

Autodesk® Scaleform®

字體和文字配置

本文件描述了 **Scaleform 4.2** 中使用的字體和文本渲染系統，詳細介紹了如何對素材資源和 **Scaleform C++ APIs** 進行國際化配置。

作者： Maxim Shemanarev, Michael Antonov

版本： 2.2

最後編輯： 2012年 6月21日

Copyright Notice

Autodesk® Scaleform® 4.2

© 2012 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, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Homestyler, Autodesk Intent, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSketch, AutoSnap, AutoTrack, Backburner, Backdraft, Beast, Beast (design/logo) Built with ObjectARX (design/logo), Burn, Buzzsaw, CAiCE, CFdesign, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Creative Bridge, Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design Web Format, Discreet, DWF, DWG, DWG (design/logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DWFx, DXF, Ecotect, Evolver, Exposure, Extending the Design Team, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, Freewheel, GDX Driver, Green Building Studio, Heads-up Design, Heidi, Homestyler, HumanIK, i-drop, ImageModeler, iMOUT, Incinerator, Inferno, Instructables, Instructables (stylized robot design/logo), Inventor, Inventor LT, Kynapse, Kynogon, LandXplorer, Lustre, MatchMover, Maya, Mechanical Desktop, MIMI, Moldflow, Moldflow Plastics Advisers, Moldflow Plastics Insight, Moondust, MotionBuilder, Movimento, MPA, MPA (design/logo), MPI (design/logo), MPX, MPX (design/logo), Mudbox, Multi-Master Editing, Navisworks, ObjectARX, ObjectDBX, Opticore, Pipeplus, Pixlr, Pixlr-o-matic, PolarSnap, Powered with Autodesk Technology, Productstream, ProMaterials, RasterDWG, RealDWG, Real-time Roto, Recognize, Render Queue, Retimer, Reveal, Revit, RiverCAD, Robot, Scaleform, Scaleform GFx, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Sparks, SteeringWheels, Stitcher, Stone, StormNET, Tinkerbox, ToolClip, Topobase, Toxik, TrustedDWG, T-Splines, U-Vis, ViewCube, Visual, Visual LISP, Vtour, WaterNetworks, 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 的聯繫方式：

檔案名	字體和文字配置概述
地址	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
網站	www.scaleform.com
電子郵件	info@scaleform.com
直接呼叫	(301) 446-3200
傳真	(301) 446-3199

目錄

1	引言：Scaleform 中的字體	1
1.1	Flash 文本和字體概要	1
1.1.1	文本域	2
1.1.2	字元嵌入	3
1.1.3	嵌入字體記憶體佔用	4
1.1.4	控制字體記憶體佔用	5
1.2	遊戲用戶介面的字體決策	5
2	第 1 部分：創建遊戲字體庫	7
2.1	字體符號	7
2.1.1	導出和導入字體符號	8
2.1.2	導出字體和字元嵌入	10
2.2	創建 gfxfontlib.swf 文件步驟	10
3	第 2 部分：選擇國際化方法	12
3.1	導入替代字體	12
3.1.1	國際文本的配置	14
3.1.2	Scaleform Player 中的國際化	15
3.1.3	設備字體仿真	16
3.1.4	创建字体库的分步指南	17
3.2	自定義資源生成	19
4	第 3 部分：設定字體資源	20
4.1	字體查找的順序	20
4.2	Gfx::FontMap	22
4.3	Gfx::FontLib	23
4.4	Gfx::FontProviderWin32	23
4.4.1	使用自動提示文本	24
4.4.2	設定自動提示	25
4.5	Gfx::FontProviderFT2	26

4.5.1	將 FreeType 字體映射入文件	27
4.5.2	將字體映射到記憶體	28
5	第 4 部分：配置字體渲染	30
5.1	配置字形緩存	30
5.2	利用動態字體緩存	32
5.3	使用字体压缩器 – gfxexport	34
5.4	預處理字體紋理 - gfxexport	34
5.5	設定字體字型填充程式	36
5.6	向量控制	37
6	第 5 部分：文本過濾效果和動作腳本擴展	40
6.1	篩檢程式類型，可用選項和限制	40
6.2	過濾品質	41
6.3	動態過濾	42
6.4	使用 ActionScript 中的过滤器	43

1 引言：Scaleform 中的字體

Scaleform 載有一個新的靈活的字體系統，它可以提供高品質的HTML格式的可轉換文本。使用這個新系統，可以從不同的字體來源選擇替代字體用於本地化，包括局部嵌入的文本、共用GFX/SWF 文件、作業系統的字體資料、或FreeType 2 字體庫。同時，字體渲染質量也得到顯著改善，開發者可以在動畫的性能、記憶體佔用空間和文本的可讀性之間折中取捨。

Scaleform 的很多字體特徵依賴於Flash® Studio固有的功能，包括能將字體的字元集嵌入到SWF文件，以及能在有系統字體的情況下使用系統字體。但是，Flash Studio最初是爲了開發個人文檔而設計，因此限制了字體的本地化能力。尤其是嵌入式字體或翻譯表不能在文檔間得到輕鬆共用。此特徵在記憶體的有效使用和遊戲資源的開發方面特別重要。此外，Flash依賴於作業系統來處理未嵌入到文檔中的國際字元的字型替代，但是遊戲控制面板無法進行此項操作，因爲在遊戲控制面板中沒有系統字體。

Scaleform分別利用 Gfx::Translator類別和Gfx::FontLib類別作爲所有可譯文本和動態字體映射的中央回叫信號，解決了這些問題。同時，它還提供額外的介面以便於控制字體緩存機制和替代本地化過程中的字體名稱，並可在適當時候使用作業系統和FreeType2字體查找。

爲了有效使用Scaleform字體系統，開發人員需要瞭解Flash Studio和Scaleform運行時間的字體特徵。通過瞭解Flash的字體特徵，開發人員可以開發出前後連貫的字體，從而可以有效渲染出期望質量的字體，並佔用最少的記憶體。通過瞭解Scaleform運行時間的字體特徵，開發人員可以爲某一特定平臺選擇一種盡可能在質量、性能和記憶體使用方面均最佳的配置。

本文件旨在指導那些不是非常熟悉Flash和Scaleform的開發人員瞭解遊戲用戶介面的字體安裝程式。引言的其餘部分安排如下：

- “Flash 文本和基本字體”一節介紹 Flash 文本和字體使用的基本知識。前兩部分描述了文本域和字元嵌入的知識，熟悉 Flash 的開發人員可以跳過這兩部分。
- “遊戲用戶介面的字體決策”一節介紹開發人員在創建一個 Flash 遊戲的用戶介面時將需要作出的與基本字體有關的決策。建議每一個開發人員都閱讀本章，因爲它描述了本文件其餘部分的結構特徵。

1.1 Flash 文本和字體概要

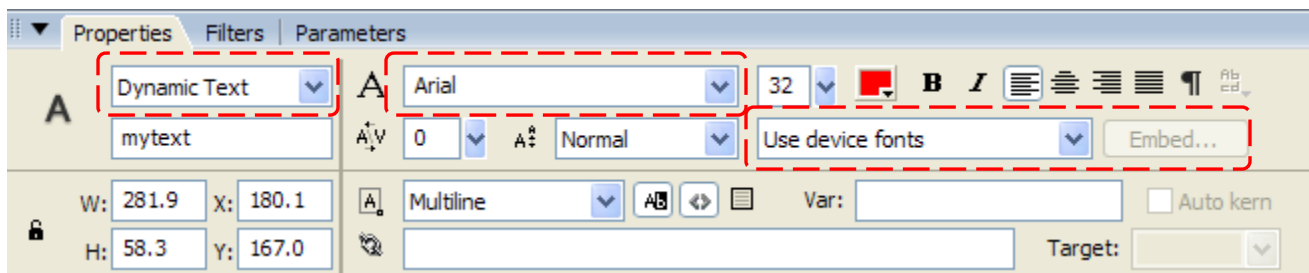
在 Flash Studio 中，開發人員首先用滑鼠在平臺上創建可視的文本域，然後輸入相應的文本。應用到文本域中的字體是從安裝在遊戲開發人員的系統裏的字體列表中按照名稱選擇的，並含有一個可用的嵌入選項，用於將內容放回不具備所需字體的系統中。

除了直接使用字體名稱以外，開發人員還可以在字體庫中創建字體符號，然後根據符號名稱間接使用他們。通過與字元嵌入相結合，字體符號能夠創建一個強大的概念框架，用於開發便攜而連貫的遊戲介面資源。本章的其餘部分簡要介紹了創建文本域和字元嵌入的基本知識。我們鼓勵開發人員參考Flash Studio文件，以便得到更詳細的解釋。

Flash字體符號系統使用方便且易於安裝，但它有若干限制，這使它很難共用字體以及進行國際化字體替代。而這些限制在Scaleform中得到瞭解決，詳情見第1部分---創建遊戲字體庫。

1.1.1 文本域

要在Flash中創建文本域，開發人員首先應選擇Text Tool，然後在平臺上繪出相應的矩形區域。文本的各種屬性，包括文本域類型、字體、大小和風格都設定在文本域屬性面板中，文本域屬性面板一般位於Flash Studio的底部。文本域屬性面板如下所示。



雖然文本域可以有很多選項，但我們要討論的關鍵屬性就是上圖中紅線圈中的部分。一個文本域最重要的選項是位於左上角的文本類型，文本類型可設置為下列值之一：

- 靜態文本
- 動態文本
- 導入文本

進行遊戲開發時，動態文本屬性是最常用的，因為它支援通過動作腳本修改內容，並支援通過使用用戶安裝的GfX::Translator 類別進行字體替代和國際化。靜態文本不具備這些特點，它在功能上非常類似于向量素材。導入文本可用來創建可編輯文本框。

上圖紅線圈中的其他兩處分別是位於屬性表最上面一行的字體名稱，以及字體渲染方法。選擇字體名稱時，可以使用(a)開發人員系統中安裝的任何一個字體的名稱，或(b)在動畫庫中創建的任何一個字體符號的名稱。在上面的例子中，“Arial”是選定的字體名稱。此外，字體風格還可以通過粗體和斜體切換按鈕進行設定。

對於國際化來說，字體渲染方法是最重要的設置，因為它控制著 SWF 文件的字元嵌入。在 Flash 8 中，它可以被設置為下列值之一：

- 使用設備字體

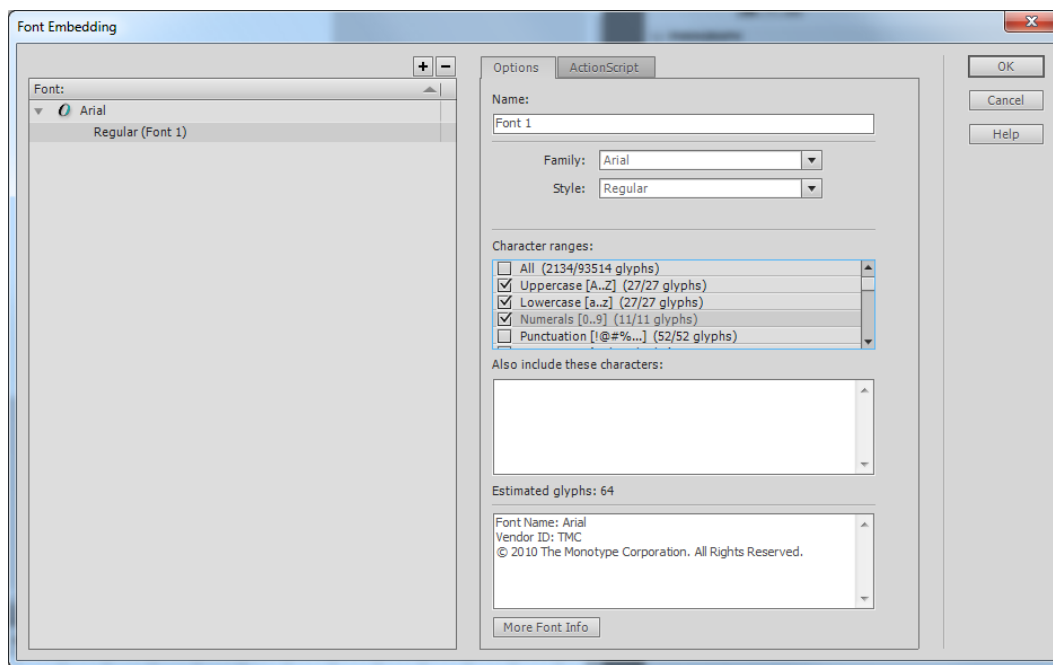
- 點陣圖文本（無反鋸齒）
- 用於製作動畫的反鋸齒
- 用於提高可讀性的反鋸齒

如果您選擇 “Use device fonts” （“使用設備字體” ），系統就會使用一個系統內置的字體渲染方法，這種方法的字體資料取自作業系統。使用設備字體的一個優勢是，用這個方法創建的SWF文件較小，而且在播放時佔用較少的系統記憶體。然而，如果當目標系統不具備所要求的字體時，Flash播放器將選擇另一種字體代替，這時就會出現一些問題。這個替代字體可能與原來的字體看起來不一樣，而且/或者不含有所有要求的字型。舉例來說，如果一個遊戲控制面板中沒有作業系統字體庫，它將顯示不了文本，那些不可顯示的字型在Scaleform中都會顯示為小方塊。

與使用系統字體不同的是，“用於製作動畫的反鋸齒”和“用於提高可讀性的反鋸齒”這兩個設置依靠嵌入式字體字元，分別將字體渲染為動態性能，並且具有高品質的優化效果。只要這兩個選項有一個被選中，嵌入按鈕就會變成可用狀態，此時用戶可以選擇嵌入字體的字元範圍。

1.1.2 字元嵌入

為了使嵌入式字體渲染工作,必須針對檔中至少一個 TextField(文字欄位)中的指定字型內嵌字元(假如匯出字體,就不需要任何文字欄位)。。如果不嵌入必要的字元，系統就會採用設備字體，這樣將帶來上文所述的所有限制（除非在 Scaleform 中使用了 Scaleform 字體庫，這一點稍後將作敘述）。如果要嵌入字體字元，用戶需要按下 Embed（嵌入）按鈕，此時將顯示如下所示的字元嵌入對話方塊。



在這個嵌入對話方塊中，用戶可以選擇所需的嵌入字元範圍，為文本域嵌入目前選定的字體和風格。使用相同字體風格的所有文本域將會共用相同的嵌入字元集，因此，通常只需設定嵌入其中一個文本域即可。

使用者應該明白，從嵌入的觀點看，字元的粗體和斜體屬性是單獨處理的(參閱上面螢幕截圖上的「Style」(樣式)下拉式列示方塊)。例如，如果同時使用普通的 Arial 和粗體的 Arial，則這兩種字體風格要分別嵌入，這樣就增加了文件的記憶體使用。不過，改變字體大小則不會不增加文件大小或記憶體佔用。關於字體和字元嵌入的更多詳情，請參閱 Flash Studio 文件中的“字體使用”章節。

嵌入字元是唯一真正便攜地使用字體的方式。當字體的字元被嵌入時，其向量表示即被保存到SWF/GFx文件，並且在後來使用時被載入到記憶體。因為字型資料是文件的一部分，所以無論系統正在使用哪一種安裝的字體，它都會永遠準確地渲染。在Scaleform中，嵌入的字體在遊戲控制面板（如Xbox 360，PS3，Wii）和臺式電腦（如Windows、Mac、Linux）上的效果同樣良好。使用嵌入式字體無可避免的副作用是，它會增加文件的大小和記憶體的使用。由於可能會大幅度地增加文件的大小，特別是使用亞洲語言的時候，開發人員將需要提前規劃遊戲的字體使用，並盡可能共用字體。

1.1.3 嵌入字體記憶體佔用

要瞭解字元嵌入在Scaleform中所佔用的記憶體，請參考下列表格。該表概述了當嵌入的字元數增加時，記憶體被佔用的情況。

嵌入的字元數	未被壓縮的 SWF 文件大小	Scaleform 運行記憶體佔用
1 個字元	1 KB	450 K
114 個字元-拉丁文 + 標點	12 KB	480 K
596 個字元-拉丁文和西裏爾字母	70 KB	630 K
7,583 個字元-拉丁文和日語	2,089 KB	3,500 K
18,437 個字元-拉丁文和繁體中文	5,131 KB	8,000 K

這個示例表是根據嵌入“Arial Unicode MS”字型創建的。正如上文所述，加入歐洲字元集佔用相對較少的記憶體，開發人員加入 500 個字元，大約增加 150K 的記憶體使用，這對於幾種不同字體風格在本地分配來說是足夠的。但是對於亞洲語言來說，由於它們字型數量龐大，它們佔用的記憶體也會大大增加。

1.1.4 控制字體記憶體佔用

因為嵌入大量的字元集會佔用數兆的記憶體，開發人員將需要提前計劃字體使用，以減少記憶體佔用量。以下幾種方法可以用來控制字體所佔用的記憶體：

1. 在開發用戶介面的素材資源之前，預先確定字體集內使用的字體和字體風格。粗體和斜體的字體風格在 **Flash** 中是作為單獨的字元集分別嵌入的，因此應將他們視為完全獨立的字體，並且只在必要時使用他們。不過，不同的字體大小不會佔用任何額外的記憶體空間，所以使用不同的字體大小不會造成記憶體增加。將來，我們將提供一個假的粗體和斜體的選項，避免粗體和斜體的版本需要儲存額外的字體字元。
2. 通過使用 **Scaleform** 字體庫和/或導入字體使文檔共用字體。在每一個單一的文件內嵌入相同的字元，可能會不合理地增加資源大小、記憶體佔用（如果同時載入稍後的文件）和載入時間。如有可能，最好是將嵌入的字體存儲入單獨的 **SWF/GFx** 文件，這樣的文件是共用文件，從而不需要重複佔用記憶體或重新載入。**Scaleform** 字體庫的使用將在本文件後面的部分加以討論。
3. 在亞洲字元集中只嵌入所用的字元。在亞洲本地化遊戲中，只嵌入在遊戲文本中使用的字元，儘量避免在語言中嵌入所有字元。經過翻譯，所有的遊戲字串可以通過掃描而生成所需的特殊字元集，這些特殊字元集隨後被嵌入到遊戲中。只要遊戲不要求通過 **IME** 導入任意動態文本，那麼對字元集加以限制就可以大大節省記憶體空間。
4. 考慮使用 **GfxEport** 字體壓縮(-fc 選項)。一般情況下,可使大小實際減少 **10%-30%**,而且不會明顯降低品質(參閱第 5.3 節)。

為得到最佳效果的國際化遊戲，用戶可以選擇結合所有這些技術，來限制所使用的字體，並通過 **Scaleform**字體庫共用這些字體。對於亞洲語言的遊戲，如果只嵌入所用的或**IME**所要求的字元，以及只嵌入一套完整的字體，都將可以大大節省記憶體空間。

1.2 遊戲用戶界面的字體決策

在**Scaleform**內創建遊戲的用戶介面時，開發人員將需要針對有關字體的使用、配置和國際化方法等問題作出一些決策。本文件細分了四個獨立的決策分類：

- 第1部分：創建遊戲字體庫——幫助您在創建字體庫時決定所要使用字體的數目和風格。
- 第2部分：選擇國際化方法——描述了創建國際化遊戲資源的不同方法。
- 第3部分：設定字體來源——描述字體查找的順序以及如何在 **Scaleform** 內配置不同的可用字體來源。
- 第4部分：設定字體渲染——描述 **Scaleform** 文本渲染的方式和其所擁有的不同配置選項。

創建遊戲字體庫是指選擇遊戲介面將使用的字體風格，這個步驟通常應該在開發遊戲用戶介面資源之前進行。在藝術字體方面，使用標準的字體庫是非常重要的，這樣可以確保在所有的用戶介面文件中為同一目的而前後一致地使用不同的字體。在技術方面，控制嵌入式字體的數目十分重要，因為這樣可以使他們符號應用程式的記憶體預算。

一旦決定了遊戲的字體，下一步就應該決定如何使他們國際化。第2部分論述了三種可能的國際化模式：

- **導入替代字體**，由導入程式和獨立的具體語言字體文件決定，這些文件被載入到播放器並通過 **Scaleform** 字體庫共用。
- **設備字體仿真**，與上述模式類似，但這個模式利用的是設備字體而不是導入的字體，並且需要適當的系統字體的支援（目前的遊戲控制面板上沒有）。
- **自定義素材資源生成**，此種模式利用 **Flash** 創建的翻譯系統為每個目標市場生成 **SWF/GFx** 文件的自定義版本。

開發人員可以選擇一個模式，也可以將這些模式適當組合起來，然後按照所選定的模式進行遊戲開發。

因為**Scaleform**中的字體可能來自多個字體來源，所以正確設定字體非常重要。為了幫助完成字體設定，第3部分“設定字體的來源”詳細描述了**Scaleform**中的字體查找程式，包括一些介紹如何設立**Scaleform**字體庫和字體提供程式的實例。字體提供程式是一個用戶可以配置的類別，用來支援系統字體和**FreeType 2**字體。它可以代替**Scaleform**字體庫進行存取字體，而不需嵌入字體。

本文件的第四部分“設定字體渲染”描述了在 **Scaleform** 中採用的字體渲染方法，並著重描述了三個可用的初始化字體紋理選擇，即在預處理時由 **gfxexport** 對紋理進行預渲染，將載入時間所產生的靜態紋理進行打包，以及使用按照要求更新的動態紋理緩存。本文件討論了每種方法的可用配置選項，以便開發人員可以為應用程式和目標平臺選擇正確的配置。

2 第 1 部分：創建遊戲字體庫

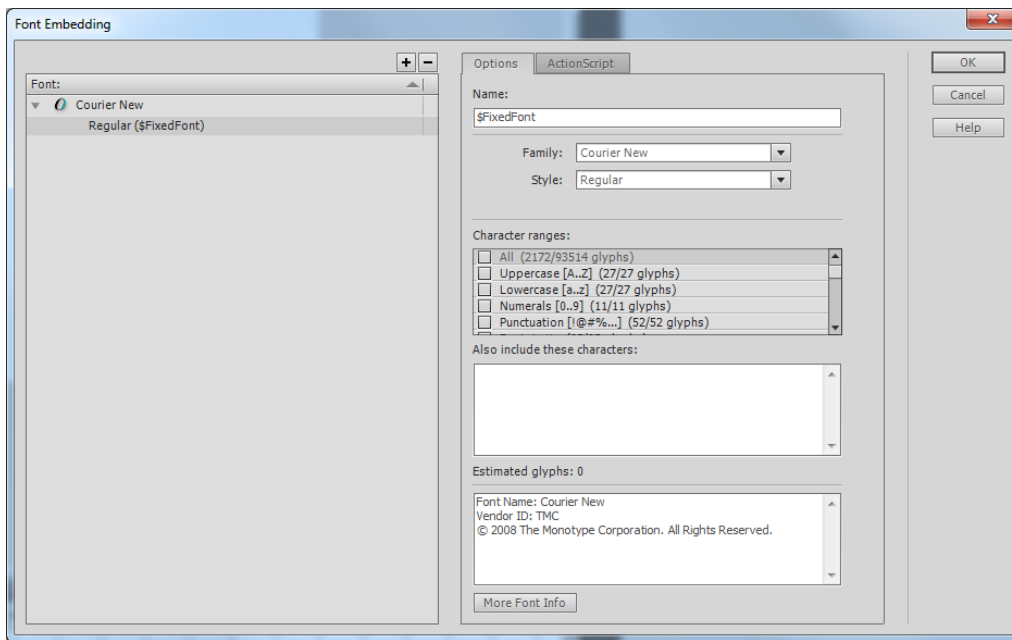
開發人員在開始進行遊戲用戶介面開發時，首先應確定一套將用於整個遊戲的潛在的字體樣式。例如，他們可以決定為所有的標題設立獨特的字體類型，而在遊戲其餘的文本中使用另一種類型。確定之後，如果沒有經過非常審慎的考慮，遊戲的用戶介面螢幕不應使用任何額外的字體類型。之所以這樣限制，主要有以下三個原因：

1. 通過使用相同的字體，開發人員可以確保呈現給用戶一致的遊戲內容。如果不同的遊戲畫面使用不同的字體，將會產生混亂的用戶介面，使用戶難以閱讀和理解。
2. 在國際化階段，開發軟體使用的字體往往需要被含有目的語言字元的字體所替代。如果有一套預先決定的固定的字體集，操作起來會更容易。
3. 如果有一套固定的字體集，則許多用戶介面的畫面可以共用這套字體集，這樣字體的資料就可以在記憶體中被共用，從而大大降低記憶體的使用，減少畫面載入時間。這是一項非常重要的技術措施，因為字體的資料需要佔用大量的存儲空間。

在 **Scaleform** 下,下一部分所述的導入字型替代方法,為某個遊戲選擇字體集的過程形式化為創建一個字體庫,該庫由一組檔構成,每種語言一個檔(例如:「**fonts_en.swf**」、「**fonts_kr.swf**」等)。。開發人員先在 **Flash** 文件字體庫中創建指定的字體符號，然後將它們導出到相應的 **SWF**，從而創建出“**fonts_en.swf**”字體庫文件。創建了這個字體庫文件後，庫內的字體符號可以被導入到其他 **Flash** 文件並在整個開發階段使用。下面的章節詳細描述了如何創建字體符號，以及如何利用字體符號創建一個遊戲字體庫。

2.1 字體符號

在引言中，我們介紹了如何將字體應用于文本域，以及如何嵌入這些字體的字元用於攜帶型播放。在 **Flash Studio**內，除了可以使用文本域屬性的系統字體外，開發人員還可以定義新的字體符號，具體做法為：把滑鼠移動到字體庫位置，點右鍵然後選擇“**New Font...**”項，此時顯示下列對話方塊：



用這種方式創建的字體符號被添加到 FLA 文件的字體庫中，並可以隨後應用於類似常用系統字體的文本域中。例如，如果您將您的遊戲字體命名為“\$FixedFont”，您就能可以選擇這個名字作為您文本域的有效字體。

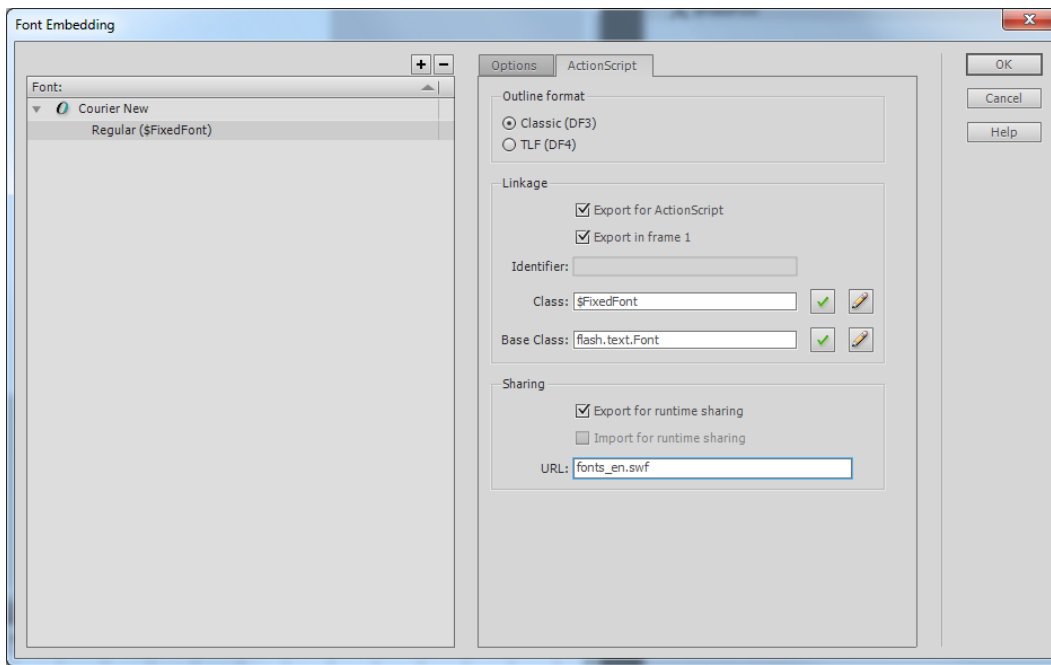
給字體庫加上字體名稱有利於發現以獨特方式使用的字體。當字體庫的字體在字體列表中顯示出來時，在每個字體旁邊都有一個星標，例如“\$ArialGame”。如果開發人員只使用字體庫中的字體名稱，這個方法將有助於確保沒有錯誤地引用用戶介面文件中的其他字體。此外，如果所有字體符號都以美元符號字元“\$”開頭來命名，它們將永遠顯示在文本域屬性中字體列表的上方，給開發工作提供幫助。

一些網站推薦使用下劃線字元“_”作為字體符號名稱的字首。雖然下劃線看起來更整潔，但是使用下劃線會導致文本域“Font rendering method”（“字體渲染方法”）這個選項框遭到破壞，因為Flash Studio會把任何一個以下劃線開頭的字體名稱看作是一個內置字體。為了避免出現這個問題，我們選擇使用美元符號字元。

2.1.1 導出和導入字體符號

與字體庫其他條目相似，其他文件可以通過複製其內容或利用導入/導出機制，使用位於FLA文件的字體符號。要複製一個字體庫的符號，先右擊FLA字體庫來源，然後選定“Copy”選項；做完這一步之後，切換到目標FLA文件字體庫，右擊這個目標FLA文件字體庫並選定“Paste”選項。此時即成功複製出一個字體符號，同時這個符號的所有資料內容也在第二個文件中複製出來。

如果要避免複製資料，可使用Flash的導出/導入機制。要匯出一個庫符號，開發者可以右擊該符號並選擇「Properties」（屬性），然後選擇「ActionScript」選項卡，此選項卡將會顯示下面的屬性工作表：



通過按「OK」(確定)按鈕,Flash 就會警告沒有類定義。忽略此警告資訊是安全的。

匯出一個符號時,會給其指定一個匯出識別碼,並給其標上「Export for runtime sharing」(針對運行時共用匯出)、「Export for ActionScript」(針對 ActionScript 匯出)和「Export in first frame」(在第一個幀中匯出)標誌。建議設定與符號名稱相同的導入識別字。URL 應給此 SWF 檔指定路徑,因為導入此符號時會用到該路徑;假如打算通過一個 Gfx::FontLib 物件替換此字體符號,就應將 URL 欄位設為您的預設字型庫。本文中我們將使用「fonts_en.swf」作為預設名稱,不過,也可以使用任何其他名稱。請注意,必須在 Gfx 中設置預設字型庫才能使用字型替代。

例如：

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

您可以在 fonconfig.txt 中為 GfxPlayer 設置預設字型庫。

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

第一個 fontlib 外觀將用作預設庫

一旦一個符號被導出,它就會在字體庫的鏈結欄被標上“Export: \$identifier”標籤。當對一個導入的字體庫符號進行複製/粘貼或拖放操作時,該符號將不會被複製,而會在目標文件創建一個導入鏈結。這意味著該目標文件會更小,並且當被載入到記憶體中時,它將拖出其從源 SWF 導入的資料。

在 Scaleform 內,即使資料是從多個文件導入的,導入的 SWF 的資料也只會被載入一次,這樣就大量節省了的系統記憶體。

字體符號的名稱將不會被作為 `TextField.htmlText` 字串的一部分或作為一個在 `TextFormat` 的字體名稱而返回；相反，系統字體的原始名稱，例如 “Arial” 將被返回。同樣地，除非字體符號能為一個導出的字體匹配導入名（不推薦），否則將不能通過動作腳本指定符號名稱。

2.1.2 導出字體和字元嵌入

CS4 Flash Studio 以上版本才允許指定要針對匯出的字型內嵌的字元。相反，要嵌入的字元集是通過系統地區設定來確定的。在 Windows 中，這一點由“控制面板\區域和語言選項\高級”內的“用於非 Unicode 程式的語言”設置控制。如果語言設置為“英語”，則每個字體將只有 243 個字型被導出。如果設置為朝鮮語，則將有 11920 個字型被導出。很明顯，這不利於遊戲開發，因此，創建了採用 `'gfontlib.swf'` 的整套方法。

幸運的是，從 Flash CS5 起，可以指定要匯出的字形 (Glyph)，因此不再需要 `'gfontlib.swf'`。開發者可以創建針對語言的字體庫(如 `'fonts_en.swf'`、`'fonts_jp.swf'` 等)，並直接指定應在這些字體庫中嵌入的字形。

不過，假如仍然在用 CS4 Flash Studio(或更舊版本)，就必須使用 `'gfontlib.swf'` 方法(該方法仍然受到支援，不過不建議在當前版本中使用)。有關此方法的詳細資訊，請參閱針對 GfX 4.0 和更低版本的此文檔的舊版本。

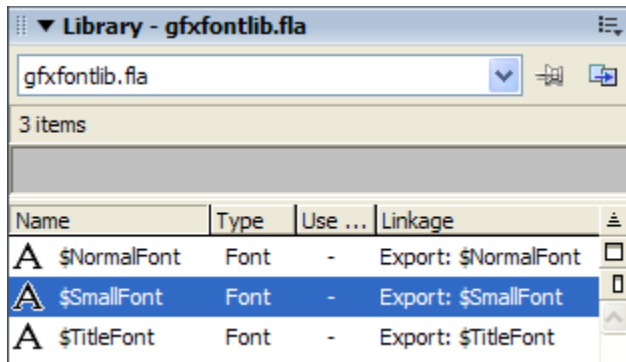
2.2 創建 `gfontlib.swf` 文件步驟

瞭解了字體符號後，我們現在就可以創建字體庫檔了。要重申的是，創建這些檔是為了提供按語言的源字體。下一節將介紹如何映射和替換庫字體的詳細資訊。

要創建針對某種語言的字體庫檔，請在 Flash Studio 中創建一個新 FLA，並用字體符號填充其庫。
。具體來說，你可以按照下列步驟進行操作。

1. 指定用於所有遊戲畫面的字體，並根據它們的目的給他們單獨命名。對於遊戲的標題和正常大小的字體來說，`'$TitleFont'` 和 `'$NormalFont'` 都是不錯的名稱。
2. 為字體庫創建一個新的 FLA 文件。
3. 為在第 1 步指定的每個字型創建一個新的字體符號。
4. 如果打算使用導入的字體替代，那麼您應該為每個字體配置鏈結屬性，以便使它在導出時攜帶與字體符號名稱相同的識別字。
5. 指定應嵌入的字形。
6. 將產生的檔另存為 `fonts_<lang>.swf`，其中 `'lang'` 表示語言(例如：`'en'` – 英語、`'ru'` – 俄語、`'jp'` – 日語、`'cn'` – 漢語、`'kr'` – 韓語等)。此名稱可以是任意名稱，但以後在 FontMap 配置中會用到此名稱 `</lang>`。

除非出於文檔編寫目的想使用其他文字欄位,否則不需要將任何文字欄位添加到一個字體庫階段中。
除了字體以外,不應將其他類型的符號添加到字體庫文件中。FLA完成後,字體庫顯示出類似於下圖的狀態。



字體庫完成後,其符號就可以在用戶介面螢幕的其餘部分使用了。爲了能夠在導入文件中使用字體庫的字體,您可以將字體符號下拉到目標動畫平臺中,或使用前面介紹的複製/粘貼技巧。

3 第 2 部分：選擇國際化方法

在用戶介面國際化的過程中，兩個主要條目，即文本域字串及字體，需要被替代。替代文本域字串是爲了實現語言翻譯，用目的語言中相應的片語取代開發文本。替代字體是爲了爲目的語言提供正確的字元集，因爲原本的開發字體可能不包含所有需要的字元。

在本文件中，我們介紹了三種不同的方法，您可以利用任何一種對藝術字體資源進行國際化。這三種方法分別是：

1. 導入的字體替代。
2. 設備字體仿真。
3. 自定義素材資源生成。

導入替代字體是推薦的字體國際化方法，這個方法利用原本由‘**fonts_en.swf**’字體庫提供的、用於取代字型符號的**GFX::FontLib** 和 **GFX::FontMap**物件。

設備字體仿真與導入替代字體類似，不同的一點是這個方法利用字體映射而非其導入的符號，來提供實際的字體。這個方法容易設置，但有若干限制。

導入替代字體和設備字體仿真這兩種方法都依賴於用戶創建的用於翻譯文本字串的**GFX::Translator**物件。

與上述兩種方法不同，自定義素材資源生成不利用字體映射或翻譯物件，而利用 **Flash Studio** 的特點，爲每個目的語言生成自定義的用戶介面資源檔案。對於含有少量靜態的用戶介面資源、記憶體極其有限的平臺來說，選擇這種做法比較適合。

3.1 導入替代字體

導入替代字體這個方法利用 **Flash** 的字體符號導入/導出機制來將文本域綁定到可替代字體。要使用此方法，開發者首先創建一個預設字型庫檔，名稱可以是，例如，第 1 部分所述的「**fonts_en.swf**」，然後在所有遊戲 UI 檔中使用從該檔匯出的字體符號。當在 **Adobe Flash** 播放器中被測試時，它們會被從

“**fonts_en.swf**”中導入，並用開發語言對其內容進行正確處理。然而，當在 **Scaleform** 中運行這些資源時，國際配置卻可以被載入，使翻譯和字體替代得以完成。

請注意,應該調用 `Loader::SetDefaultFontLibName (const char* filename)` C++ 方法,並將一個檔案名(僅僅是檔案名,不包含路徑!)作為一個 'filename' 參數傳遞。例如,假如預設開發語言為「韓語」,您需要進行如下調用：

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

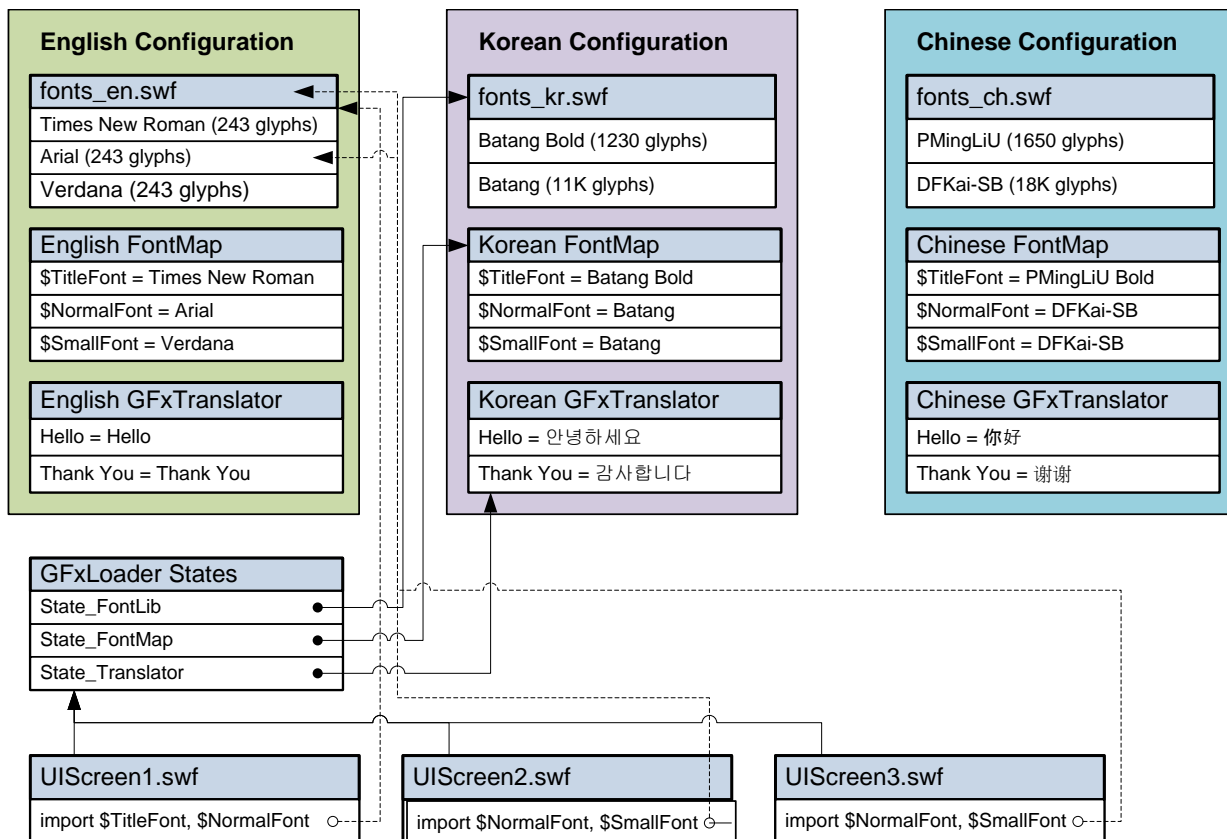
應在針對主要 SWF 檔調用 `CreateInstance` 之前調用此方法。

從根本上說，遊戲的國際配置由以下資料集組成：

1. 字串翻譯表，這個表將遊戲中的每一個短語映射到其翻譯版本。
2. 一套字體，它們為翻譯中使用的字元提供字型。
3. 字體替代表格，它將如 "\$NormalFont" 這樣的字體符號名稱映射到配置中可用的字體。

在 `Scaleform` 中，這些元件表現為 `GFx::Translator`、`GFx::FontLib` 和 `GFx::FontMap` 三個狀態，它們可以安裝在 `GFx::Loader` 上。當一個動畫文件被下載並含有調用指令 `GFx::Loader::CreateMovie` 時，裝入程式根據字體配置狀態中渲染文本的字體，將這個動畫文件與字體配置狀態綁定。要創建一個不同的字體

綁定，用戶需要在裝入程式中設置一個新的綁定狀態並再次調用 `CreateMovie` 指令。



上圖表示的是一個潛在的國際字體設置。在這種設置內，三個名為“UIScreen1.swf”的用戶介面文件通過“UIScreen3.swf”從“fonts_en.swf”字體庫導入一套字體。導入文件鏈結由虛線箭頭線表示。然而，Scaleform 不是從導入文件取用字體，相反，它是依靠設置於 Scaleform 裝入程式的 Gfx::FontLib 和 Gfx::FontMap states 來替代字體。在下述情況下此替換將會自動發生：(1) 導入檔與 Loader::SetDefaultFontLibName() 方法指定的那個導入檔相匹配，以及 (2) 載入 UI 屏之前在載入程式上設置了一個非空 Gfx::FontLib 狀態。

為了支援國際翻譯，上圖使用了三個配置：英語，韓語和中文。每個配置包括自己的字體文件和一個字體映射，如朝鮮語配置中含有“fonts_kr.swf”文件，而字體映射可以將導入的字型符號名稱映射到嵌入到字體文件中的實際字體。在我們的例子中，載入程式狀態設定為朝鮮語的配置，所以能創建朝鮮字體和翻譯表以供使用。然而，載入程式的狀態可以簡單地被設定為中文或英文的配置，並將用戶介面翻譯成這兩種語言。進行這樣的翻譯時不需要修改 UIScreen 文件。

3.1.1 國際文本的配置

當對含有導入替代字體的遊戲進行國際化時，開發人員首先需要創建國際化藝術資源和所需的語言配置。通常藝術資源和字體資源會以SWF格式儲存（調用前可轉換為Scaleform格式），而字體映射和翻譯表中的資料可以以具體遊戲的資料格式存儲，這種資料格式支援創建Gfx::FontMap和Gfx::Translator物件。在Scaleform樣例中，我們利用“fontconfig.txt”配置文件提供這方面的資料；然而，遊戲開發人員應該創建一個更先進的生產方案。

形式上，創建國際化配置的過程可細分為以下步驟：

1. 創建一個如第 1 部分所述的預設庫檔(如「fonts_en.swf」),並用它來創建您的遊戲素材。
2. 為每種語言創建一個翻譯字串表，這個翻譯字串表可以用來創建一個相應的 Gfx::Translator 物件。
3. 決定每種語言的字體映射；將這個映射以可以用來創建 Gfx::FontMap 的格式存儲。
4. 為每個目的語言創建針對語言的字體檔,如「fonts_kr.swf」。

國際配置創建後，開發人員可以將國際化的用戶介面螢幕載入到他們的遊戲中。選定目的語言後，採取下列步驟：

1. 為這種語言創建一個翻譯程式，並將它設置於 Gfx::Loader 上。從 Gfx::Translator 引出你自己的類別並覆蓋翻譯虛擬函數，即可進行翻譯。開發人員可以通過覆蓋這個類別，用他們選擇的任何格式顯示翻譯資料。
2. 創建一個 Gfx::FontMap 物件，並將它設置於 Gfx::Loader 上。通過調用 MapFont 功能，給它添加必要的字體映射。第 3 部分提供了一個利用 Gfx::FontMap 的例子。

3. 在載入器上設置遊戲所必需的與任何其它字體相關的狀態,例如,GFx::FontProvider 和/或 GFx::FontPackParams。如有必要,通過渲染執行緒上的 GlyphCacheConfig 來配置動態緩存。
4. 創建一個 GFx::FontLib 物件,並將它設置於 GFx::Loader 上。
5. 針對預設字型庫(swf 內容中使用的那個字體庫),調用 SetDefaultFontLibName(filename) 方法。

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

創建 GFx::FontLib 後,載入目的語言使用的字體源 SWF/GFx 文件,並通過調用指令 GFx::FontLib::AddFromFile 將其添加到字體庫。通過這一調用可以使嵌入在參數文件中的字體被作為字體導入或設備字體使用。

6. 通過調用 GFx::Loader::CreateMovie 載入用戶介面文件。字體映射、字體庫和翻譯狀態將自動適用於這個用戶介面。

3.1.2 Scaleform Player 中的國際化

爲了演示國際化,Scaleform Player支援使用國際化用戶參數文件,通常稱爲“fontconfig.txt”。當您拖動SWF/GFX 文件到播放器時,如有可用,這個‘fontconfig.txt’配置將自動從本地文件目錄載入;也可以通過命令行的/fc選項將其載入。載入國際化用戶參數後,用戶可以通過按Ctrl+N鍵切換它的語言配置。按F2鍵,當前的配置就會在播放器的HUD底部顯示出來。

國際用戶參數文件既可以保存爲 8 位元組的 ASCII,也可以保存爲 UTF-16 格式,並通過列出帶有各自屬性的字體配置而形成線性結構。下列的國際用戶參數文件可以用來說明前一節介紹的遊戲設置。

```
[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" = "謝謝"
```

從這個例子可以看出，有三個單獨的配置部分，每一個都以一個[FontConfig "name"]標題開始。當選中某個配置時，該配置的名稱就在Scaleform Player的HUD處顯示出來。在每一個配置內部，將使用適用於該配置的指令。可能使用的指令如下表。

配置語句	含義
fontlib "fontfile"	將指定的 SWF/GFx 字體文件載入 GFx::FontLib。第一個 fontlib 外觀將用作預設字型庫。
map "\$UIFont" = "PMingLiU"	將一個條目加入到GFx::FontMap，GFx::FontMap把在遊戲用戶介面螢幕上使用的一個字體映射到字體庫提供的一個目標字體。導入替代字體後，\$UIFont應是最初 default “fonts_en.swf” 導入到UI文件的字體符號的名稱。
tr "Hello" = "你好"	將源字串的一個翻譯添加到它在目的語言中的對應字串。開發人員應該使用更先進的解決辦法來存儲翻譯表。

用戶應該知道，配置文件的功能最初是作為一個例子，用於展示如何實現Scaleform的國際化。配置文件的結構今後可能會改變，將不另行通知；此外，隨著Scaleform國際化功能的演進，我們計劃將其過渡到XML格式。開發人員可以參照FontConfigParser.h/.cpp文件中的源代碼，來瞭解如何配置字體狀態。

Bin\Data\AS2\Samples\FontConfig and Bin\Data\AS3\Samples\FontConfig 目錄中有一個更為完整的Scaleform 導入替代字體的例子。該目錄包括 “sample.swf” 文件和一個 “fontconfig.txt” 文件，“sample.swf” 文件可以被放入 Scaleform Player，“fontconfig.txt” 文件可以將目錄翻譯成各種語言。我們鼓勵開發人員對該目錄的文件進行研究分析，以更好地理解國際化程式。

3.1.3 設備字體仿真

設備字體仿真與導入替代字體不同的是，它不利用導入的字體符號，而是依靠GFx::FontMap 替代系統字體名稱。在創建用戶介面的階段，開發人員選擇一套開發系統字體，例如“Arial”和“Verdana”，並直接將他們用作用戶介面所有資源的設備字體。當一個字體配置文件在Scaleform Player中運行時，它可以將系統字體的名稱映射到它們在目的語言中的對應字體。舉一個此類映射的例子，“Arial”在朝鮮語的用戶介面可以被映射到“Batang”。反過來，“Batang”字體又可以通過使用GFx::FontLib從“font_kr.swf”文件被載入。

如果開發人員計劃不依賴GFx::FontLib，而通過GFx::FontProviderWin32直接使用系統字體，或通過GFx::FontProviderFT2直接使用字體文件，設備字體仿真將是非常有幫助的。雖然這種做法看起來很容易設置，但由於設備字體的一些限制，導致使用設備字體不如利用導入的字體替代靈活：

- 設備字體在 Flash 不能被轉換。如果設備字體應用了任何旋轉的轉換，Adobe Flash Player 將無法顯示設備字體文本域。此外，播放器將不能準確調整設備字體文本，並會忽略應用於它的遮罩。

雖然所有這些功能在 **Scaleform** 中將正常運行，但由於 **Flash** 中支援有限，使得用戶介面的藝術資源的測試更為困難。

- 設備字體不能配置有可視的 “**Anti-alias for animation**” 設置。除非 **ActionScript** 中的 **TextField.antiAliasType** 屬性另有說明，否則 **Scaleform Player** 將一直爲了提高設備字體文本可讀性而使用反鋸齒。
- 當使用設備字體時，開發人員必須重新編輯所有用戶介面資源才能修改開發字體。當然，在 **Scaleform** 中可以使用字體映射，但當文件在 **Adobe Flash** 播放器中被測試時字體映射就不能使用了。相反，如果使用導入的字型替代，就可以通過簡單地編輯和重新生成預設字型庫檔 (「**fonts_en.swf**」) 來選擇預設開發字體。
- 在處理 **ActionScript** 內的 **TextFormat** 時，**Flash** 只能辨認導入的字體符號的名稱。這就是說，開發人員將需要使用直接的字體名稱，例如 “**Arial**”，而不應該使用符號的名稱，如 “**\$TitleFont**”。

設備字體仿真中，雖然在開發過程中 “**fonts_en.swf**” 文件作爲一種字體的符號貯存庫仍然是有用的，但不一定必須使用該文件。如果開發人員選擇使用 “**fonts_en.swf**” 文件，那麼就不應導出其字體符號。相反，可以將這些符號複製到目標用戶介面的文件中，在那裏他們可以作爲他們所代表的映射設備字體的別名。

3.1.4 创建字体库的分步指南

下面我們介紹創建一個簡單 **SWF** 的步驟，該 **SWF** 使用採用兩種語言的字體庫，並使用 **fontconfig.txt**。

1. 創建一個新的 **FLA**，並選擇 **ActionScript 2.0** 或 **3.0**。將其稱爲 ‘**main-app.fl**’。這將是一個使用字體庫的主要應用程式 **SWF**。
2. 創建另一個 **FLA**，選擇您在上一步所選擇的同一 **ActionScript**。將其稱爲 ‘**fonts_en.swf**’。這將是我們的預設字型庫。
3. 轉到 ‘**Library**’(庫)視窗，按一下滑鼠右鍵，選擇 ‘**New Font...**’(新建字體...)。將名稱指定爲 **\$Font1**，選擇 ‘**Family**’(家族)爲 ‘**Arial**’，‘**Style**’(樣式)爲 ‘**Regular**’。
4. 在 ‘**Character ranges**’(字元範圍)中，選擇要嵌入的字元。對於 **English**(英語)，選擇 ‘**Basic Latin**’ 就足夠了。
5. 切換到 ‘**ActionScript**’ 選項卡。複選 ‘**Export for ActionScript**’(針對 **ActionScript** 匯出)、‘**Export for frame 1**’(針對幀 1 匯出)、‘**Export for run-time sharing**’(針對運行時共用匯出)。在 ‘**URL**’ 中，輸入欄位類型 ‘**fonts_en.swf**’。按一下 ‘**OK**’(確定)按鈕。對於 **ActionScript 3.0**，**Flash Studio** 將給您顯示一條警告 ‘**A definition for this class could not be found...**’(找不到此類的定義...) – 忽略此警告，並再次按一下 ‘**OK**’(確定)。
6. 為另一個字體重複步驟 3 – 5，將其命名爲 ‘**\$Font2**’，並選擇不同的 ‘**Family**’(家族)和/或 ‘**Style**’(樣式)。在兩個字體中，‘**Family**’(家族)和 ‘**Style**’(樣式) 是相同的，然後 **Flash Studio** 可能會由於與第一個字體重複而丟棄第二個字體。比如說，我們這次選擇 ‘**Times New Roman**’。
7. 再創建一個 **FLA**，同樣的 **ActionScript** 設置。將其稱爲 ‘**fonts_kr.fl**’ 或 ‘**fonts_ru.fl**’，或者任何其他名稱，具體取決於您要使用的語言。假如是 **Korean**(韓語)，那麼這個名稱就是 ‘**fonts_kr.fl**’。

8. 轉到 'Library'(庫)視窗,按一下滑鼠右鍵,選擇 'New Font...' (新建字體...)。將名稱指定為 \$Font1,但選擇 'Family'(家族)為某種韓語字體(例如作為 「바탕」),「Style」(樣式)為 「Regular」。
9. 在 「Character ranges」(字元範圍)中,選擇要嵌入的字元。對於韓語,它可能是 'Korean Hangul (All)' (韓文(全部))。
10. 切換到 'ActionScript' 選項卡。複選 'Export for ActionScript'(針對 ActionScript 匯出)、'Export for frame 1'(針對幀 1 匯出)、'Export for run-time sharing'(針對運行時共用匯出)。在 'URL' 輸入欄位中,鍵入 'fonts_en.swf'。按一下 'OK'(確定)按鈕。對於 ActionScript 3.0,Flash Studio 將給您顯示一條警告 'A definition for this class could not be found...' (找不到此類的定義...) – 忽略此警告,並再次按一下 'OK'(確定)。
11. 為另一個字體重複步驟 8 – 10,將其命名為 '\$Font2',並選擇 'Family'(家族)為,比如說,「바탕체」。
12. 同時發佈 'fonts_en.swf' 和 'fonts_kr.swf'。
13. 現在就可以回到我們的 main-app.flx 上來了。但首先轉到預設字型庫檔(即 'fonts_en.swf'),轉到 Library(庫),在那裡選擇 '\$Font1' 和 '\$Font2',並將其複製到一個剪貼簿(Ctrl-C 或右擊 -> 'Copy'(複製))。
14. 切換到 'main-app.flx',轉到 Library(庫),粘貼字體符號(Ctrl-V 或右擊 -> 'Paste'(粘貼))。您就會看到您的字體,其中包含 'Import:' (導入:) 首碼。
15. 在 'main-app.flx' 的階段上創建一個文字欄位。使其成為 'Classic Text'(經典文本)、'Dynamic Text'(動態文本)。從下拉清單 'Family'(家族)中選擇 '\$Font1*'。鍵入 '\$TEXT1',作為該文字欄位的上下文(翻譯時,這將用作 ID)。
16. 再創建一個文字欄位,選擇 '\$Font2*',在其中鍵入 '\$TEXT2'。
17. 發佈 'main-app.swf',我們在這裡的工作就完成了。
18. 接下來,需要在與存儲所有 swf 相同的目錄中創建一個 fontconfig.txt。打開 'Notepad'(記事本)或支援 Unicode 或 UTF-8 的任何其他文字編輯器。鍵入以下代碼:

```
[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!"
```

重要提示:儘管我們使用了字體的韓語名稱(「바탕」和「바탕체」),但 Flash 可以在 SWF 中使用英語名稱(「Batang」和「BatangChe」)。這就是我們在 fontconfig.txt 中使用英語字體名稱的

原因。要確保在 SWF 中使用哪些字體名稱,您可以使用 `gfxexport -fntlst<swfname>` 命令並分析輸出 *.lst 檔</swfname>。

19. 保存該檔(不要忘記將其另存為 Unicode 或 UTF-8!)。大功告成!在 GfPlayer 中打開 'main-app.swf',您就會看到預設文本 'This is'(這是)和 'English!'(英語!)。使用 Ctrl-N 切換到 Korean(韓語),您就會發現如何使用 fonts_kr.swf 中的字體(由於我們僅在 fonts_kr.swf 中嵌入了韓語字形)用韓語替換英語文本。

3.2 自定義資源生成

顧名思義,自定義資源生成就是為每一個目的語言發行創建自定義的 SWF/GFx 文件。例如,如果一個遊戲使用的是 "UIScreen.fla" 文件,每一個遊戲發行就會手動生成一個 SWF 的不同版本,即朝鮮語的 "UIScreen.swf" 版本和英文的 "UIScreen.swf" 不同版本。這個文件的不同版本可以放置到不同的文件系統目錄,或者只放在目標市場而不作發行。

使用自定義資源的主要優點是,可以在每個文件中嵌入最小數目的字型。雖然一般我們不建議這種做法,但當記憶體極為有限(如供用戶介面使用的記憶體少於600K),並且藝術資源符合下列條件時,可以考慮使用自定義資源生成:

- 用戶介面資源檔案相對較小,並且其文本內容很簡單。
- 很少有多個用戶介面文件一起下載(如果它們是一起下載的,其他方式所提供的字體分享功能將非常有用)。
- 動態文本域更新有限,並且嵌入字元的使用數量很少。
- 用戶介面不需要亞洲語言 IME 支援。

如果開發人員選擇採用自定義資源生成一個遊戲名稱,建議閱讀Flash Studio文件中的“利用字串面板編輯多語種文本”主題,並考慮使用字串面板(按Ctrl + F11鍵)編輯國際化文本。開發人員需要將“Replace strings”設定為“manually using stage language”,以便使Flash替代文本可以在當前Scaleform版本中正常工作。

4 第 3 部分：設定字體資源

Flash的每個文本域都帶有一個字體名稱，這個字體名稱或直接儲存，或編碼於HTML標記內。當顯示文本域時，通過搜索以下字體資源可以獲得所用的渲染字體：

1. 本地嵌入字體。
2. 從單獨的 SWF / GfX 文件導入的字體符號。
3. 通過 GfX::FontLib 安裝的、通過字體名稱或導入替代字體搜索到的 SWF/GfX 文件。
4. 系統字體提供程式，如 GfX::FontProviderWin32。前提是用戶安裝了這樣的程式。

嵌入和導入字體的使用在本文件的開始部分已經做了描述。在大多數情況下，嵌入字體的運作與Flash相同，因此不需要自定義配置。在查找字體的目的方面，導入字體符號的運作與嵌入字體類似。

用於設定非嵌入字體查找的三種可安裝狀態分別是GfX::FontLib、GfX::FontMap和GfX::FontProvider。正如本文件前面介紹的那樣，GfX::FontLib和GfX::FontMap用於為導入替代字體或設備字體仿真查找SWF/GfX載入的字體。下面將詳細介紹如何使用系統字體提供程式。

系統字體提供程式與調用指令GfX::Loader::SetFontProvider一起安裝，它允許字體資料來自另一種非SWF文件源。字體提供程式只能與動態緩存一起使用，並且只有當字體資料不是嵌入在SWF或字體庫中的時候才被搜索。目前，有兩種字體提供程式含有Scaleform：

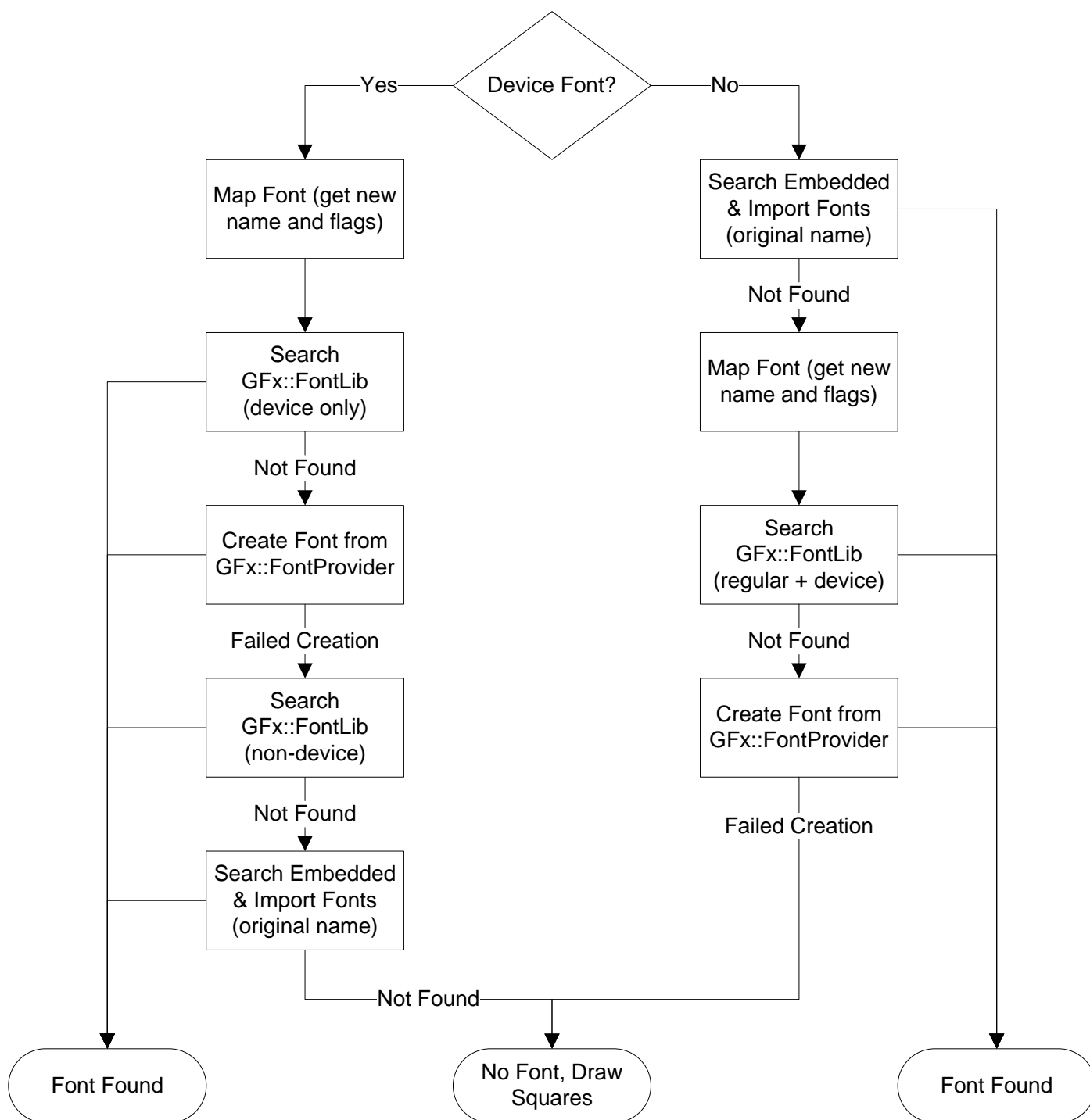
- GfX::FontProviderWin32 – 依靠 Win32 APIs 來獲取字體字型資料。
- GfX::FontProviderFT2 – 使用 David Turner 開發的 FreeType-2 字體庫來讀取和解讀獨立的字體文件。

在這部分我們詳細描述了字體查找的順序，並提供了代碼實例，用來說明如何配置不同的字體資源。

4.1 字體查找的順序

下一頁的流程圖解釋說明了在Scaleform中查找字體的順序。從中可以看出，通過設備字體標記可以修改查找，在文本域屬性中選定“Use device fonts”渲染方法就可以設置設備字體標記。

在 Adobe Flash 中，設置設備字體標記即可從系統中獲得相應的字體，且先於命名相同的嵌入字體（在這種情況下，後者被作為一種備選方案）。Scaleform 重復這個做法，但它是先搜索其安裝的字體庫，然後再搜索系統字體提供程式。這種設置並不會引起衝突，因為大多數控制面板攜帶型遊戲利用共用字體庫，而選擇使用系統字體提供程式的遊戲會使字體庫處於初始化（空）狀態。



從上圖可以看出，如果一個文本域不使用設備字體，就會首先搜索嵌入字體，反之，則最後搜索嵌入字體。此外，搜索嵌入式字體時總是以文本域中使用的原有字體名稱為基礎，而字體映射則可用於替代從 `Gfx::FontLib` 和 `Gfx::FontProvider` 中查找的字體名稱。

4.2 Gfx::FontMap

Gfx::FontMap是一個用來替代字體名稱的狀態。如果在國際化過程中開發字體不具備所需的字元，Gfx::FontMap允許使用替代的字體。導入替代字體和設備字體仿真都使用字體映射。當字體映射用於導入替代字體時，它將字體符號識別字轉換為字體名稱。當字體映射用於設備字體仿真時，它將原來的字體名稱映射到翻譯的字體中。

在第3部分，通過使用下列字體配置文件中的命令行，來創建字體映射：

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

在上面的例子中，使用映射命令將字體導入的識別字，例如“\$TitleFont”，映射到實際的字體名稱，實際的字體名稱又被嵌入在字體庫文件中。

使用下列的C++命令可以得到同等的字體映射設置。

```
#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);
```

在這種情況下，所有這三個字體都被映射到相同的字體名稱;特別是“\$NormalFont” “\$SmallFont”也有著相同的字體風格，這樣就節省了字體庫文件所佔用的記憶體。對MapFont設定第三個參數，就可以形成不同的字體風格，這樣就迫使映射使用特定的嵌入。如果不指定參數，MFF_Original值就會被使用，這意味著字體查找應保留文本域被指定的原有風格。字體大小可以由scaleFactor參數設置來改變。默認情況下，scaleFactor設置為1.0f。這個參數在字體顯示不清楚時需要適當增強效果時將被用到。

開發人員應該注意，Gfx::FontMap 是一個綁定狀態，這意味著利用 Gfx::FontMap 創建的動畫即使後來載入時被改變，也仍將使用這個狀態。如果設置不同的字型映射，Gfx::Loader::CreateMovie 調用指令就會給它返回一個相同檔案名的不同的 Gfx::MovieDef。這也適用於其他所有的字體配置狀態。

4.3 Gfx::FontLib

如第 3 部分所述,Gfx::FontLib 狀態代表一個可安裝的字體庫,該字體庫 (1) 為從預設「font_en.swf」中導入的字體提供替代選擇,並 (2) 提供用於設備字體模擬的字體。先在字體庫中搜索,然後在系統字體提供程式中搜索。下面的例子清楚的解釋了如何使用 Gfx::FontLib。

```
#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);
```

這個原理是像正常情況一樣創建並載入動畫,但不同的是,此處的動畫是用來作為字體存儲,而不是用於播放。它可以載入盡可能多的所需動畫。如果不同的動畫界定了相同的字體,則只使用第一個。

AddFontsFrom的第一個參數是動畫的清晰度,這個清晰度將作為字體的一個資源。第二個參數是插件標記,如果裝入程式要給記憶體中的動畫添加AddRef,則應設置此參數。只有當用戶不在載入動畫上使用智慧指標(例子中的M1和M2)的時候,才有必要使用此標記。如果插件標記處於非活躍狀態,那麼字體綁定資料,如導出或打包的紋理就可能被儘早釋放,使用字體時就不得不重新載入/再生這些綁定資料。

與字體映射類似,Gfx::FontLib也是一個綁定狀態,並為所創建的動畫提供參照,直至消失。

4.4 Gfx::FontProviderWin32

Gfx::FontProviderWin32字體提供程式可以在Win32 API中使用,並依靠 GetGlyphOutline函數來檢索向量資料。它的使用方法概述如下:

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

Gfx::FontProviderWin32的構造函數將Windows Display Context的處理程式作為一個參數。在大多數情況下,可以使用螢幕DC (::GetDC(0))。

如有需要,字體將被創建。

4.4.1 使用自動提示文本

一般來講，字體提示是一個非同尋常的難題。**Scaleform**提供了一個自動提示的機制，但是在中文，日文和韓文（中日韓）字元上這個自動提示的機制運行不佳。此外，大多數設計良好的中日韓字體中某些尺寸的字型含有柵格圖像，因為中日韓的提示遇到小尺寸的字型就會變得非常複雜。複製含有柵格圖像的向量字型是一個很好的而且切實可行的解決辦法。以**font APIs (Gfx::FontProviderWin32 and Gfx::FontProviderFT2)**為基礎的系統字體提供程式能產生向量和光柵形式的字型。字體提供程式有一個介面，用來控制自動提示。各種介面略有不同，但有相同的原則。有四個參數：

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

參數**VectorRange**和**RasterRange**控制使用自動提示的字元的範圍。有以下各值：

Font::DontHint – 不使用自動提示，
Font::HintCJK – 中文，日文和韓文字元使用自動提示，
Font::HintAll – 所有字元都使用自動提示。

RasterRange 優先於 **VectorRange**。參數 **maxVectorHintedSize** 和 **maxRasterHintedSize** 界定使用自動提示時圖元中的最大字體尺寸。下圖包括 **SimSun** 字體的例子，且含有不同的選項。

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ	vectorRange=GfxFont::DontHint rasterRange=GfxFont::DontHint
--	--

傷偽饑饉德僂僭僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇
僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ	vectorRange=GfxFont::HintAll rasterRange=GfxFont::DontHint
--	---

傷偽饑饉德僂僭僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇
僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ	vectorRange=GfxFont::HintAll rasterRange=GfxFont::HintCJK
--	--

傷偽饑饉德僂僭僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇
僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇僇

可以看出，自動向量提示對這個字型沒有多大用處，而使用柵格圖像則可以明顯地更改文本。然而，對於某些字體來說，向量提示是有意義的，例如“**Arial Unicode MS**”字體。

abcdefghijklmnopqrstuvwxyz
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
 僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞
 僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞僞

請注意，與Adobe Flash不同的是，Scaleform支援設備字體的任意仿射變換。旋轉和傾斜當然會使字型模糊，但任意仿射變換仍然相當適合動畫文本。

默認情況下，Win32和FreeType兩個提供程式都使用以下值：

```

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

```

設定字體提示的特別介面將在下面進行介紹。

4.4.2 設定自動提示

請注意，如果設置了SetHinting()之後再調用SetHintingAllFonts()函數，那麼這個具體字體的提示操作不會更改。確切的函數原型是：

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

或

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

請注意，如果在設置了提示 SetHinting() 後調用 SetHintingAllFonts() 函數，那麼這個具體字體的提示操作不會更改。確切的函數原型是：

```

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);

```

參數Name可以使用UTF-8 編碼。

在默認的情況下，`Gfx::FontProviderWin32` 在亞洲字元中使用非反鋸齒光柵。然而，在某些情況下，最好使用反鋸齒效果。`Windows API` 允許您對其進行控制，`Gfx::FontProviderWin32` 也可以實現這一功能。可通過如下設置實現反鋸齒效果：

```
void FontProviderWin32::SetRasterFormat(unsigned format, UByte* gamma=0);
```

格式參數可選擇下列類型之一：

```
GGO_BITMAP (默認),  
GGO_GRAY2_BITMAP,  
GGO_GRAY4_BITMAP,  
GGO_GRAY8_BITMAP
```

這些值將在 `<windows.h>` 進行定義，它們將包含在 `Gfx::FontProviderWin32.h` 中。

伽馬參數是一個無符號位元組的陣列，圖元映射範圍是 `0...255`。對於 `GGO_GRAY2_BITMAP` 來說，這是可選的，但是必須對 `GGO_GRAY4_BITMAP` 和 `GGO_GRAY8_BITMAP` 進行說明。

`GGO_GRAY2_BITMAP` 生成的圖元範圍是 `0...4`，`GGO_GRAY4_BITMAP` 為 `0...16`，`GGO_GRAY8_BITMAP` 為 `0...64`。因此，伽馬陣列必須分別包含 `5`、`17` 和 `65` 的值。第一個值必須是 `0`，最後一個必須是 `255`。其他的值必須是 `0` 到 `255` 之間的伽馬曲線上的數值。最簡單的情況是線性的，例如 `GGO_GRAY2_BITMAP`，線性伽馬值為：`0, 63, 127, 191, 255`。

4.5 Gfx::FontProviderFT2

這個字體提供程式使用 David Turner 開發的 `FreeType-2 library` 字體庫。它的使用方法與 `Gfx::FontProviderWin32` 類似，不同的是使用 `Gfx::FontProviderFT2` 需要將字體名稱和屬性映射到實際的字體文件。

首先，我們建議開發人員閱讀 `FreeType` 手冊，以便能恰當地配置和創建字體庫。開發人員需要決定使用靜態還是動態鏈結，以及適當的運行時間配置設置，例如字體驅動器、記憶體配置器、外部文件流等。

使用靜態鏈結的 `Windows MSVC` 編譯程式應該使用下列字體庫：

```
freetype<ver>.lib – 用於釋放多流 DLL 代碼生成，  
freetype<ver>_D.lib – 用於調試多流 DLL，  
freetype<ver>MT.lib – 用於釋放多流 DLL(靜態 CRT)，  
freetype<ver>MT_D.lib – 用於調試多流 DLL (靜態 CRT)。
```

其中 “<ver>” 指的是 `FreeType` 版本，例如，對於 `2.1.9` 版本，是 `219`。

此外，要確保 `freetype2/include` 和 `freetype2/objs` 這兩個目錄分別可以在額外的引用功能和字體庫兩個路徑上使用。

Gfx::FontProviderFT2物件的創建過程如下：

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

構造函數含有一個參數：

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

這是FreeType字體庫處理。如果這個值為零（預設值），供應程式將在內部初始化FreeType字體庫。當應用程式已經使用了FreeType字體庫，並欲在Scaleform和其他系統上共用這一處理時，能夠指定一個現有的初始化外部處理程式。注意，當使用外部處理程式的時候，開發人員應注意妥善釋放字體庫（調用指令：FT_Done_FreeType）。此外，應用程式必須保證處理的壽命要長於Gfx::Loader的壽命。使用外部處理程式也是因為有時需要配置FreeType字體庫（記憶體配置器等）的運行時間回調。請注意，此配置也可以被內部初始化。fontProvider->GetFT_Library()這個函數將使用的FT_Library 處理程式返回，並確保在實際使用字體之前供應程式不會調用FreeType字體庫。這樣就有機會在創建Gfx::FontProviderFT2後重新配置FreeType字體庫運行時間。

4.5.1 將 FreeType 字體映射入文件

與Win32 API不同的是，FreeType字體不會將映射字體的名稱（和屬性）提供給實際的字體文件。因此，這個映射必須從外部提供。Gfx::FontProviderFT2 有一個簡單的映射機制，可以將字體映射到文件和記憶體。

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

參數“fontName”指定字體字型的名稱，例如，“Times New Roman”。參數“filename”指定字體文件的路徑，例如，“C:\WINDOWS\Fonts\times.ttf”。“fontFlags”的值可能為“Font::FF_Bold”，“Font::FF_Italic”，或“Font::FF_BoldItalic”。“Font::FF_BoldItalic”這個值實際上等於“Font::FF_Bold | Font::FF_Italic”。“faceIndex”是被傳遞到 FT_New_Face 的。在大多數情況下，這個參數是 0，但並非總是如此，這取決於字體文件的類型和內容。請注意，這個函數實際上並不打開字體文件;而只是構成映射表。只有真正被要求時字體文件才被打開。與 Win32 字體提供程式不同的是，FreeType 字體只是使用函數參數來配置自動提示。

4.5.2 將字體映射到記憶體

FreeType允許將字體映射到記憶體。例如，如果一個單獨的字體文件包含了很多字型，或如果字體文件已經被應用程式載入，這個配置可能是有意義的。

```
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);
```

下面是一個將字體映射到記憶體的簡單例子。

```
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);
}
```

在這個例子中，記憶體沒有被釋放（而且最終會導致記憶體泄漏）。這是因為映射表只在字體資料保持一個裸指標常數。所以開發人員應該負責妥善釋放記憶體。應用程式必須保證這個記憶體塊的壽命要長於Gfx::Loader的壽命。它旨在為字體映射機制提供盡可能多的自由。例如，應用程式可能已經使用了含有預先載入記憶體的字體的FreeType，其中字體的分配和銷毀都由Gfx 在外部處理。

在Windows中使用FreeType字體映射如下例所示。

```
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);

```

請注意，這只是一個例子，在一個真正的應用程式中指定絕對硬編碼文件路徑的做法是不可取的。通常它應該來自一個配置文件，或通過自動掃描字體的字型得到。指定明確的字體名稱看起來好像小題大做，因為字體通常包含這些名字，但這樣做避免了會佔用很大記憶體空間的文件解析操作。函數 **MapFontToFile()** 只存儲這些資料，但並不打開文件，除非它被要求打開文件。這樣，可以指定大量的映射字體，而只有少數被真正使用。在這種情況下不用執行任何額外的文件操作。

5 第 4 部分：配置字體渲染

Scaleform可以用兩種方式渲染文本字元。第一種，將字體字型光柵化為紋理，然後使用成批的三角形紋理（每字元兩個）對它進行渲染。或者還可以將字體字型鑲嵌成三角形網格，並將其渲染為向量圖形。

對於程式中的大多數文本來說，操作時應始終使用有紋理的三角形。字型的紋理通過動態或靜態的緩存生成。動態字型緩存更加靈活，並能產生更高質量的圖像，而靜態緩存的優勢是可以進行脫機計算。

開發人員可以根據目標平臺和標題的需要，從下列專案中選擇字體渲染選項：

1. 使用動態緩存，進行快速動畫運行及高品質的導出。動態緩存會使用一個固定數額的紋理記憶體，並會減少載入時間，這是因為不是所有字型都需要被光柵化和/或載入。
2. 使用靜態緩存，從磁片載入預先生成的霧化貼圖紋理。開發人員可以使用 **gfxexport** 工具來提前生成打包的字型紋理，剔除字體的向量資料，這樣它就不需要被載入。
3. 使用靜態緩存，在載入時間使用 **GFx::FontPackParams** 程式自動生成紋理。這點類似以前的方法，不同的是紋理不需要從磁片載入。
4. 使用向量圖形直接渲染文本字型。

建議在高端系統，如PC、Xbox 360、PS3中，以及很多情況下Nintendo Wii中使用動態緩存。動態緩存能確保某一解析度使用最少的載入時間和最高質量的字體渲染。如果開發人員計劃通過 **GFx::FontProviderWin32**或**GFx::FontProviderFT2**程式使用系統或外部字體支援，也需要使用動態緩存。

對於CPU和記憶體有限的平臺，如PSP和PS2來說，使用靜態緩存並由**gfxexport**工具導出紋理是一個很好的選擇，因為在CPU和記憶體有限的平臺中，對字型運行時光柵化可能會佔用過大的存儲空間。如果開發人員需要在執行渲染時避免動態紋理更新，也可以使用靜態緩存。然而，由於我們優化了向量資料存儲佔用，即使在較低等級的作業系統中，動態緩存也可能成為一個好的選擇。我們建議開發人員對他們的遊戲資料進行實驗，以確定最佳的解決辦法。

雖然**Scaleform**可以被這樣配置，向量圖形很少單獨用於文本渲染。相反，字型鑲嵌通常只用於大型字型，而小型文本通常結合字型鑲嵌和以紋理為基礎的方法，以達到高效率的渲染。開發人員可以參考“向量控制”章節以獲取更多關於如何控制或禁用此選項的詳細介紹。

5.1 配置字形緩存

Scaleform 中的字體渲染是通過 `Render::GlyphCacheConfig` 或 `Gfx::FontPackParams` 狀態物件配置的。預設情況下,字形緩存是由渲染執行緒上的 `Renderer2D` 物件自動創建的,並為動態字形進行了初始化。字型填充只用於靜態紋理初始化;其參數默認為零。在這樣的設置狀態下,所有創建的動畫除非來自被預處理為靜態紋理的 **Scaleform** 文件,否則都會自動使用動態字體緩存。

GFx 4.0 升級說明

Scaleform 3.x 和 4.0 版本之間的動態字形緩存配置變化很大。儘管 GFx 3.3 依賴于 `GFxLoader` 維護並在主執行緒上配置的 `GFxFontCacheManager` 物件,但 GFx 4.0 還是用 `Renderer2D` 維護且在渲染執行緒上配置的 `GlyphCacheConfig` 介面所取代。有關在 Scaleform GFx 4.0 中配置緩存的更多詳情,請參閱第 5.2 節。

當從一個紋理渲染光柵字形時,始終使用字形緩存系統。從程式內部來說,緩存管理器負責維護文本批次頂點陣列,當靜態和動態紋理緩存同時被使用時就會產生頂點陣列。當啟用動態緩存時,緩存管理器負責分配緩存紋理,使用光柵字元對它們進行更新,並使紋理與批次頂點資料保持同步。當只有靜態緩存被使用時,緩存管理仍需要保持文本頂點陣列,但它並不需要分配或更新動態紋理。

靜態緩存呈現出預先光柵化點陣圖紋理,含有密集的填充字型。靜態紋理的光柵化和填充可以在載入時間通過 `Gfx::FontPackParams` 進行,也可以脫機時用 `'gfxexport'` 工具進行。在這兩種情況下,字體都有靜態的紋理。嵌入字體的字型有或沒有一套靜態的紋理,依字體載入方式而定。如果字體有靜態的紋理,他們將始終被用來渲染字體的字型;否則,就會啟動動態緩存以渲染字體的字型。

渲染方法除了要根據使用的字體紋理類型而定以外,還要根據目標字型的圖元大小而定。更正式地說,要使用以下邏輯進行:

```
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;
    }

```

正如上文所述，如果紋理緩存不可用，或者如果字型過大不能夠被紋理渲染，就使用向量渲染。緩存方法的選擇要根據字體填充字型紋理的可用性而定。以下各節將詳細介紹如何設置這兩種方法。

5.2 利用動態字體緩存

當字體和字元集被限制為例如基本拉丁語、希臘語、斯拉夫語等時，靜態緩存是很有效的。然而，對於大多數亞洲語言來說，靜態緩存可能會消耗太多的系統和視頻記憶體，從而導致載入緩慢。此外，如果使用太多不同的字體也可能發生這種情況；換句話說，也就是當嵌入式字型總數過大（比如10000個或更多）的時候。在這些情況下，應該使用動態緩存機制。此外，動態緩存可以提供更多功能，如優化可讀性，大大提高字體的質量。預設情況下，動態緩存已啟用，並分配其緩衝區；要禁用動態緩存，您可以調用：

```

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

```

在上面調用中，`renderer` 為一個 `Render::Renderer2D` 物件，並在渲染執行緒上進行維護。此調用將把字形緩存使用的動態紋理數量設置為零，從而有效地將其禁用。要避免預設臨時緩衝區分配，應在初始化渲染 `HAL` 之前執行此調用。

當渲染文本時，動態緩存將字型光柵化，並根據要求更新各個紋理。它使用一個簡單的 **LRU**（近來最少使用的）緩存方案，但卻含有一個智慧適應字型即時填充紋理。您可以按照如下方式設定紋理參數：

```

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);

```

上述值是默認使用的。

`TextureWidth`, `TextureHeight` - 緩存紋理的大小。這兩個值在 2 幕運算時都四捨五入。

`MaxNumTextures` - 用於緩存的紋理的最大數目。

MaxSlotHeight - 字型的最大高度。圖元字型的實際高度不能超過此值。更大的字型被渲染為向量圖行。


SlotPadding - 用來防止字型削波和重疊的盈餘值。在大多數實際情況下第二個值就很適用。

TexUpdWidth, TexUpdWidth - 用來更新紋理的圖像大小，在實際情況下通常大小為 256x512 (128K 的系統記憶體)的圖像就很合適。也可以將大小降至 256x256 或甚至 128x128，但如果這樣做的話，需要更頻繁地進行紋理更新操作。

動態字形緩存 用來壓縮字體輪廓，動態運行，採用“最近使用最小”緩存策略。要瞭解字體字形緩存如何工作,請看一個非常簡單的記憶體分配器,它可以在一個給定的 1MB 的空間中僅分配 4 KB 區塊。分配器只能對這些 4KB 存儲塊進行分配或刪除。顯然，分配器同時最多可以分配 256 個資料塊。但是在大多數情況下，請求的資料塊遠遠小於 4K。可能為 16 位元組、100 位元組、256 位元組等，但是不能超過 4K 位元組。在這種情況下，就需要設計一種機制來處理 4K 空間裏的小資料存儲塊以獲得 1MB 存儲空間中獲得更大的存儲容量。但是支援的最小存儲單元是相同的 – 同時可分配 256 個存儲區域。動態字形緩存中有一個類似的運行機制，但為二維的文本空間。也就是說，動態字形緩存 可以存儲 $\text{MaxSlotHeight} + 2 * \text{SlotPadding}$ 個單元，但是對字體壓縮度更高，特別是在小的存儲空間。平均情況下這可以增加動態字體緩存容量 2-5 倍（壓縮字體輪廓，增加存儲空間需要 10 秒時間）。通常，60-80% 紋理空間被用來有效載荷，不依賴於字體大小。

總緩存能力（即，同時儲存在緩存記憶體的若干不同字型的最大數量）要根據平均字型大小而定。對於典型的遊戲用戶介面，我們粗略估計上述參數緩存能力在 500 至 2000 個不同字型之間。在典型的情況下，約 70-75% 的介面是有效運用的。緩存字型的最大數目必須足夠多，以便能夠處理任何一個單行文本域的可見部分。在這種情況下，如果一個單行文本域可見部分的不同字型的數量超過緩存記憶體的能力，其餘的字型就被渲染為向量圖行。

前面提到，動態緩存允許額外的能力並支援“提高可讀性的反鋸齒”和“提高動畫效果的反鋸齒”選項。這些選項是文本域的屬性，Flash 設計者可以在文本對話方塊面板中選擇它們。優化了可讀性的文本看起來更為銳化和更具可讀性。雖然它也可以被製成動畫，但這個動畫耗費較大，因為它會造成更頻密的紋理更新活動。此外，字型是自動添加圖元網格以減少圖元自動擬合運作的模糊強度的，這就意味著文本行也被添加了圖元。在動畫時，尤其是縮放時，視覺上有一個抖動的效果。下圖顯示了這些選項之間的差別。

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

當使用“提高可讀性的反鋸齒”選項時，字型光柵器執行自動擬合程式（也稱為自動提示）。Scaleform 不使用來自Flash文件或任何其他字體來源的字型提示。它渲染文本所需要的就是字型概述。它提供字體

源以外的外觀相同的文本。**Scaleform**自動提示程需要字體的某些資料，即平頂的拉丁字母的高度。必須準確管理超調量字型(例如**O**、**G**、**C**、**Q**、**o**、**g**、**e**等)。通常情況下，字體不提供這些資料，因此，必須在某種程度上將這些資料推導出來。為此**Scaleform**使用平頂的拉丁字母，他們是：

- 大寫字母有：**H, E, F, T, U, V, W, X, Z**，
- 小寫字母有：**z, x, v, w, y**。

爲了使用自動適應程式，字體必須包含至少一個上述的大寫字母和至少一個上述小寫字母。如果字體不包含上述字母，**Scaleform**將產生一個日誌警告：

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

“警告：字體'Arial':沒有提示字元('HEFTUVWXZ' and 'zxvwy'中的任何一個)。自動提示爲禁用狀態。”

如果顯示了這個警告，**Flash**設計者應該立即嵌入這些字母。通常情況下，只要在“嵌入字元”對話方塊添加“**Zz**”或嵌入“**Basic Latin**”就足夠了。

5.3 使用字体压缩器 – gfxexport

命令列 **gfxexport** 工具的目的是預處理 **SWF** 檔,生成分發並載入到遊戲中的 **GFX** 檔。預處理過程中,該工具可將圖像從 **SWF** 檔中剝離,並將它們提取到外部檔中。可以用許多有用的格式存儲外部檔,例如,**DDS** 和 **TGA**。選定 **-fc** 選項時,**gfxexport** 也將壓縮字體向量資料。這是失真壓縮,因此您可能需要用參數進行試驗,以便於找到記憶體消耗與字體品質之間的最佳折衷方案。

命令行选项	行为
-fc	啟用字體壓縮器。
-fcl <size>	設置名義字形大小。小的名義大小將會導致較小的資料大小,但字形較不精確。預設值為 256。多數情況下,名義大小 256 可節約大約 25% 的內存(與一般在 Flash 中使用的 1024 的名義大小相比),而且品質不會明顯降低。不過,對於確實大的字形,您可能需要增加名義大小。
-fcm	設置名義字形大小。小的名義大小將會導致較小的資料大小,但字形較不精確。預設值為 256。多數情況下,名義大小 256 可節約大約 25% 的內存(與一般在 Flash 中使用的 1024 的名義大小相比),而且品質不會明顯降低。不過,對於確實大的字形,您可能需要增加名義大小。

5.4 預處理字體紋理 - gfxexport

當**-fonts**這個選項被指定時，**gfxexport**也可以光柵化和導出填充字體紋理，並將他們保存在用戶指定的一個格式裏。當在**Scaleform Player**中載入**Scaleform**文件時，外部的紋理將被自動載入，並在文本渲染時被用作一個靜態緩存。

我們不建議將匯出的紋理用於典型 **GFx** 應用中。不過,這可能對低端移動平臺有益,以及在某些特殊情況下有益。

利用**gfxexport**來生成字體紋理具有以下優點：

- 保存外部的紋理，這樣即可去除字型的向量資料，節省了記憶體。相反，字型資料被載入的靜態紋理所取代。
- 載入紋理文件可能比在載入時間僅在 **CPU** 系統生成紋理文件的速度快。。
- 紋理文件可以轉化為緊湊的具體遊戲格式，並可以通過覆蓋 **GFx::ImageCreator** 而被載入。在您執行直接支援這些格式的 **Render::Renderer** 時，會發現這一點很有幫助。

使用預先生成的紋理確實也有缺點，如文本渲染質量較低，而且與使用動態緩存相比，載入時間可能更長。靜態文本使用霧化貼圖，導致不能提示，這意味著它不支援“提高可讀性的反鋸齒”的設置。

下面的命令行將把**'test.swf'**文件預先處理為**'test.gfx'**文件，為所有的嵌入式字體創建額外的紋理文件。

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

選項**-strip_font_shapes**將把嵌入字體的向量資料從產生的**Scaleform**文件中去除。雖然這樣做會節省記憶體，卻使這些資料不可能被調回用於大型字元的向量渲染，使字體紋理字型超過標稱大小時質量下降。

下表列出了**gfxexport**與字體相關的選項。這些選項用於控制紋理的大小、字型的標稱大小和目標文件格式。大部分選項與字型填充參數相符，因為**gfxexport** 在生成字體紋理時要使用字型填充程式，字型填充參數在下一節中將進行描述。

命令行選項	操作
-fonts	導出字體紋理。如果未指定，將不會產生字體紋理（可以在動態緩存或載入時間填充時進行）。
-fns <size>	紋理字型圖元的標稱大小；如果沒有指定，則預設值為 48。如果紋理字型為最大尺寸，採用標稱大小。使用 tri-linear mip-map 濾器對較小的字元進行運行時間渲染。
-fpp <n>	在單個字型圖像周圍留有的圖元空間，預設值為 3。
-fts <WxH>	字型被填充的紋理的尺寸。默認大小為 256x256。在指定方形紋理時，只可指定一個大小，例如：'-fts 128'指的是 128x128'。'-fts 512x128' 指定的是矩形的紋理。

命令行選項	操作
-fs	為每個字體添加單獨的紋理。默認情況下，字體共用紋理。
-strip_font_shapes	不要將字型形狀資料寫入生成的 Scaleform 文件。
-fi <format>	當<format>為 TGA8 (灰度化)， TGA24 (灰度化)， TGA32 或 DDS8 時，為每個字體紋理指定導出格式。默認情況下，如果圖像格式(-i option)為 TGA ，那麼 TGA8 就被用於字體的紋理，否則就用 DDS A8 。

警告:如果您計畫在您的遊戲中僅使用打包的靜態字體或字形打包器,您應如 5.2 節所述禁用動態字形緩存紋理。如果不執行此操作,就會仍然分配預設紋理,而且預設紋理仍然保持未使用狀態。

5.5 設定字體字型填充程式

載入文件時，字體字型填充程式將嵌入字型光柵化並填充。此外，字體也可能被**GFxExport**預光柵化。前面講過，字體緩存管理器可以同時使用動態和靜態兩種機制。舉例來說，字體字型填充程式可以被設置為使大型字元集的字體使用動態緩存，而使只有**Basic Latin**字元的字體預先光柵化並使用靜態緩存。

前面講過，一般的策略是，如果靜態緩存可用，就使用預光柵化靜態紋理。否則如果動態緩存被啟用，就使用動態緩存。

默認狀態下字體字型填充程式是禁用狀態，這意味著**Scaleform**將對所有的嵌入式字體將使用動態緩存，除非程式明確創建並設定了填充參數。按照下列命令可以做到這一點：

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

然而，當使用**GFxExport**（以及各個**Scaleform**文件）時，字型可預光柵化。上述指令只是表示“載入時不要填充任何字型”；如果字型被**GFxExport**預填充，它們將被用來作為靜態緩存。

如果應該使用靜態緩存（嵌入式字型數量較少），可以按照以下方式進行配置。

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

SetUseSeparateTextures() 控制填充。如果它被採用，填充程式將對每一個字體使用單獨的紋理。

否則，程式會把所有字型填充的盡可能緊密。使用單獨的紋理可能減少紋理之間切換的次數，因此也會減少基本渲染的次數。但它通常會增加系統的已用記憶體和影像記憶體。其預設值為：無。

SetGlyphCountLimit() 這個參數控制填充字型的最大數目。預設值為0時，意味著數目沒有限制。如果字體的嵌入字型總數超過此限制，字體就不會被填充。當亞洲語言與基於拉丁文字的語言或其他語言一起

使用時，應該設定這個參數。如果您將這個限制設置到500，大部分的亞洲字體將被動態緩存（如果動態緩存被啓用），而對字型數目較少的字體使用靜態的紋理。

SetFontConfig()控制所有的紋理參數，它們是：

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

上述值是默認的。

NominalSize -存儲在紋理中反鋸齒字型的標稱尺寸（圖元）。此參數控制紋理中最大字型的尺寸；大多數字型比這個參數要小得多。這個參數也控制紋理記憶體的使用和大型文本的銳化之間的權衡。注意它被稱作“**NominalSize**”。與動態緩存不同的是，靜態緩存使用實際包圍盒填充不同大小的字型，而在動態緩存中字型是被拉到字型插槽的。**NominalSize** 這個值的大小與文本的圖元高度是完全一樣的值。這也意味著動態緩存的“解析度的能力”比靜態緩存稍微好一些。

PadPixels - 在單個字型圖像周圍留有空間的大小。這個值至少應為1。這個值越大，被縮小的文本的邊緣將越平滑，但同時也浪費了更多的紋理空間。

TextureWidth, TextureHeight - 字型將要被填充的紋理的尺寸，這些值將使用2冪運算。
幾種使用的情況及其意義列舉如下：

- 1) 一切都採用默認設置。動態緩存被啓用;字體字型填充程式未被使用。除了從預處理載入的預光柵化字型的紋理以外，其他部分都使用動態緩存。

- 2)

```
Ptr<FontPackParams> packParams = *new FontPackParams();
Loader.SetFontPackParams(packParams);
...
renderer->GetGlyphCacheConfig()->SetParams(Render::GlyphCacheParams(0));
```

字體字型填充程式一直被使用。動態緩存被禁用。

- 3)

```
Ptr<FontPackParams> packParams = *new FontPackParams();
Loader.SetFontPackParams(packParams);
Loader.GetFontPackParams()->SetGlyphCountLimit(500);
```

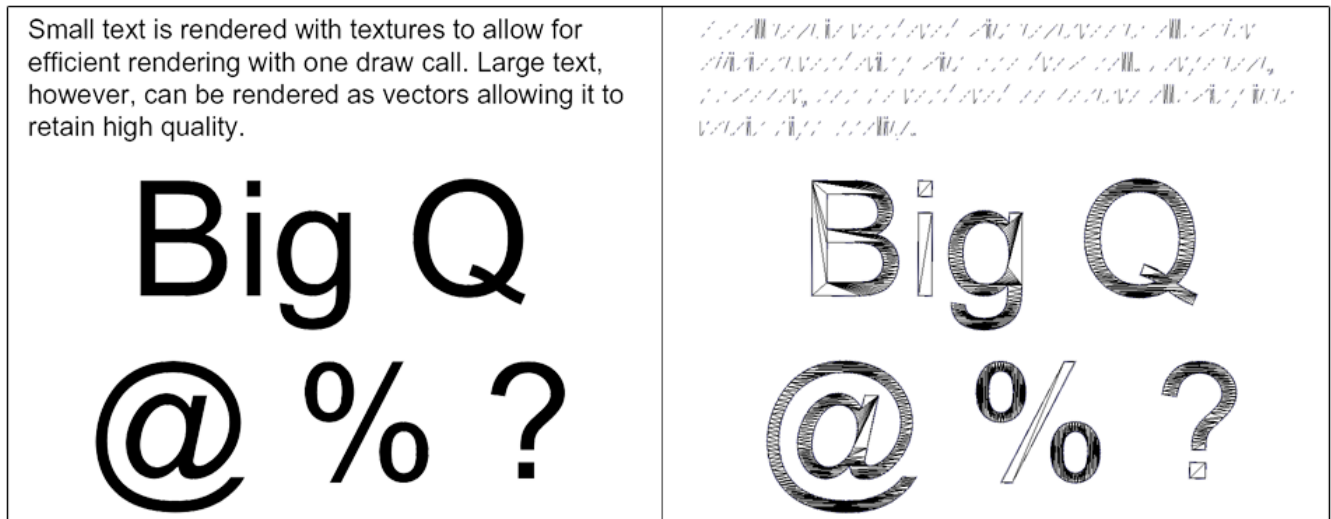
字體字型填充程式和動態緩存都被使用。含有500個或更少嵌入式字型的字體使用預光柵化靜態的紋理，其他的字體使用動態緩存。

同時有必要提到的是，這些功能中的一些僅適用於動態緩存。舉例來說，優化可讀性的文本含有自動圖元網格擬合功能（所謂的自動提示），只適用於動態緩存。任何效果，如模糊，陰影和輝光等也只適用於動態緩存。一般來講，字型填充程式和靜態緩存適合於性能欠佳的低預算系統。

5.6 向量控制

本文件前面已經介紹過，鑲嵌向量圖形在**Scaleform** 中用作渲染大型字型時的備選，因為大型字型不適合紋理緩存。因為在**Flash**中對個體文本字元的大小沒有限制，對它們進行紋理渲染所需的記憶體量在一個特定點之外就被禁止。解決這一問題的一個方法是限制字型點陣圖的大小，並從那個特定點開始使用雙線性過濾。但是，這樣做將導致渲染質量迅速下降。

為了能夠渲染大型的高品質文本，**Scaleform** 能夠將字型渲染轉換為三角形圖形，然後利用邊緣反走樣技術。在大多數情況下，用戶不會注意到這個轉換，因為當文本字元尺寸增大時，字型圖形的邊緣將保持平滑。下圖表明瞭紋理和三角形網格渲染的文本之間的差異。在 **Scaleform Player** 中，可執行按鍵 **Ctrl+W** 可對線幀模式進行切換，用戶可以看到渲染是如何完成的。



雖然三角形渲染的字型看起來不錯，但它們在渲染時需要更多的處理時間，因為三角形和原始渲染計數增加了。當使用動態緩存時，紋理字型的最高高度是由緩存記憶體體的 **MaxSlotHeight** 值決定的，這個值可以通過 **Render::GlyphCacheConfig::SetParams** 指令進行變更。如果將要顯示的字型放到紋理插槽的內部，這個字型會被紋理渲染，否則就使用向量渲染。由於要使用的渲染技術是根據每個字型確定的，所以當其圖元的高度接近緩存記憶體插槽的最高高度時，某一單行文本可能同時包含點陣圖和向量標誌。由於有亞圖元精度，進行不同渲染的符號的類型幾乎難以區分彼此。

靜態緩存的運作與此不同。靜態緩存時，所有字型都是根據字型的指定標稱尺寸預先光柵化的，並且使用三線性篩檢程式進行了調整。由於字型的標稱尺寸是固定的，文本渲染的方法也是根據字型的標稱尺寸進行選擇的，而不是根據個體字型的高度而選擇的。

對於動態緩存和靜態緩存，開發者可以通過修改 **GlyphCacheParams::MaxRasterScale** 值來控制紋理到向量渲染開關發生的點。

SetMaxRasterScale的參數指定了紋理槽的最大尺寸（圖元）的乘數，隨後就會切換到向量。**MaxRasterScale**的預設值為1。1.25這個值意味著除非螢幕上的字型尺寸大於存儲在紋理裏的字型的標稱尺寸的1.25倍，否則播放器將使用紋理渲染文本。因為默認的標稱尺寸值為48圖元，向量渲染將被用於在螢幕上大於60圖元的字型。

如果 **MaxRasterScale** 的值設置得足夠高，則向量渲染將永遠不會被使用。對於通過啓用 **-strip_font_shapes** 選項執行 **gfxexport** 工具而產生的 **Scaleform** 文件，向量渲染也是禁用。在後一種情況下，字體字型的資料不再保存在文件中，所以向量化也是不可能的。

6 第 5 部分：文本過濾效果和動作腳本擴展

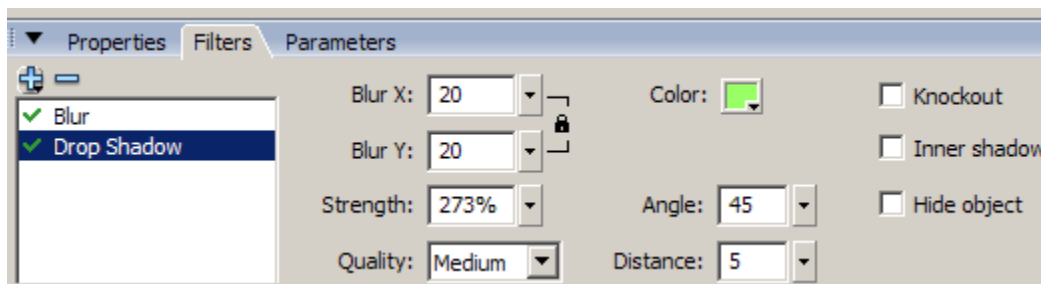
當動態字型緩存被啓用時，**Scaleform 2.x**支援將模糊、下拉陰影和輝光濾鏡這些效果應用到文本。支援文本篩檢程式需要動態緩存；當使用字型填充程式或靜態字體紋理時，系統不支援過濾效果。在播放器的當前版本，篩檢程式只有被直接用於文本域時才能運作，當被應用到動畫剪輯時，將不會有任何效果。

雖然**Scaleform**文本篩檢程式的運作類似於**Flash**，兩者之間還是有一些不同。在**Adobe Flash**中，篩檢程式持續適用於整個被光柵化的文本域中。這意味著模糊、下拉陰影、以及輝光篩檢程式可能會被相繼使用。這種方法具有很好的靈活性，但是計算量太大。與**Flash**不同的是，**Scaleform**內對篩檢程式的支援是有限的，但運行非常

快。與動態適配的字型緩存配合使用，篩檢程式的運作幾乎與普通的文本一樣快。雖然對篩檢程式的支援有限，**Scaleform**卻可以提供很好地製作柔化陰影和輝光效果的能力。

6.1 篩檢程式類型，可用選項和限制

在**Flash Studio**內，篩檢程式中可一次添加一個文本域。



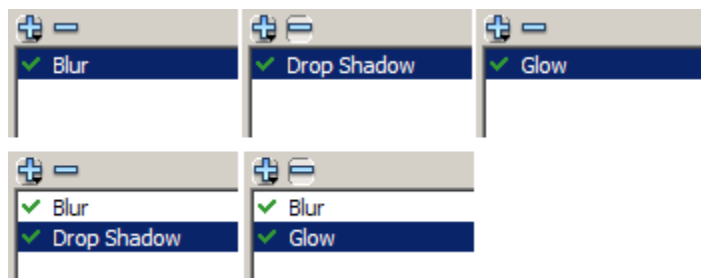
Flash和**Scaleform**篩檢程式之間的主要差別是，**Scaleform**在緩存中創建並存儲模糊化字型的點陣圖副本，而**Flash**將篩檢程式應用到所產生的文本域中。

Scaleform 只支援模糊和下拉陰影篩檢程式。輝光篩檢程式實際上是下拉陰影篩檢程式的一個子集。與**Flash** 不同的

是，篩檢程式不是被持續使用的，相反，他們作為兩個不同的層獨立運行。在 **Scaleform** 內，程式只考慮一個下拉陰影或輝光篩檢程式，即兩者中位於篩檢程式列表最後的那個。另外，一個可選的模糊篩檢程式也可用于文本本身。一般演算法如下。

- 通過篩檢程式列表疊代；
- 如果列表最後的是“輝光”或“下拉陰影”，則存儲陰影濾波參數；
- 每個“輝光”或“下拉陰影”覆蓋列表中的前一個“輝光”或“下拉陰影”；
- 如果列表最後的是“模糊”，則存儲模糊濾波參數；
- 每個“模糊”覆蓋列表中的前一個“模糊”。

最後組合成的有效的篩檢程式為：

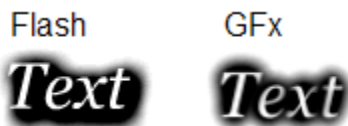


如果您在這個列表內添加下拉陰影和輝光篩檢程式，程式只考慮後者。

在Scaleform內的渲染也與Flash不同。先是“陰影”或“輝光”層被渲染，然後是文本本身被渲染上陰影，並有一個可選的“模糊”篩檢程式適用於原來的字型。如果“陰影”或“輝光”篩檢程式上顯示“Knockout”（脫模）或“Hide Object”（隱藏物件）標記，文本（第二層的）將不會被渲染。此時Scaleform不支援“Inner Shadow”（內部陰影）和“Inner Glow”（內部輝光）。

由於這個演算法的性質上的問題，在Scaleform內的篩檢程式有一定的限制。X和Y上的最大模糊值都被限制為15.9圖元。最大的陰影或輝光強度也被限制為1590%。

“Knockout”選項的運作也不相同。在Flash中，原始圖像是被整體脫去陰影的，同時保持陰影偏移量。而在Scaleform中，這個操作是逐個字型進行的，並忽略陰影偏移。此外，如果陰影或輝光半徑較大，字型圖像可能重疊和互相模糊：









因此，從基本上講，只有對於半徑較小的輝光篩檢程式（或對於陰影偏移為零的下拉陰影篩檢程式）來講，“Knockout”這個選項才是有用的。

6.2 過濾品質

Flash使用一個簡單的方塊濾波來模糊圖像。“品質”控制著篩檢程式運作的次數，而篩檢程式運作的次數嚴重影響生成圖像的效果。舉例來說，3x3的低品質模糊篩檢程式只能計算3x3圖元區域的平均值。中等品質意味著相同的篩檢程式被應用兩次；高品質意味著相同的篩檢程式被應用三次。這意味著品質對過濾的視覺半徑影響很大。因為方法不同，視覺的結果也不同，但是很類似。事實上，在一個簡單的、只應用一次的方塊濾波生成的結果質量很差。

與Flash不同的是，Scaleform使用的是一個很好的高斯模糊濾波器和智慧的遞迴演算法，其速度並不依賴於過濾半徑。在Scaleform內只有兩個品質層：低品質和高品質，並且它們能生產非常類似的視覺效

果。只有當篩檢程式面板上設定為“Quality: Low”（品質：低）時，程式才會應用低品質。否則，Scaleform 使用的是高品質的篩檢程式。兩個品質的區別在於，高品質的篩檢程式可以以分數的半徑值 (sigma) 進行操作，而低品質的篩檢程式使用的是整數半徑。在大多數情況下 Scaleform 類比分數半徑並適當調整字型。但如果文本域被反鋸齒化以優化閱讀，低質量的陰影可能看起來略有不準確。一般建議小型文本使用高品質的篩檢程式，以提高可讀性。

	Low quality	Medium quality	High quality
Flash			
GFx			

從上圖可以看到，在Flash中三種品質的差異是非常明顯的;而在Scaleform中則差異很小。在Scaleform中只有低品質與其他兩者不同，中品質和高品質產生的效果是相同的。

低品質篩檢程式運作更快，但字型緩存系統不存在這個差異。還有，低性能系統最好使用低品質的篩檢程式，尤其是當沒有可用的硬體浮點運算的時候。

高品質篩檢程式需要浮點計算，並使用一種遞迴的執行程式，在此鏈結中有詳細說明：

<http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Smoothin.html>.

6.3 動態過濾

通過使用 Flash 的時間表和 Scaleform 的動作腳本擴展，可以生成動態的陰影效果。不過，這裏有必要瞭解一下這種動態操作要付出的耗費。當改變半徑、強度、或品質時，Scaleform 不得不重新生成字型圖像並將其存儲在緩存中。這將導致更頻繁地更新緩存。事實上，更改前面提到的值相當於增加了字母的複合度。每個版本都得被儲存在緩存中。相反，改變顏色（包括透明度）、角度、距離，並不影響性能。舉例來

說，如果你想要實現陰影或輝光的淡入或淡出效果，最好使顏色的半透明度（透明度）動態化，而不是去改變半徑和/或強度。

6.4 使用 **ActionScript** 中的过滤器

Scaleform 支援標準的篩檢程式類(如 DropShadowFilter、BlurFilter、ColorMatrixFilter、BevelFilter),這些篩檢程式類用於 AS2 和 AS3 的動態/輸入文字欄位。有關更多詳細資訊,請參閱 Flash 說明文檔。