𐌸𐌹𐌺𐌻𐌼𐌽𐌾𐌿𐍀𐍁𐍂𐍃𐍄𐍅𐍆𐍇𐍈𐍉𐍊𐍋𐍌𐍍𐍎𐍏𐍐𐍑𐍒𐍓𐍔𐍕𐍖𐍗𐍘𐍙𐍚𐍛𐍜𐍝𐍞𐍟𐍠𐍡𐍢𐍣𐍤𐍥𐍦𐍧𐍨𐍩𐍪𐍫𐍬𐍭𐍮𐍯𐍰𐍱𐍲𐍳𐍴𐍵𐍶𐍷𐍸𐍹𐍺𐍻𐍼𐍽𐍾𐍿𐎀𐎁𐎂𐎃𐎄𐎅𐎆𐎇𐎈𐎉𐎊𐎋𐎌𐎍𐎎𐎏𐎐𐎑𐎒𐎓𐎔𐎕𐎖𐎗𐎘𐎙𐎚𐎛𐎜𐎝𐎞𐎟𐎠𐎡𐎢𐎣𐎤𐎥𐎦𐎧𐎨𐎩𐎪𐎫𐎬𐎭𐎮𐎯𐎰𐎱𐎲𐎳𐎴𐎵𐎶𐎷𐎸𐎹𐎺𐎻𐎼𐎽𐎾𐎿𐏀𐏁𐏂𐏃𐏄𐏅𐏆𐏇𐏈𐏉𐏊𐏋𐏌𐏍𐏎𐏏𐏐𐏑𐏒𐏓𐏔𐏕𐏖𐏗𐏘𐏙𐏚𐏛𐏜𐏝𐏞𐏟𐏠𐏡𐏢𐏣𐏤𐏥𐏦𐏧𐏨𐏩𐏪𐏫𐏬𐏭𐏮𐏯𐏰𐏱𐏲𐏳𐏴𐏵𐏶𐏷𐏸𐏹𐏺𐏻𐏼𐏽𐏾𐏿𐐀𐐁𐐂𐐃𐐄𐐅𐐆𐐇𐐈𐐉𐐊𐐋𐐌𐐍𐐎𐐏𐐐𐐑𐐒𐐓𐐔𐐕𐐖𐐗𐐘𐐙𐐚𐐛𐐜𐐝𐐞𐐟𐐠𐐡𐐢𐐣𐐤𐐥𐐦𐐧𐐨𐐩𐐪𐐫𐐬𐐭𐐮𐐯𐐰𐐱𐐲𐐳𐐴𐐵𐐶𐐷𐐸𐐹𐐺𐐻𐐼𐐽𐐾𐐿𐑀𐑁𐑂𐑃𐑄𐑅𐑆𐑇𐑈𐑉𐑊𐑋𐑌𐑍𐑎𐑏𐑐𐑑𐑒𐑓𐑔𐑕𐑖𐑗𐑘𐑙𐑚𐑛𐑜𐑝𐑞𐑟𐑠𐑡𐑢𐑣𐑤𐑥𐑦𐑧𐑨𐑩𐑪𐑫𐑬𐑭𐑮𐑯𐑰𐑱𐑲𐑳𐑴𐑵𐑶𐑷𐑸𐑹𐑺𐑻𐑼𐑽𐑾𐑿𐒀𐒁𐒂𐒃𐒄𐒅𐒆𐒇𐒈𐒉𐒊𐒋𐒌𐒍𐒎𐒏𐒐𐒑𐒒𐒓𐒔𐒕𐒖𐒗𐒘𐒙𐒚𐒛𐒜𐒝𐒞𐒟𐒠𐒡𐒢𐒣𐒤𐒥𐒦𐒧𐒨𐒩𐒪𐒫𐒬𐒭𐒮𐒯𐒰𐒱𐒲𐒳𐒴𐒵𐒶𐒷𐒸𐒹𐒺𐒻𐒼𐒽𐒾𐒿𐓀𐓁𐓂𐓃𐓄𐓅𐓆𐓇𐓈𐓉𐓊𐓋𐓌𐓍𐓎𐓏𐓐𐓑𐓒𐓓𐓔𐓕𐓖𐓗𐓘𐓙𐓚𐓛𐓜𐓝𐓞𐓟𐓠𐓡𐓢𐓣𐓤𐓥𐓦𐓧𐓨𐓩𐓪𐓫𐓬𐓭𐓮𐓯𐓰𐓱𐓲𐓳𐓴𐓵𐓶𐓷𐓸𐓹𐓺𐓻𐓼𐓽𐓾𐓿𐔀𐔁𐔂𐔃𐔄𐔅𐔆𐔇𐔈𐔉𐔊𐔋𐔌𐔍𐔎𐔏𐔐𐔑𐔒𐔓𐔔𐔕𐔖𐔗𐔘𐔙𐔚𐔛𐔜𐔝𐔞𐔟𐔠𐔡𐔢𐔣𐔤𐔥𐔦𐔧𐔨𐔩𐔪𐔫𐔬𐔭𐔮𐔯𐔰𐔱𐔲𐔳𐔴𐔵𐔶𐔷𐔸𐔹𐔺𐔻𐔼𐔽𐔾𐔿𐕀𐕁𐕂𐕃𐕄𐕅𐕆𐕇𐕈𐕉𐕊𐕋𐕌𐕍𐕎𐕏𐕐𐕑𐕒𐕓𐕔𐕕𐕖𐕗𐕘𐕙𐕚𐕛𐕜𐕝𐕞𐕟𐕠𐕡𐕢𐕣𐕤𐕥𐕦𐕧𐕨𐕩𐕪𐕫𐕬𐕭𐕮𐕯𐕰𐕱𐕲𐕳𐕴𐕵𐕶𐕷𐕸𐕹𐕺𐕻𐕼𐕽𐕾𐕿𐖀𐖁𐖂𐖃𐖄𐖅𐖆𐖇𐖈𐖉𐖊𐖋𐖌𐖍𐖎𐖏𐖐𐖑𐖒𐖓𐖔𐖕𐖖𐖗𐖘𐖙𐖚𐖛𐖜𐖝𐖞𐖟𐖠𐖡𐖢𐖣𐖤𐖥𐖦𐖧𐖨𐖩𐖪𐖫𐖬𐖭𐖮𐖯𐖰𐖱𐖲𐖳𐖴𐖵𐖶𐖷𐖸𐖹𐖺𐖻𐖼𐖽𐖾𐖿𐗀𐗁𐗂𐗃𐗄𐗅𐗆𐗇𐗈𐗉𐗊𐗋𐗌𐗍𐗎𐗏𐗐𐗑𐗒𐗓𐗔𐗕𐗖𐗗𐗘𐗙𐗚𐗛𐗜𐗝𐗞𐗟𐗠𐗡𐗢𐗣𐗤𐗥𐗦𐗧𐗨𐗩𐗪𐗫𐗬𐗭𐗮𐗯𐗰𐗱𐗲𐗳𐗴𐗵𐗶𐗷𐗸𐗹𐗺𐗻𐗼𐗽𐗾𐗿𐘀𐘁𐘂𐘃𐘄𐘅𐘆𐘇𐘈𐘉𐘊𐘋𐘌𐘍𐘎𐘏𐘐𐘑𐘒𐘓𐘔𐘕𐘖𐘗𐘘𐘙𐘚𐘛𐘜𐘝𐘞𐘟𐘠𐘡𐘢𐘣𐘤𐘥𐘦𐘧𐘨𐘩𐘪𐘫𐘬𐘭𐘮𐘯𐘰𐘱𐘲𐘳𐘴𐘵𐘶𐘷𐘸𐘹𐘺𐘻𐘼𐘽𐘾𐘿𐙀𐙁𐙂𐙃𐙄𐙅𐙆𐙇𐙈𐙉𐙊𐙋𐙌𐙍𐙎𐙏𐙐𐙑𐙒𐙓𐙔𐙕𐙖𐙗𐙘𐙙𐙚𐙛𐙜𐙝𐙞𐙟𐙠𐙡𐙢𐙣𐙤𐙥𐙦𐙧𐙨𐙩𐙪𐙫𐙬𐙭𐙮𐙯𐙰𐙱𐙲𐙳𐙴𐙵𐙶𐙷𐙸𐙹𐙺𐙻𐙼𐙽𐙾𐙿𐚀𐚁𐚂𐚃𐚄𐚅𐚆𐚇𐚈𐚉𐚊𐚋𐚌𐚍𐚎𐚏𐚐𐚑𐚒𐚓𐚔𐚕𐚖𐚗𐚘𐚙𐚚𐚛𐚜𐚝𐚞𐚟𐚠𐚡𐚢𐚣𐚤𐚥𐚦𐚧𐚨𐚩𐚪𐚫𐚬𐚭𐚮𐚯𐚰𐚱𐚲𐚳𐚴𐚵𐚶𐚷𐚸𐚹𐚺𐚻𐚼𐚽𐚾𐚿𐛀𐛁𐛂𐛃𐛄𐛅𐛆𐛇𐛈𐛉𐛊𐛋𐛌𐛍𐛎𐛏𐛐𐛑𐛒𐛓𐛔𐛕𐛖𐛗𐛘𐛙𐛚𐛛𐛜𐛝𐛞𐛟𐛠𐛡𐛢𐛣𐛤𐛥𐛦𐛧𐛨𐛩𐛪𐛫𐛬𐛭𐛮𐛯𐛰𐛱𐛲𐛳𐛴𐛵𐛶𐛷𐛸𐛹𐛺𐛻𐛼𐛽𐛾𐛿𐜀𐜁𐜂𐜃𐜄𐜅𐜆𐜇𐜈𐜉𐜊𐜋𐜌𐜍𐜎𐜏𐜐𐜑𐜒𐜓𐜔𐜕𐜖𐜗𐜘𐜙𐜚𐜛𐜜𐜝𐜞𐜟𐜠𐜡𐜢𐜣𐜤𐜥𐜦𐜧𐜨𐜩𐜪𐜫𐜬𐜭𐜮𐜯𐜰𐜱𐜲𐜳𐜴𐜵𐜶𐜷𐜸𐜹𐜺𐜻𐜼𐜽𐜾𐜿𐝀𐝁𐝂𐝃𐝄𐝅𐝆𐝇𐝈𐝉𐝊𐝋𐝌𐝍𐝎𐝏𐝐𐝑𐝒𐝓𐝔𐝕𐝖𐝗𐝘𐝙𐝚𐝛𐝜𐝝𐝞𐝟𐝠𐝡𐝢𐝣𐝤𐝥𐝦𐝧𐝨𐝩𐝪𐝫𐝬𐝭𐝮𐝯𐝰𐝱𐝲𐝳𐝴𐝵𐝶𐝷𐝸𐝹𐝺𐝻𐝼𐝽𐝾𐝿𐞀𐞁𐞂𐞃𐞄𐞅𐞆𐞇𐞈𐞉𐞊𐞋𐞌𐞍𐞎𐞏𐞐𐞑𐞒𐞓𐞔𐞕𐞖𐞗𐞘𐞙𐞚𐞛𐞜𐞝𐞞𐞟𐞠𐞡𐞢𐞣𐞤𐞥𐞦𐞧𐞨𐞩𐞪𐞫𐞬𐞭𐞮𐞯𐞰𐞱𐞲𐞳𐞴𐞵𐞶𐞷𐞸𐞹𐞺𐞻𐞼𐞽𐞾𐞿𐟀𐟁𐟂𐟃𐟄𐟅𐟆𐟇𐟈𐟉𐟊𐟋𐟌𐟍𐟎𐟏𐟐𐟑𐟒𐟓𐟔𐟕𐟖𐟗𐟘𐟙𐟚𐟛𐟜𐟝𐟞𐟟𐟠𐟡𐟢𐟣𐟤𐟥𐟦𐟧𐟨𐟩𐟪𐟫𐟬𐟭𐟮𐟯𐟰𐟱𐟲𐟳𐟴𐟵𐟶𐟷𐟸𐟹𐟺𐟻𐟼𐟽𐟾𐟿𐠀𐠁𐠂𐠃𐠄𐠅𐠆𐠇𐠈𐠉𐠊𐠋𐠌𐠍𐠎𐠏𐠐𐠑𐠒𐠓𐠔𐠕𐠖𐠗𐠘𐠙𐠚𐠛𐠜𐠝𐠞𐠟𐠠𐠡𐠢𐠣𐠤𐠥𐠦𐠧𐠨𐠩𐠪𐠫𐠬𐠭𐠮𐠯𐠰𐠱𐠲𐠳𐠴𐠵𐠶𐠷𐠸𐠹𐠺𐠻𐠼𐠽𐠾𐠿𐡀𐡁𐡂𐡃𐡄𐡅𐡆𐡇𐡈𐡉𐡊𐡋𐡌𐡍𐡎𐡏𐡐𐡑𐡒𐡓𐡔𐡕𐡖𐡗𐡘𐡙𐡚𐡛𐡜𐡝𐡞𐡟𐡠𐡡𐡢𐡣𐡤𐡥𐡦𐡧𐡨𐡩𐡪𐡫𐡬𐡭𐡮𐡯𐡰𐡱𐡲𐡳𐡴𐡵𐡶𐡷𐡸𐡹𐡺𐡻𐡼𐡽𐡾𐡿𐢀𐢁𐢂𐢃𐢄𐢅𐢆𐢇𐢈𐢉𐢊𐢋𐢌𐢍𐢎𐢏𐢐𐢑𐢒𐢓𐢔𐢕𐢖𐢗𐢘𐢙𐢚𐢛𐢜𐢝𐢞𐢟𐢠𐢡𐢢𐢣𐢤𐢥𐢦𐢧𐢨𐢩𐢪𐢫𐢬𐢭𐢮𐢯𐢰𐢱𐢲𐢳𐢴𐢵𐢶𐢷𐢸𐢹𐢺𐢻𐢼𐢽𐢾𐢿𐣀𐣁𐣂𐣃𐣄𐣅𐣆𐣇𐣈𐣉𐣊𐣋𐣌𐣍𐣎𐣏𐣐𐣑𐣒𐣓𐣔𐣕𐣖𐣗𐣘𐣙𐣚𐣛𐣜𐣝𐣞𐣟𐣠𐣡𐣢𐣣𐣤𐣥𐣦𐣧𐣨𐣩𐣪𐣫𐣬𐣭𐣮𐣯𐣰𐣱𐣲𐣳𐣴𐣵𐣶𐣷𐣸𐣹𐣺𐣻𐣼𐣽𐣾𐣿𐤀𐤁𐤂𐤃𐤄𐤅𐤆𐤇𐤈𐤉𐤊𐤋𐤌𐤍𐤎𐤏𐤐𐤑𐤒𐤓𐤔𐤕𐤖𐤗𐤘𐤙𐤚𐤛𐤜𐤝𐤞𐤟𐤠𐤡𐤢𐤣𐤤𐤥𐤦𐤧𐤨𐤩𐤪𐤫𐤬𐤭𐤮𐤯𐤰𐤱𐤲𐤳𐤴𐤵𐤶𐤷𐤸𐤹𐤺𐤻𐤼𐤽𐤾𐤿𐥀𐥁𐥂𐥃𐥄𐥅𐥆𐥇𐥈𐥉𐥊𐥋𐥌𐥍𐥎𐥏𐥐𐥑𐥒𐥓𐥔𐥕𐥖𐥗𐥘𐥙𐥚𐥛𐥜𐥝𐥞𐥟𐥠𐥡𐥢𐥣𐥤𐥥𐥦𐥧𐥨𐥩𐥪𐥫𐥬𐥭𐥮𐥯𐥰𐥱𐥲𐥳𐥴𐥵𐥶𐥷𐥸𐥹𐥺𐥻𐥼𐥽𐥾𐥿𐦀𐦁𐦂𐦃𐦄𐦅𐦆𐦇𐦈𐦉𐦊𐦋𐦌𐦍𐦎𐦏𐦐𐦑𐦒𐦓𐦔𐦕𐦖𐦗𐦘𐦙𐦚𐦛𐦜𐦝𐦞𐦟𐦠𐦡𐦢𐦣𐦤𐦥𐦦𐦧𐦨𐦩𐦪𐦫𐦬𐦭𐦮𐦯𐦰𐦱𐦲𐦳𐦴𐦵𐦶𐦷𐦸𐦹𐦺𐦻𐦼𐦽𐦾𐦿𐧀𐧁𐧂𐧃𐧄𐧅𐧆𐧇𐧈𐧉𐧊𐧋𐧌𐧍𐧎𐧏𐧐𐧑𐧒𐧓𐧔𐧕𐧖𐧗𐧘𐧙𐧚𐧛𐧜𐧝𐧞𐧟𐧠𐧡𐧢𐧣𐧤𐧥𐧦𐧧𐧨𐧩𐧪𐧫𐧬𐧭𐧮𐧯𐧰𐧱𐧲𐧳𐧴𐧵𐧶𐧷𐧸𐧹𐧺𐧻𐧼𐧽𐧾𐧿𐨀𐨁𐨂𐨃𐨄𐨅𐨆𐨇𐨈𐨉𐨊𐨋𐨌𐨍𐨎𐨏𐨐𐨑𐨒𐨓𐨔𐨕𐨖𐨗𐨘𐨙𐨚𐨛𐨜𐨝𐨞𐨟𐨠𐨡𐨢𐨣𐨤𐨥𐨦𐨧𐨨𐨩𐨪𐨫𐨬𐨭𐨮𐨯𐨰𐨱𐨲𐨳𐨴𐨵𐨶𐨷𐨹𐨺𐨸𐨻𐨼𐨽𐨾𐨿𐩀𐩁𐩂𐩃𐩄𐩅𐩆𐩇𐩈𐩉𐩊𐩋𐩌𐩍𐩎𐩏𐩐𐩑𐩒𐩓𐩔𐩕𐩖𐩗𐩘𐩙𐩚𐩛𐩜𐩝𐩞𐩟𐩠𐩡𐩢𐩣𐩤𐩥𐩦𐩧𐩨𐩩𐩪𐩫𐩬𐩭𐩮𐩯𐩰𐩱𐩲𐩳𐩴𐩵𐩶𐩷𐩸𐩹𐩺𐩻𐩼𐩽𐩾𐩿𐪀𐪁𐪂𐪃𐪄𐪅𐪆𐪇𐪈𐪉𐪊𐪋𐪌𐪍𐪎𐪏𐪐𐪑𐪒𐪓𐪔𐪕𐪖𐪗𐪘𐪙𐪚𐪛𐪜𐪝𐪞𐪟𐪠𐪡𐪢𐪣𐪤𐪥𐪦𐪧𐪨𐪩𐪪𐪫𐪬𐪭𐪮𐪯𐪰𐪱𐪲𐪳𐪴𐪵𐪶𐪷𐪸𐪹𐪺𐪻𐪼𐪽𐪾𐪿𐫀𐫁𐫂𐫃𐫄𐫅𐫆𐫇𐫈𐫉𐫊𐫋𐫌𐫍𐫎𐫏𐫐𐫑𐫒𐫓𐫔𐫕𐫖𐫗𐫘𐫙𐫚𐫛𐫜𐫝𐫞𐫟𐫠𐫡𐫢𐫣𐫤𐫦𐫥𐫧𐫨𐫩𐫪𐫫𐫬𐫭𐫮𐫯𐫰𐫱𐫲𐫳𐫴𐫵𐫶𐫷𐫸𐫹𐫺𐫻𐫼𐫽𐫾𐫿𐬀𐬁𐬂𐬃𐬄𐬅𐬆𐬇𐬈𐬉𐬊𐬋𐬌𐬍𐬎𐬏𐬐𐬑𐬒𐬓𐬔𐬕𐬖𐬗𐬘𐬙𐬚𐬛𐬜𐬝𐬞𐬟𐬠𐬡𐬢𐬣𐬤𐬥𐬦𐬧𐬨𐬩𐬪𐬫𐬬𐬭𐬮𐬯𐬰𐬱𐬲𐬳𐬴𐬵𐬶𐬷𐬸𐬹𐬺𐬻𐬼𐬽𐬾𐬿𐭀𐭁𐭂𐭃𐭄𐭅𐭆𐭇𐭈𐭉𐭊𐭋𐭌𐭍𐭎𐭏𐭐𐭑𐭒𐭓𐭔𐭕𐭖𐭗𐭘𐭙𐭚𐭛𐭜𐭝𐭞𐭟𐭠𐭡𐭢𐭣𐭤𐭥𐭦𐭧𐭨𐭩𐭪𐭫𐭬𐭭𐭮𐭯𐭰𐭱𐭲𐭳𐭴𐭵𐭶𐭷𐭸𐭹𐭺𐭻𐭼𐭽𐭾𐭿𐮀𐮁𐮂𐮃𐮄𐮅𐮆𐮇𐮈𐮉𐮊𐮋𐮌𐮍𐮎𐮏𐮐𐮑𐮒𐮓𐮔𐮕𐮖𐮗𐮘𐮙𐮚𐮛𐮜𐮝𐮞𐮟𐮠𐮡𐮢𐮣𐮤𐮥𐮦𐮧𐮨𐮩𐮪𐮫𐮬𐮭𐮮𐮯𐮰𐮱𐮲𐮳𐮴𐮵𐮶𐮷𐮸𐮹𐮺𐮻𐮼𐮽𐮾𐮿𐯀𐯁𐯂𐯃𐯄𐯅𐯆𐯇𐯈𐯉𐯊𐯋𐯌𐯍𐯎𐯏𐯐𐯑𐯒𐯓𐯔𐯕𐯖𐯗𐯘𐯙𐯚𐯛𐯜𐯝𐯞𐯟𐯠𐯡𐯢𐯣𐯤𐯥𐯦𐯧𐯨𐯩𐯪𐯫𐯬𐯭𐯮𐯯𐯰𐯱𐯲𐯳𐯴𐯵𐯶𐯷𐯸𐯹𐯺𐯻𐯼𐯽𐯾𐯿𐰀𐰁𐰂𐰃𐰄𐰅𐰆𐰇𐰈𐰉𐰊𐰋𐰌𐰍𐰎𐰏𐰐𐰑𐰒𐰓𐰔𐰕𐰖𐰗𐰘𐰙𐰚𐰛𐰜𐰝𐰞𐰟𐰠𐰡𐰢𐰣𐰤𐰥𐰦𐰧𐰨𐰩𐰪𐰫𐰬𐰭𐰮𐰯𐰰𐰱𐰲𐰳𐰴𐰵𐰶𐰷𐰸𐰹𐰺𐰻𐰼𐰽𐰾𐰿𐱀𐱁𐱂𐱃𐱄𐱅𐱆𐱇𐱈𐱉𐱊𐱋𐱌𐱍𐱎𐱏𐱐𐱑𐱒𐱓𐱔𐱕𐱖𐱗𐱘𐱙𐱚𐱛𐱜𐱝𐱞𐱟𐱠𐱡𐱢𐱣𐱤𐱥𐱦𐱧𐱨𐱩	



ご覧のように、ネイティブなベクトルヒンティングはこのフォントにはさほど有効に働かないのですが、一方ラスタイメージではテキストの外観を大きく変化させています。

とはいえ、ある種のフォント、例えば以下の“Arial Unicode MS”などではベクトルヒンティングでも上手く行くこともあります。

abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
儸僨儀播僂倂僭倅僛僞僟僠僡僢僣僤僥僦僧僨僩僪僫僬僭僮僯僰僱僲僳僴僵

Adobe FLASH とは異なり、Scaleform はデバイスフォントの任意のアフィン変換をサポートしていることは特筆すべきです。回転や歪曲させるとグリフはぼやけてしまいましたが、アニメーテッドテキストに使うには十分な品質を保っています。

Win32 プロバイダーも、FreeType プロバイダーも、デフォルトで下記の値を使用します：

```
vectorRange = Font::DontHint;
rasterRange = Font::HintCJK;
maxVectorHintedSize = 24;
maxRasterHintedSize = 24;
```

ヒントイングを設定するための特別なインターフェイスについて、以下記述します。

#### 4.4.2 ネイティブヒンティングの設定

GFx::FontProviderWin32 は、デフォルトで「CJK」文字のネイティブなラスティングを使用するようになっており、ベクトルのネイティブヒンティングは（ ）内です。この挙動を変更するには、次の呼び出しを使います：

```
fontProvider->SetHintingAllFonts(. . .);
```

or

```
fontProvider->SetHinting(fontName, . . .);
```

SetHintingAllFonts ( )関数が SetHinting( )の後に呼び出された時には、( )内のフォントのヒンティングの挙動は変化しないことを特記しておいてください。模範的な関数群は以下のようになります：

```
void SetHintingAllFonts(Font::NativeHintingRange vectorRange,
                        Font::NativeHintingRange rasterRange,
                        unsigned maxVectorHintedSize=24,
                        unsigned maxRasterHintedSize=24);
```

```
void SetHinting(const char* Name,
               Font::NativeHintingRange vectorRange,
               Font::NativeHintingRange rasterRange,
               unsigned maxVectorHintedSize=24,
               unsigned maxRasterHintedSize=24);
```

Name パラメータには UTF-8 エンコーディングを使用できます。

## 4.5 Gfx::FontProviderFT2

このフォントプロバイダは、David Turner による FreeType-2 ライブラリを使用します。この使い方は、フォント名と属性を実際のフォントファイルにマップする必要があること以外は、Gfx::FontProviderWin32 と同様です。

開発者には、まず FreeType のマニュアルを参照し、ライブラリを正しく設定して構築することをお勧めします。スタティックリンクを使用するかダイナミックリンクを使用するか決定、さらに適切なランタイム設定 (フォントドライバ、メモリアロケータ、外部ファイルストリームなど) の決定は、開発者が行なわなければなりません。

スタティックリンクを伴う Windows MSVC コンパイラでは、以下のライブラリを使用します。

freetype<ver>.lib – リリースマルチスレッド DLL コード生成用

freetype<ver>\_D.lib – デバッグマルチスレッド DLL 用

freetype<ver>MT.lib – リリースマルチスレッド (スタティック CRT) 用

freetype<ver>MT\_D.lib – デバッグマルチスレッド (スタティック CRT) 用

ただし、“<ver>” は FreeType のバージョンです (例えば、バージョン 2.1.9 の場合は 219)。

また、ディレクトリ freetype2/include と freetype2/objs が、それぞれ追加のインクルードとライブラリのパスにあることを確認してください。

Gfx::FontProviderFT2 オブジェクトの作成は次のようになります。

```
#include "Gfx/Gfx_FontProviderFT2.h"
. . .
Ptr<FontProviderFT2> fontProvider = *new FontProviderFT2;
<Map Font to Files or Memory>
Loader.SetFontProvider(fontProvider);
```

このコンストラクタは、一個の引数を持ちます。

```
FontProviderFT2(FT_Library lib=0);
```

これは FreeType ライブラリハンドルです。ゼロ (デフォルト値) の場合は、プロバイダが内部的に FreeType を初期化します。アプリケーションがすでに FreeType を使用しており、Scaleform とその他のシステムの間でハンドルを共有する場合は、初期化された既存の外部ハンドラを指定する機能が提供されています。

ただし、外部ハンドラを使用する場合に、(FT\_Done\_FreeType) を呼び出して、開発者は、ライブラリを適切に解放してやる必要があります。また、アプリケーションは、ハンドルの寿命が Gfx::Loader の寿命より「長い」ことを保証する必要があります。FreeType のランタイムコールバックの再設定が望ましい場合にも (メモリアロケータなど)、外部のハンドラの使用がサポートされます。ただし、この設定は内部の初期化を使用して作成することもできます。fontProvider->GetFT\_Library() 関数は使用された FT\_Library ハンドラを返すので、実際にフォントを使用するまで、プロバイダからの FreeType の呼び出しが行われないことが保証されます。したがって、Gfx::FontProviderFT2 を作成した後、実行時に FreeType を再設定することができます。

### 4.5.1 FreeType フォントのファイルへのマッピング

Win32 API とは異なり、FreeType は タイプフェイス名 (および属性) から実際のフォントファイルへのマッピングを提供していません。したがって、このマッピングは外部的に提供されることになります。Gfx::FontProviderFT2 は、フォントをファイルとメモリにマッピングするシンプルなメカニズムを持っています。

```
void MapFontToFile(const char* fontName, unsigned fontFlags,
                  const char* fileName, unsigned faceIndex=0
                  Font::NativeHintingRange vectorHintingRange = Font::DontHint,
                  Font::NativeHintingRange rasterHintingRange = Font::HintCJK,
                  unsigned maxVectorHintedSize=24,
                  unsigned maxRasterHintedSize=24);
```

引数 “fontName” は、フォントのタイプフェイス名 (例えば “Times New Roman”) を指定します。引数 “fileName” は、フォントファイルへのパス (例えば “C:\\WINDOWS\\Fonts\\times.ttf”) を指定します。“fontFlags” は、“Font::FF\_Bold”、“Font::FF\_Italic”、“Font::FF\_BoldItalic” のいずれかの値をとります。値 “Font::FF\_BoldItalic” は、実際には “Font::FF\_Bold | Font::FF\_Italic” と同等です。“faceIndex” は FT\_New\_Face に渡されます。ほとんどの場合、このパラメーターは 0 ですが、常にというわけではありません。あくまでもフォントファイルのタイプとコンテンツに依存しています。この関数は実際にフォントファイルを開くわけではありません。マッピングテーブルを構成するだけです。フォントファイルは、実際に要求されたときに開きます。Win32 フォントプロバイダーとは違って、FreeType フォントプロバイダーでは関数の引数がネイティブヒンティングを設定することに使われます。

### 4.5.2 フォントのメモリへのマッピング

FreeType では、フォントをメモリにマップできます。例えば、一個のフォントファイルに多くのタイプフェイスが含まれているような場合や、フォントファイルがアプリケーションによってすでにロードされているような場合には、有効だと思われます。

```
void MapFontToMemory(const char* fontName, unsigned fontFlags,
                    const char* fontData, unsigned dataSize, unsigned faceIndex=0
                    Font::NativeHintingRange vectorHintingRange = Font::DontHint,
                    Font::NativeHintingRange rasterHintingRange = Font::HintCJK,
                    unsigned maxVectorHintedSize=24,
                    unsigned maxRasterHintedSize=24);
```

単純な（あまり良い例ではありませんが）フォントのメモリへのマッピングの例を以下に示します。

```
FILE* fd = fopen("C:\\WINDOWS\\Fonts\\times.ttf", "rb");
if (fd) {
    fseek(fd, 0, SEEK_END);
    unsigned size = ftell(fd);
    fseek(fd, 0, SEEK_SET);
    char* font = (char*)malloc(size);
    fread(font, size, 1, fd);
    fclose(fd);
    fontProvider->MapFontToMemory("Times New Roman", 0, font, size);
}
```

この例ではメモリは解放されません（メモリリークが発生してしまいます）。マッピングテーブルはフォントデータへのそのままの定数ポインタのみを保存するためです。開発者は、メモリを適切に解放してやらなければなりません。アプリケーションは、このメモリブロックの寿命が Gfx::Loader の寿命より「長い」ことを保証する必要があります。フォントマッピングメカニズムは、可能な限りの自由を提供するように設計されています。例えば、アプリケーションはすでに事前にロードされたメモリフォントで FreeType を使用しており、割り当てと解除は Scaleform から外部的に処理されているともいえるでしょう。

Windows で FreeType フォントマッピングを使用する一つの例を以下に示します。

```
Ptr<FontProviderFT2> fontProvider = *new FontProviderFT2;
fontProvider->MapFontToFile("Times New Roman", 0,
                           "C:\\WINDOWS\\Fonts\\times.ttf");
fontProvider->MapFontToFile("Times New Roman", Font::FF_Bold,
                           "C:\\WINDOWS\\Fonts\\timesbd.ttf");
fontProvider->MapFontToFile("Times New Roman", Font::FF_Italic,
                           "C:\\WINDOWS\\Fonts\\timesi.ttf");
fontProvider->MapFontToFile("Times New Roman", Font::FF_BoldItalic,
                           "C:\\WINDOWS\\Fonts\\timesbi.ttf");

fontProvider->MapFontToFile("Arial", 0,
                           "C:\\WINDOWS\\Fonts\\arial.ttf");
fontProvider->MapFontToFile("Arial", Font::FF_Bold,
                           "C:\\WINDOWS\\Fonts\\arialbd.ttf");
fontProvider->MapFontToFile("Arial", Font::FF_Italic,
                           "C:\\WINDOWS\\Fonts\\ariali.ttf");
fontProvider->MapFontToFile("Arial", Font::FF_BoldItalic,
                           "C:\\WINDOWS\\Fonts\\arialbi.ttf");

fontProvider->MapFontToFile("Verdana", 0,
                           "C:\\WINDOWS\\Fonts\\verdana.ttf");
fontProvider->MapFontToFile("Verdana", Font::FF_Bold,
                           "C:\\WINDOWS\\Fonts\\verdanab.ttf");
fontProvider->MapFontToFile("Verdana", Font::FF_Italic,
                           "C:\\WINDOWS\\Fonts\\verdanai.ttf");
fontProvider->MapFontToFile("Verdana", Font::FF_BoldItalic,
```

```
"C:\\WINDOWS\\Fonts\\verdanaz.ttf");
```

```
. . .
```

```
Loader.SetFontProvider(fontProvider);
```

ただし、これは一例に過ぎません。実際のアプリケーションで絶対的なハードコードのファイルパスを指定するのは、良くありません。設定ファイルから、またはフォントのタイプフェイスを自動スキャンすることによって取得するのが一般的です。フォントには通常タイプフェイス名が含まれるので、明示的にタイプフェイス名を指定するのは行き過ぎに見えるかもしれません。しかし、場合によってはコストのかかるファイルの解析操作が回避されます。関数 `MapFontToFile()` はこの情報を格納するだけで、要求がない限りファイルは開きません。結果として、マップされたフォントの多くを指定でき、そのほんのいくつかだけが実際に使用されます。この場合、余計なファイル操作が実行されるようなことはありません。

## 5 パート 4:フォント レンダリングの設定

Scaleform では、テキスト文字を二通りの方法でレンダリングできます。一つの方法では、フォントグリフをテクスチャとしてラスターライズし、後でテクスチャ トライアングル(1 文字あたり 2 個)のバッチを使用して描画することができます。もう一つの方法では、フォントグリフをトライアングルメッシュにテッセレートし、ベクトルシェイプとしてレンダリングすることもできます。

アプリケーション内のほとんどのテキストでは、パフォーマンス上の理由から、常にテクスチャ トライアングルを使用する必要があります。グリフテクスチャは、ダイナミックキャッシングまたはスタティックキャッシングによって生成されます。グリフのダイナミックキャッシングは柔軟性が高く、高品質のイメージを生成できます。他方、スタティックキャッシングにはオフラインで計算できるという利点があります。

開発者は、ターゲットプラットフォームとタイトルのニーズに従って、以下のフォントレンダリングのオプションを選択することができます。

1. ダイナミックキャッシュの使用。高速のアニメーションと高品質な出力の両方が得られます。ダイナミックキャッシュは一定量のテクスチャ メモリを使用しており、すべてのグリフをラスターライズまたはロードする必要がないので、ロード時間が短縮されます。
2. スタティックキャッシュの使用 (事前に生成され、mip-map 化されたテクスチャをディスクからロード)。開発者は、事前に gfxexport ツールを使用してパックされたグリフテクスチャを生成でき、フォントのベクトルデータをストリッピングできるので、ロードする必要がなくなります。
3. スタティックキャッシュの使用 (ロード時に Gfx::FontPackParams でテクスチャを自動生成)。前の方法と同様ですが、テクスチャをディスクからロードする必要はありません。
4. ベクトルシェイプを使用してテキストグリフを直接レンダリングします。

PC、Xbox 360、PS3 などのハイエンドシステムや、Wii でも多くの場合、ダイナミックキャッシュの使用をお勧めします。ダイナミックキャッシュでは、最短のロード時間と、所定の解像度で可能な最高品質のフォント レンダリングが保証されます。

Gfx::FontProviderWin32 または Gfx::FontProviderFT2 によるシステムまたは外部のフォントサポートを利用する場合も、ダイナミックキャッシュを使用する必要があります。

スタティックキャッシュで gfxexport ツールで書き出されたテクスチャを使用するのは、CPU とメモリに制約のあるプラットフォーム (PSP や PS2 など) に適した方法です。ランタイムでグリフをラスターライズするのは負担が大きすぎます。開発者がレンダラにおける動的なテクスチャ更新を避けたい場合も、スタティックキャッシュを使用します。ただし、ベクトルデータのメモリ使用量は最適化されているため、低性能システムでもダイナミックキャッシュの方がよい場合もあります。開発者がゲームデータで試験した上で、最適なソリューションを決定することをお勧めします。

Scaleform では設定することも可能ですが、ベクトルシェイプを単独で使用してテキストレンダリングを行うことはほとんどありません。代わりに、大きなグリフのみにグリフテッセレーションが適用され、小さなテキストの効率的なレンダリングを行うためのテクスチャベースの方法と併用されます。このオプションを設定または無効化する方法の詳細は、「ベクトル化の設定」のセクションを参照してください。

## 5.1 フォントキャッシュマネージャ

Scaleform でのフォントのレンダリングの設定はステートオブジェクトの `Render::GlyphCacheConfig` または `GFx::FontPackParams` で行います。デフォルトではグリフのキャッシュはレンダースレッド上の `Renderer2D` オブジェクトで自動的に生成され、またダイナミックグリフキャッシングに初期化されます。グリフパッカーは、スタティックテキストチャの初期設定にのみ使用します。パックパラメータはデフォルトでは `NULL` です。この設定では、作成されたすべてのムービーに、自動的にダイナミックフォントキャッシュが使用されます。スタティックテキストチャを含むように前処理された `GFX` ファイルからムービーが取得される場合には、この限りではありません。

### GFx 4.0 アップグレードメモ

ダイナミックグリフのキャッシュ設定は Scaleform のバージョン 3.x と 4.0 の間で大きく変更されました。GFx 3.3 は `GFxLoader` の維持する `GFxFontCacheManager` オブジェクトに依存して、メインスレッドで設定されていましたが、GFx 4.0 ではこれが `Renderer2D` が維持してレンダースレッドで設定される `GlyphCacheConfig` インターフェイスに替わりしました。Scaleform GFx 4.0 でのキャッシュの設定に関する詳細はセクション 5.2 を参照してください。

テキストチャからラスターグリフがレンダーされる時は、グリフキャッシュシステムは必ず使用されます。キャッシュマネージャの役割は、スタティックとダイナミックのいずれの場合でも作成されるテキストバッチの頂点配列を、内部的に保持することです。ダイナミックキャッシングが有効な場合は、キャッシュマネージャがキャッシュテキストチャの割り当てを行い、ラスターライズされた文字でテキストチャを更新し、バッチの頂点データと同期するテキストチャを保持します。スタティックキャッシングだけを使用する場合にも、キャッシュマネージャはテキストの頂点配列を保持する必要がありますが、ダイナミックテキストチャの割り当てや更新の必要はありません。

スタティックキャッシュは、密にパックされたグリフによって事前にラスターライズされたビットマップテキストチャで表されます。スタティックテキストチャのラスターライズとパックは、`GFx::FontPackParams` に基づいてロード時に行うことも、`'gfxexport'` ツールを使用してオフラインで行うこともできます。いずれの場合も、スタティックテキストチャはフォントに関連付けられます。ロードする方法によって、埋め込みフォントにフォントグリフを含むスタティックテキストチャのセットが含まれる場合と含まれない場合があります。フォントにスタティックテキストチャが含まれる場合は、常にテキストチャを使用してフォントのグリフがレンダリングされ、それ以外の場合はダイナミックキャッシュ (有効にしてある場合ですが) が使用されます。

さらに、使用するフォントテキストチャのタイプによって、レンダリング方法はグリフのデスティネーション ピクセルサイズにも依存します。形式的には以下のロジックが使用されます。

```
bool Done = false;

if (Font has Static Textures with Packed Glyphs)
{
    if (GlyphSize <
        FontPackParams.TextureConfig.NominalSize * MaxRasterScale)
    {
        Draw the Glyph as a Texture using Static Cache;
        Done = true;
    }
}
```

```

    }
}
else if (Dynamic Cache is Enabled AND
        GlyphSize < GlyphCacheParams.MaxSlotHeight)
{
    Draw the Glyph as a Texture using Dynamic Cache;
    Done = True;
}

if (Not Done)
{
    Draw the Glyph as Vector Shape;
}

```

前述のように、テクスチャキャッシュが使用できない場合、またはグリフが大きすぎてテクスチャからレンダリングできない場合はベクトルレンダリングを使用します。キャッシュのアプローチは、フォント内にパックされたグリフテクスチャがあるかどうかに基づいて選択されます。二つのアプローチを設定する方法については、以下のセクションで詳しく設定します。

## 5.2 ダイナミックフォントキャッシュの使用

フォントや文字セットが制限されている場合、例えば、基本ラテン、ギリシャ文字、キリル文字などの場合は、スタティックキャッシュが効率的です。しかし、多くのアジア言語では、スタティックキャッシュは非常に多くのシステムメモリとビデオメモリを使用するので、ロード時間が長くなります。多くの異なるタイプフェイスを使用する場合、つまり埋め込みグリフの総数が多い場合も (例えば、10000以上) 同様です。ダイナミックキャッシュメカニズムの使用は、このような場合に意味があります。さらに、ダイナミックキャッシュは読みやすくするための最適化など、多くの機能を提供し、品質を著しく向上させます。ダイナミックキャッシュはデフォルトで有効です。デフォルトではダイナミックキャッシュはイネーブルされ、そのバッファを割り当てます。これをディスエーブルするには次のように呼び出しをします。

```
renderer->GetGlyphCacheConfig()->SetParams(Render::GlyphCacheParams(0));
```

上の呼び出しで、renderer はレンダースレッドで維持されている Render::Renderer2D オブジェクトです。この呼び出しはグリフキャッシュの使用するダイナミックテクスチャーの数をゼロにし、実効的にこれをディスエーブルします。デフォルトの一時バッファ割り当てを避けるには、この呼び出しはレンダー、HAL の初期化前に行う必要があります。

ダイナミックキャッシュは、テキストを描画する場合に、要求に応じてグリフをラスタライズし、それぞれのテクスチャを更新します。単純な LRU (Least Recently Used) キャッシュスキームを使用しますが、テクスチャへのスマートアダプティブグリフパッキングが「その場で」実行されます。テクスチャパラメータは次のように設定できます。



```

Render::GlyphCacheParams gcparams;
gcparams.TextureWidth    = 1024;
gcparams.TextureHeight   = 1024;
gcparams.MaxNumTextures  = 1;
gcparams.MaxSlotHeight   = 48;
gcparams.SlotPadding     = 2;
gcparams.TexUpdWidth     = 256;
gcparams.TexUpdHeight    = 512;

renderer->GetGlyphCacheConfig()->SetParams(gcparams);

```

デフォルトでは、上の値が使用されます。

`TextureWidth`, `TextureHeight` - キャッシングテクスチャのサイズ。いずれの値もそれを上回る最小の 2 のべき乗に切り上げられます。

`MaxNumTextures` - キャッシングに使用するテクスチャ数の上限。

`MaxSlotHeight` - グリフの高さの最大値。実際のグリフの高さのピクセル値は、これ以下になります。これより大きいグリフは、ベクトルシェイプとしてレンダリングされます。

`SlotPadding` - グリフのクリッピングとオーバーラッピングを防止するためのマージンの値。実用的には、多くの場合 2 で十分です。

`TexUpdWidth`, `TexUpdHeight` - テクスチャの更新に使用するイメージのサイズ。通常、実用的にはすべての場合に 256x512 (128K のシステム メモリ) で十分です。256x256 または 128x128 に縮小することもできますが、この場合はテクスチャの更新がより頻繁に発生します。



ダイナミックグリフキャッシュは可能な限り密に、グリフをテクスチャ内にパックしようとしています。これはダイナミックに行われ、LRU キャッシュ 手法に従っています。フォントグリフのキャッシングの動作を理解するには、ある 1MB 空間中で 4KB だけのブロックのメモリーを割り当てることができる非常に簡単なメモリー割り当て子であると考えてください。アロケータはこの 4 K ブロックのみの割り当て/割り当て解除を行うことができます。このアロケータには最大で 256 の同時割り当てを行う能力があることは明らかです。ただしほとんどの場合、要求されるサイズは 4 K よりもはるかに小さく、16 バイト、100 バイト、256 バイトなどであり 4K を超えることはありません。このような場合、その 4K ブロック内でさらに小さなメモリー ブロックを処理して、指定された 1 MB のスペース内でより大きな割り当て容量を得るための何らかのメカニズムを開発者は考案しようとするでしょう。ただし、保証される最小限度は同じままです。256 の同時割り当てそのままです。同様のメカニズムはダイナミックグリフキャッシュでは動作しますが、それは 2 次元のテクスチャ スペースにおいてです。言い換えると、ダイナミックグリフキャッシュは  $\text{MaxSlotHeight} + 2 * \text{SlotPadding}$  の 2 乗を保管する容量を保証するということです。しかし、グリフが小さな場合は特にそうですが、このマネージャはグリフをもっと密にパックしようとしています。平均的な場合、これはダイナミック フォント キャッシュの容量を 2-5 倍増やすことになります (非常に小さなグリフの場合、この容量の増加は何十倍にもなる可能性があります)。通常、グリフのサイズに依存せずに、テクスチャ スペースの 60-80% がペイロードに使用されます。

総キャッシュ容量 (キャッシュに同時に格納される異なるグリフの最大数) は、グリフサイズの平均によって変わります。通常のゲーム UI の場合は、上のパラメータで約 500~2000 個の異なるグリフをキャッシングできます。キャッシングされるグリフの最大数は、任意の 1 つのテキストフィールドの表

示部分进行处理するのに十分な値でなければなりません。この場合は、「1 つの」テキストフィールドの「表示部分」に含まれる「異なる」グリフの数がキャッシュ容量を超える場合は、残りのグリフがベクトルシェイプとして描画されます。

前述のように、ダイナミックキャッシュを使用すると機能が拡張され、「アンチエイリアス (読みやすさ優先)」オプションと「アンチエイリアス (アニメーション優先)」オプションが提供されます。これらのオプションは、Flash のデザイナーがテキストダイアログパネルで選択できるテキストフィールドのプロパティです。「読みやすさ」を最適化されたテキストは、より鮮明で容易に判読できます。アニメーションも可能ですが、こうしたアニメーションではテキストの更新操作の頻度が高くなるので、負担も大きくなります。さらに、グリフは自動的にピクセルグリッドにはめ込まれ、ピクセルの自動はめ込み操作によってぼかしが軽減されます。つまり、テキストラインもピクセルにはめ込まれます。視覚的には、アニメーション中、特にスケール変更中にジッタ効果があります。

下の図は、こうしたオプションの違いを示しています。

GFX, Dynamic cache	Readability	Animation
<i>Anti-alias for readability</i> <i>Anti-alias for animation</i>		

「アンチエイリアス (読みやすさ優先)」が使用された場合、グリフラスタイザは自動的なはめ込み手順を実行します (自動ヒンティングとも呼ばれる) を実行します。Scaleform では、Flash ファイルまたは他の任意のフォントソースからのグリフヒントは一切使用されません。テキストのレンダリングに必要なのはグリフの外形だけです。フォントのソースに関係なく、テキストの外観は同等です。Scaleform 自動ヒンタは、フォントから特定の情報、すなわち上部が平坦なラテン文字の高さを要求します。これは、はみ出したグリフ (O、G、C、Q、o、g、e など) を正しくはめ込むために必要です。通常、フォントからは、こうした情報は得られません。したがって、何とかして推測する必要があります。このために、Scaleform では上部が平坦なラテン文字を使用します。これは次のとおりです。

大文字の場合は H、E、F、T、U、V、W、X、Z

小文字の場合は z、x、v、w

自動はめ込み機能を使用するために、フォントには上にあるラテン文字から、少なくとも 1 つの大文字と少なくとも 1 つの小文字が含まれている必要があります。もし、含まれていないと、Scaleform が次のログ警告を生成します。

**“Warning: Font 'Arial': No hinting chars (any of 'HEFTUVWXZ' and 'zxvwy').Auto-Hinting is disabled.”**

この警告が表示された場合は、Flash のデザイナーは、どれかの文字を埋め込まなければなりません。通常は [文字の埋め込み] ダイアログのフォームで “Zz” だけを追加すれば十分です。「基本ラテン」を埋め込んでおかまいません。

## 5.3 フォントのコンパクター、*gfxexport* の使用

コマンドラインの *gfxexport* ツールの目的は SWF ファイルを事前に処理し、ゲームに配布、ロードする GFX ファイルを生成することです。プリプロセスの間、このツールは SWF ファイルから画像をストリップし、外部ファイルに抽出します。外部ファイルは DDS や TGA といったいくつかの便利なフォーマットで保存できます。-fc オプションを選択した場合、*gfxexport* はまたフォントベクトルのデータも圧縮します。これは損失性の圧縮ですので、パラメータをいくつか試して見てメモリー使用量と品質の兼ね合いを見る必要があります。

コマンドラインのオプション	ビヘイビア
-fc	フォント圧縮を有効にします。
-fcl <size>	名目上のグリフサイズを設定します。サイズを小さくするとデータサイズは小さくなりますが、グリフの正確さは損なわれます。デフォルトの値は 256 です。多くの場合名目上のサイズを 256 にしておくと、目に見える質の劣化なくメモリーを約 25%節約できます（Flash で一般的に使用されている 1024 の名目上のサイズにくらべて）。しかし、非常に大きなグリフには名目上のサイズを大きくすると良いでしょう。
-fcm	圧縮されたフォントのエッジをマージします。FontCompactor が同じ輪郭とグリフをマージするべきかどうかを示すブール値のフラグです。マージされたとき、フォントによってはデータはさらに圧縮されて、メモリーを 10~70%節約します。しかし、フォントにグリフが多すぎる場合、ハッシュテーブルは追加のメモリーを消費することがあり、これは各ユニークなパス当たり 12 バイト（32 ビット）または 16 バイト（64 ビット）、と加えて各ユニークなグリフ当たり 12 バイト（32 ビット）または 16 バイト（64 ビット）です。

## 5.4 フォント テクスチャの前処理 - *gfxexport*

*gfxexport* に -fonts オプションを指定すると、パックされたフォントテクスチャをラスタライズして書き出し、ユーザが指定した形式で保存することもできます。Scaleform Player に GFX ファイルをロードすると、外部テクスチャが自動的にロードされ、テキストをレンダリングするときにスタティックキャッシュとして使用されます。

**普通の GFx アプリケーションにはエクスポートされたテクスチャの使用は推奨しません。**しかし、低価格のモバイルプラットフォームや、いくつかの特別な場合にはこれが良いこともあります。

*gfxexport* を使用してフォントテクスチャを生成する方法には、次の利点があります。

外部テクスチャを保存することでグリフベクトルデータがストリッピングされ、メモリが節約されます。ただし、グリフデータは代わりにロードされたスタティックテクスチャに置き換えられます。

CPU に制約のあるシステムでは、テクスチャファイルのロードは、ロード時に生成するより短時間で実行できます。

テクスチャファイルをコンパクトで、ゲームに適した形式に変換でき、Gfx::ImageCreator よりも優先してロードすることができます。これは、こうした形式を直接サポートする独自の Render::Renderer を実装する場合に有利です。

ただし、事前に生成されたテクスチャはテキストレンダリングの品質が劣り、ダイナミックキャッシュを使用する場合よりロード時間が長くなる可能性があるという欠点があります。静止テキストでは mip-map を使用するのでヒントは使えません。つまり、「アンチエイリアス (読みやすさ優先)」設定は適用できないことになります。

次のコマンドラインステートメントは、'test.swf' を前処理して 'test.gfx' を生成し、全埋め込みフォント用に追加のテクスチャファイルを生成します。

```
gfxexport -fonts -strip_font_shapes test.swf
```

strip\_font\_shapes オプションを指定すると、生成された Scaleform ファイルから埋め込みフォントのベクトル データが除外されます。これでメモリが節約されますが、大きな文字用のベクトルレンダリングに変更できなくなるので、フォントテクスチャの名目グリフサイズより大きく引き伸ばしてしまうと品質低下を惹き起こします。

フォントに関連する gfxexport のオプションを次の表に示します。テクスチャサイズ、名目グリフサイズ、ターゲットファイル形式を設定するオプションが示されています。こうしたオプションの多くは、次のセクションで説明するグリフパッカーのパラメータに対応しています。グリフパッカーは、gfxexport でフォントテクスチャの生成に使用します。

コマンドライン オプション	動作
-fonts	フォントテクスチャを書き出します。指定しないとフォントテクスチャは生成されません (代わりにロード時にダイナミックキャッシュまたはパッキングが実行されます)。
-fns <size>	テクスチャグリフの名目サイズ (ピクセル単位)。指定しないと、デフォルト値 48 が使用されます。名目サイズはテクスチャ内の 1 つのグリフの最大サイズです。これより小さい文字は実行時にトリリニア mip-map フィルタを使用してレンダリングされます。
-fpp <n>	個々のグリフイメージの周囲に配置するスペース。デフォルト値は 3 です。
-fts <WxH>	グリフがパックされるテクスチャの横縦サイズ。デフォルトサイズは 256x256 です。正方形のテクスチャの場合は、1 つのサイズで指定できます (たとえば、'-fts 128' は 128x128 を意味します)。'-fts 512x128' は長方形のテクスチャのサイズを表します。
-fs	フォントごとに強制的に別のテクスチャを使用します。デフォルトでは、複数のフォントがテクスチャを共有します。

コマンドライン オプション	動作
- strip_font_shapes	生成される GFX ファイルにフォントのグリフシェイプデータが書き込まれません。
-fi <format>	フォントテクスチャの出力形式を指定します。ただし、<format> は TGA8 (grayscaled)、TGA24 (grayscaled)、TGA32 あるいは DDS8 のいずれかです。デフォルトでは、イメージ形式 (-i オプション) が TGA の場合はフォントテクスチャに TGA8 が使用されます。それ以外の場合は DDS A8 が使用されます。

**警告** 同梱のスタティクなグリフのみ、もしくはゲーム同梱のグリフのみを使用するならば、セクション 5.2 n 方法でダイナミックグリフキャッシュテクスチャーの割り当てをディスエーブルしてください。これを行っておかなければ、デフォルトのテクスチャーが割り当てられて使用されないままになります。

## 5.5 フォント グリフ パッカーの設定

フォントグリフパッカーは、ファイルをロードするときに埋め込みグリフをラスタライズしてパックします。GFxExport でフォントを事前にラスタライズすることもできます。前述のように、フォントキャッシュマネージャではダイナミックとスタティクの両方のメカニズムを同時に使用することができます。例えば、大きな文字セットのフォントではダイナミックキャッシュを使用し、基本ラテンのフォントは事前にラスタライズして静的に使用するよう、フォントグリフパッカーを設定することもできます。

既に述べたように、一般的な手法は、可能な場合に事前にラスタライズされたスタティクテクスチャを使用し、それ以外の場合にダイナミックキャッシュ (有効にしてある場合) を使用します。

フォントグリフパッカーはデフォルトでは無効なので、Scaleform では、パックパラメータを明示的に作成してローダ上で設定しておかない限り、すべての埋め込みフォントにダイナミックキャッシュが使用されることになります。パックパラメータの作業は、次のステートメントで実行できます。

```
Ptr<FontPackParams> packParams = *new FontPackParams();
Loader.SetFontPackParams(packParams);
```

ただし、GFxExport (および個々の .Scaleform ファイル) を使用する場合は、事前にグリフをラスタライズしておけます。上の呼び出しは、単に「ロード時にグリフをパックしない」ことを意味しています。

GFxExport でグリフが事前にラスタライズされている場合は、スタティクキャッシュとして使用されます。

スタティクキャッシュが望ましい場合 (埋め込みグリフの総数が少ない場合) は、次のように設定できます。

```
Loader.GetFontPackParams()->SetUseSeparateTextures(Bool flag);
Loader.GetFontPackParams()->SetGlyphCountLimit(int lim);
Loader.GetFontPackParams()->SetTextureConfig(fontPackConfig);
```

SetUseSeparateTextures() はパッキングを制御します。これが "true" の場合は、パッカーがフォントごとに別々のテクスチャを使用します。それ以外の場合は、すべてのグリフをできるだけコンパクトにパックしようとします。別々のテクスチャを使用するとテクスチャの切り替え回数、したがって描画プ

リミティブの数を減らすことができます。しかし、通常はシステムメモリとビデオメモリの使用量が増大します。デフォルトでは "false" です。

`SetGlyphCountLimit()` は、パックするグリフ数の上限を設定します。デフォルトは 0 (無制限) です。フォントの埋め込みグリフの総数がこの上限を超える場合は、フォントがパックされません。このパラメータは、アジア言語をラテンベースの文字などと併用する場合に意味があります。この上限を 500 に設定すると、アジア言語のフォントのほとんどがダイナミックにキャッシングされますが (ダイナミックキャッシュが有効な場合)、一部のグリフではスタティックテクスチャが使用されます。

`SetTextureConfig()` は、以下のようなテクスチャ パラメータを設定します。

```
FontPackParams::TextureConfig fontPackConfig;
fontPackConfig.NominalSize      = 48;
fontPackConfig.PadPixels        = 3;
fontPackConfig.TextureWidth     = 1024;
fontPackConfig.TextureHeight    = 1024;
```

上の値はデフォルトで使用されます。

**NominalSize** - テクスチャに格納されるアンチエイリアス処理されたグリフの名目サイズ (ピクセル単位)。このパラメータは、テクスチャ内で最大のグリフのサイズを指定します。多くのグリフはこれよりも、かなり小さくなります。このパラメータは、テクスチャによる RAM の使用量と大きなテキストの鮮明さとのバランスも制御します。これは、"NominalSize" と呼ばれます。グリフスロットに合わせて引き伸ばすダイナミックキャッシュとは異なり、スタティックキャッシュでは実際のバウンディングボックスを使用して別のサイズのグリフをパックします。NominalSize は、テキストの高さ (ピクセル) を設定するのと全く同じ値です。これは、ダイナミックキャッシュはスタティックキャッシュより「解像度」の点で、やや優れていることも示しています。

**PadPixels** - 個々のグリフイメージの周囲に配置するスペース。1 以上を指定します。この値を大きくすると、縮小されたテキストの境界が滑らかになりますが、多くのテクスチャのスペースが無駄になります。

**TextureWidth, TextureHeight** - グリフがパックされるテクスチャのサイズ。2 のべき乗を指定する必要があります。

いくつかの使用例のシナリオとその意味を以下に示します。

- 1) すべてをデフォルトに設定します。ダイナミックキャッシュは、有効。フォントグリフパッカーは使用しません。前処理された .GFX ファイルから事前にラスタライズされたグリフテクスチャをロードする場合以外、すべてのケースでダイナミック キャッシュを使用します。
- 2) 

```
Ptr<FontPackParams> packParams = *new FontPackParams();
Loader.SetFontPackParams(packParams);
...
```

`renderer->GetGlyphCacheConfig()->SetParams(Render::GlyphCacheParams(0));` すべてのケースでフォントグリフパッカーを使用します。ダイナミックキャッシュは無効。
- 3) 

```
Ptr<FontPackParams> packParams = *new
FontPackParams();Loader.SetFontPackParams(packParams);
Loader.GetFontCacheManager()->EnableDynamicCache(false);
```

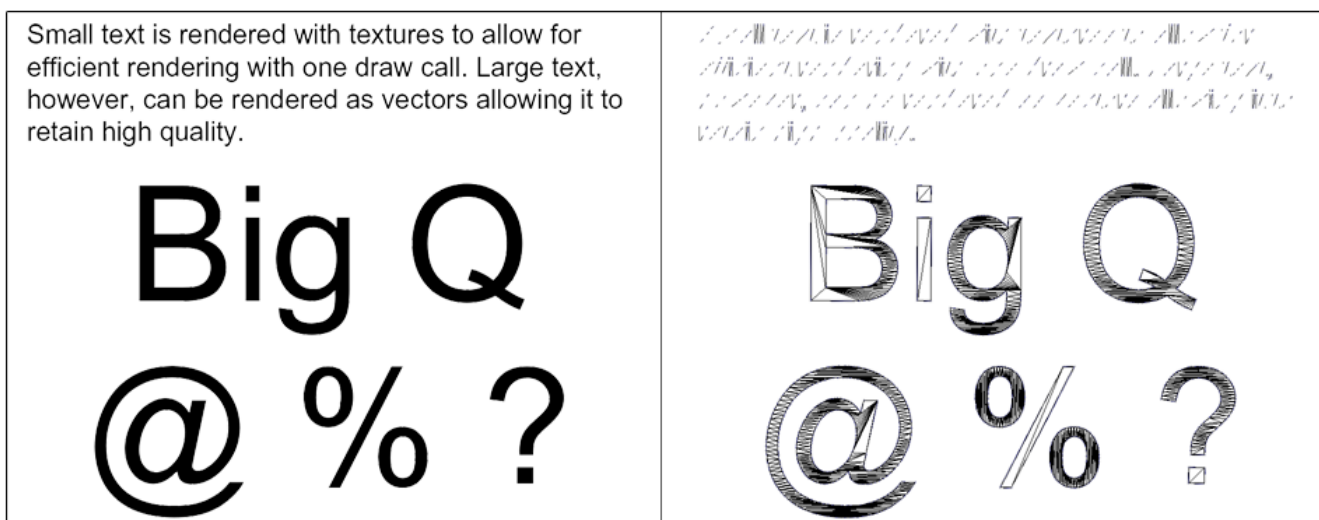
フォントグリフパッカーとダイナミックキャッシュの両方を使用します。ここでは、500 個以下のグリフが埋め込まれたフォントには事前にラスタライズされたスタティックテクスチャを使用し、それ以外のフォントにはダイナミックキャッシュを使用します。

ダイナミックキャッシュでしか使用できない機能があることにも注意する必要があります。例えば、ピクセルグリッドへの自動的なはめ込み (いわゆる自動ヒンティング) によるテキストの読みやすさに関する最適化は、ダイナミックキャッシュでのみ機能します。ほかし、シャドウ、グローなどのエフェクトも、ダイナミックキャッシュだけの機能です。大まかに言えば、グリフパッカーとスタティックキャッシュを組み合わせた手法は、性能とメモリに制約のあるシステムに適しているでしょう。

## 5.6 ベクトル化の制御

本書ですでに説明したように、Scaleform ではテクスチャキャッシュに収まらない大きなグリフをレンダリングする場合に、代替策としてテッセレーションされたベクトルシェイプを使用します。Flash では、テキスト文字のサイズに制限がないので、テクスチャを使用したレンダリングに必要なメモリ容量が、あるポイントを法外に超えてしまうことがあります。この問題に対処する 1 つの方法は、グリフのビットマップサイズを制限し、このポイントを超える場合にバイリニアフィルタを使用することです。残念ながら、これではレンダリング品質が直ちに低下します。

Scaleform では、大きなテキストをレンダリングするために、グリフのレンダリングを、トライアングルシェイプに切り替え、さらにエッジのアンチエイリアス技術を使用して高品質の出力を実現します。ほとんどの場合に、テキスト文字のサイズが大きくなってもエッジは滑らかな状態に維持されるため、切り替えはユーザには認識されません。下の図は、テキストのレンダリングでテクスチャによるものとトライアングルメッシュによるものとの違いを示しています。Scaleform Player の実行ファイルでは、Ctrl+W キーを使用してワイヤフレームモードに切り替えると、レンダリングがどのように実行されるかを確認することができます。



トライアングルを使用してレンダリングされたグリフはきれいに見えますが、トライアングルと描画プリミティブの数が増大するので、レンダリングの処理時間は長くなります。ダイナミックキャッシュを使用してグリフをレンダリングする場合は、グリフの高さの最大値がスロットの高さの最大値 (MaxSlotHeight) によって定義されます。Render::GlyphCacheConfig::SetParams を使って、この MaxSlotHeight を変更することが可能です。したがって、グリフがテクスチャスロットに収まる場合は

テクスチャとしてレンダリングされ、それ以外の場合はベクトルレンダリングが使用されます。このため、テキストのサイズが最大値に近い場合は、1 行のテキストにビットマップとベクトルの両方のシンボルが含まれる可能性があります。サブ・ピクセルレベルの正確性があるので、シンボルの外観を弁別することはほとんど不可能です。

しかし、スタティックキャッシュでは、異なります。すべてのグリフは、所定の名目フォントサイズで事前にラスターライズされ、バイリニアまたはトリリニア (mip-maps) フィルタを使用してスケーリングされています。したがって、ダイナミックキャッシュとは異なり、このバランスは個々のグリフの高さでなく名目フォントサイズによって決まります。

ダイナミックとスタティックなキャッシュに関して、テクスチャーからベクトルレンダリングに移行する点は `GlyphCacheParams::` の `MaxRasterScale` の値を変更してコントロールできます。

ここで、`SetMaxRasterScale` の引数は、テクスチャスロットサイズの最大値 (ピクセル) の倍数を指定します。指定したポイントを超えるとベクトルへの切り替えが発生するようになります。

`MaxRasterScale` のデフォルト値は 1 です。1.25 という値は、画面上のグリフのサイズが、テクスチャに格納された名目グリフ サイズの 1.25 倍を超えない限り、Player はテクスチャを使用してテキストをレンダリングすることを意味します。デフォルトの名目サイズが 48 ピクセルであれば、画面上では 60 ピクセルを超えるグリフにベクトルレンダリングが使用されることになります。

`MaxRasterScale` を非常に大きな値に設定してしまうと、ベクトルレンダリングは使用されなくなることにご注意ください。`strip_font_shapes` オプションを指定して `gfxexport` ツールを実行した場合も、GFX ファイルのベクトル レンダリングが無効になります。この場合は、フォントグリフデータがファイル内に存在しなくなるので、ベクトル化は不可能ということになります。



## 6 パート 5:テキストのフィルタエフェクトと ActionScript 拡張

ダイナミックグリフキャッシュが有効な場合、Scaleform は、テキストに適用する「ぼかし」、「ドロップシャドウ」、「グロー」のフィルタをサポートします。テキストのフィルタエフェクトには、ダイナミックキャッシュが必要です。グリフパッカーやスタティックフォントテクスチャーを使う場合には、フィルタエフェクトはサポートされていません。現行の Player バージョンでは、フィルタは直接テキストフィールドに用いることは出来ませんが、ムービークリップには動きません。

Scaleform のテキストフィルタは、Flash と同様に機能しますが、異なる点も少なくありません。Adobe Flash では、テキストフィールド全体のラスターデータに対してフィルタが連続的に適用されます。つまり、「ぼかし」、「ドロップシャドウ」、「グロー」といったフィルタを次々に適用することができます。確かにこれは、非常に使いやすいのですが、計算上の負担は非常に大きくなります。Flash とは対照的に、Scaleform でサポートするフィルタは限定的ですが、非常に高速です。ダイナミックアダプティブグリフキャッシュと組み合わせると、フィルタは標準のテキストとほとんど変わらず高速で機能します。フィルタ サポートが限定されているにもかかわらず、Scaleform は非常に優れたソフトシャドウとグローのエフェクトを作成します。

### 6.1 フィルタの種類、使用可能なオプションと制約

Flash Studio のフィルタは、テキストフィールドに 1 つずつ追加できるようになっています。



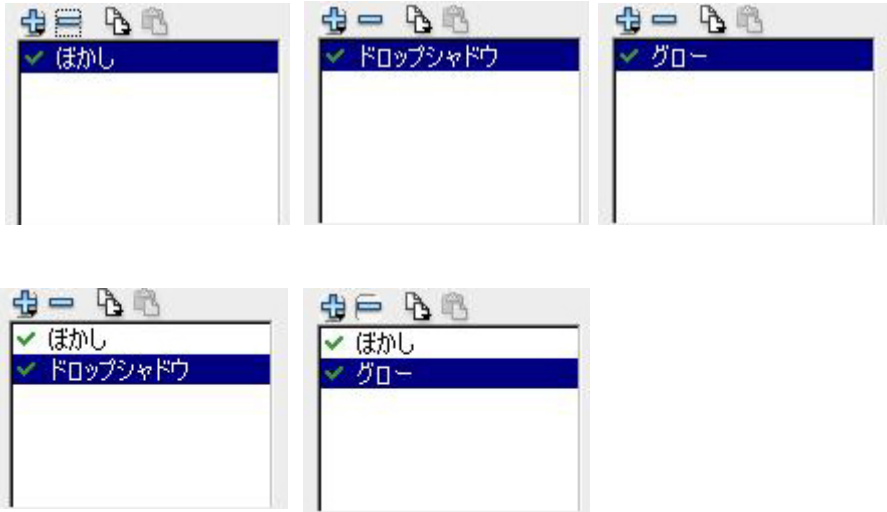
Flash のフィルタと Scaleform のフィルタの大きな違いは、Scaleform は、ぼかしを適用したグリフのビットマップコピーを作成してキャッシュに格納していますが、Flash の方は、出来上がったテキストフィールドにフィルタを適用します。

Scaleform では、ぼかしフィルタとドロップシャドウフィルタのみをサポートしています。グロー フィルタは実際にはドロップシャドウフィルタのサブセットです。Flash とは異なり、フィルタは連続的には適用されず、別の二つのレイヤーとして独立して機能します。Scaleform では、「ドロップシャドウ」と「グロー」のどちらか一方、つまりフィルタ リストの最後に表示されるものだけが考慮されます。その上で、オプションの「ぼかし」フィルタをテキスト自体に適用できます。アルゴリズムの概観は、次のようになります。

- フィルタリストの操作を繰り返します。
- 「グロー」または「ドロップシャドウ」のとき、シャドウフィルタパラメータを格納します。
- 「グロー」または「ドロップシャドウ」は、リストの前に「グロー」または「ドロップシャドウ」の何れかがあれば、それを上書きします。
- 「ぼかし」であれば、ぼかしフィルタパラメータを格納します。

- どの「ぼかし」も、リストの前にある「ぼかし」を上書きします。

したがって、有効なフィルタの組み合わせは以下のとおりになります。

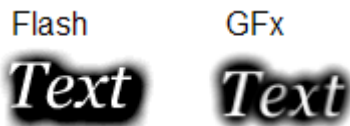


「ドロップシャドウ」フィルタと「グロー」フィルタを追加する場合は、最後に追加したものだけが考慮されます。

レンダリングも Flash とは異なります。まず、「ドロップシャドウ」と「グロー」のいずれかのレイヤーがレンダリングされます。次に、テキスト自体がシャドウの上にレンダリングされます。オプションで「ぼかし」フィルタがあれば、オリジナルのグリフに適用されます。「ドロップシャドウ」フィルタまたは「グロー」フィルタに [ノックアウト] または [オブジェクトを隠す] のフラグがセットされている場合は、テキスト (第 2 のレイヤー) がレンダリングされません。Scaleform の現行バージョンでは、[シャドウ (内側)] と [グロー (内側)] はサポートされていません。

このアルゴリズムの特性によって、Scaleform のフィルタには一定の制約があります。Blur X と Blur Y の最大値は 15.9 ピクセルに制限されています。シャドウまたはグローの強度も最大値 1590% に制限されています。

[ノックアウト] オプションの動作も異なります。Flash では、オリジナルのイメージがシャドウからそっくり除去され、シャドウオフセットが保存されます。Scaleform では「グリフごと」に実行され、オフセットは無視されます。また、シャドウまたはグローの範囲が大きい場合は、グリフイメージが重なり合って互いに「あいまい」になることがあります。









したがって、[ノックアウト] オプションは、基本的に「グロー」フィルタ (あるいは、「ドロップシャドウ」のオフセットがゼロの場合) の範囲が小さい場合にのみ使えることになります。

## 6.2 フィルタ品質

Flash では、単純なボックスフィルタを使用してイメージにぼかしを適用します。[画質]は、フィルタが実行される回数を制御し、生成されるイメージに重大な影響を及ぼします。例えば、[画質]が[低]のぼかしフィルタ 3x3 では、単に 3x3 のピクセル領域の平均値が計算されます。[画質]が[中]の場合は、同じフィルタが 2 回適用され、[高]ではこのフィルタが 3 回適用されます。つまり、[画質]は、フィルタが掛かった見えている範囲に大きく影響を与えます。アプローチの違いによって視覚的な結果は異なりますが、似たようなものです。実のところ、簡単な ボックスフィルタを一回通すだけでは、惨めな結果となります。

Flash とは違って、Scaleform はガウスぼかしフィルタと高度な再帰アルゴリズムを使用しています。処理速度がフィルタをかける範囲に影響されることはありません。Scaleform では、品質は [低] と [高] の二つだけで、見かけの結果にほとんど変わりはありません。フィルタパネルで [画質: 低] が設定されている場合にのみ、低品質が適用されます。それ以外の場合は、高品質のフィルタが使用されます。両者の違いは、高品質のフィルタでは小数値で範囲 (シグマ) を操作できますが、低品質のフィルタでは範囲設定が整数になることです。Scaleform では、ほとんどの場合に小数値のフィルタ範囲をグリフの適切なスケーリングでシミュレートしますが、テキスト フィールドに「アンチエイリアス (読みやすさ優先)」が適用されていると、低品質のグローでは、やや見劣りしてしまいます。一般的に推奨されるのは、読みやすさに関して最適化された小さなテキストに高品質のフィルタを使用することです。

	Low quality	Medium quality	High quality
Flash			
GFx			

おわかりのように、Flash では違いが明白ですが、Scaleform の方はほとんど分かりません。Scaleform では、「低品質」だけが違って見え、「中品質」と「高品質」では、同じ結果になります。

低品質のフィルタは高速ですが、グリフキャッシュシステムのために速度差は解消されてしまいます。とはいえ、低性能システムには、特に浮動小数点数を処理するハードウェアがないような場合は、低品質のフィルタを使われる方がよいでしょう。

高品質のフィルタでは浮動小数点演算を必要とし、再帰的な方法が使用されます。この詳細についてはこちらを参照してください：  
<http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Smoothin.html>

## 6.3 フィルタのアニメーション

Flash タイムラインでも、Scaleform ActionScript 拡張を使用しても、シャドウエフェクトをアニメーション化することができます。ただし、アニメーション操作による負担を認識しておく必要があります。範囲、強度、品質を変えると、Scaleform はグリフイメージを再生成し、これをキャッシュに格納します。したがって、キャッシュの更新頻度が高くなります。実際、上記のような値の変更は、アルファベットの数を増やしているようなものです。各バージョンをキャッシュに格納しておかねばなりません。ところが、色 (アルファを含む)、角度、距離を変更してもパフォーマンスには影響しません。シャドウまたはグローをフェードインおよびフェードアウトしたい場合は、半径や強度を変えるのではなく、色の透明度 (アルファ) をアニメートすることをお勧めします。

## 6.4 ActionScript からのフィルターの使用

Scaleform は AS2、AS3 の両方のダイナミック/入力テキストフィールドに標準のフィルタークラス (DropShadowFilter、BlurFilter、ColorMatrixFilter、BevelFilter など) をサポートします。詳細は Flash のマニュアルを参照してください。