

Autodesk® Scaleform®

字体和文字配置

本文件描述了 Scaleform 4.2 中使用的字体和文本渲染系统，详细介绍了如何对素材资源和 Scaleform C++ APIs 进行国际化配置。

作者: Maxim Shemanarev, Michael Antonov
版本: 2.2
最后编辑: 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 的联系方式:

文件名	字体和文字配置概述
地址	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
网站	www.scaleform.com
电子邮件	info@scaleform.com
直接呼叫	(301) 446-3200
传真	(301) 446-3199

目录

1	引言：Scaleform 中的字体	1
1.1	Flash 文本和字体概要	1
1.1.1	文本域	2
1.1.2	字符嵌入	3
1.1.3	嵌入字体内存占用	4
1.1.4	控制字体内存占用	5
1.2	游戏用户界面的字体决策	6
2	第 1 部分：创建游戏字体库	7
2.1	字体符号	7
2.1.1	导出和导入字体符号	8
2.1.2	导出字体和字符嵌入	10
2.2	创建 gfxfontlib.swf 文件步骤	10
3	第 2 部分：选择国际化方法	12
3.1	导入替代字体	12
3.1.1	国际文本的配置	14
3.1.2	Scaleform Player 中的国际化	15
3.1.3	设备字体仿真	16
3.1.4	创建字体库的分步指南	17
3.2	自定义资源生成	19
4	第 3 部分：设定字体资源	20
4.1	字体查找的顺序	20
4.2	Gfx::FontMap	22
4.3	Gfx::FontLib	23
4.4	Gfx::FontProviderWin32	23
4.4.1	使用自动提示文本	24
4.4.2	设定自动提示	25
4.5	Gfx::FontProviderFT2	26
4.5.1	将 FreeType 字体映射入文件	27
4.5.2	将字体映射到内存	27

5	第 4 部分：配置字体渲染	29
5.1	配置字形缓存	30
5.2	利用动态字体缓存	31
5.3	使用字体压缩器 – gfxexport.....	33
5.4	预处理字体纹理 - gfxexport	34
5.5	设定字体字型填充程序.....	35
5.6	矢量控制	37
6	第 5 部分：文本过滤效果和动作脚本扩展.....	39
6.1	过滤器类型，可用选项和限制	39
6.2	过滤品质	40
6.3	动态过滤	41
6.4	使用 ActionScript 中的过滤器	41

1 引言：Scaleform 中的字体

Scaleform 载有一个新的灵活的字体系统，它可以提供高品质的HTML格式的可转换文本。使用这个新系统，可以从不同的字体来源选择替代字体用于本地化，包括局部嵌入的文本、共享GfX/SWF 文件、作业系统的字体数据、或FreeType 2 字体库。同时，字体渲染质量也得到显著改善，开发者可以在动画的性能、内存占用空间和文本的可读性之间折中取舍。

Scaleform 的很多字体特征依赖于Flash® Studio固有的功能，包括能将字体的字符集嵌入到SWF文件，以及能在有系统字体的情况下使用系统字体。但是，Flash Studio最初是为了开发个人文档而设计，因此限制了字体的本地化能力。尤其是嵌入式字体或翻译表不能在文档间得到轻松共享。此特征在内存的有效使用和游戏资源的开发方面特别重要。此外，Flash依赖于操作系统来处理未嵌入到文档中的国际字符的字型替代，但是游戏控制面板无法进行此项操作，因为游戏控制面板中没有系统字体。

Scaleform分别利用 GfX::Translator类别和GfX::FontLib类别作为所有可译文本和动态字体映射的中央回叫信号，解决了这些问题。同时，它还提供额外的接口以便于控制字体缓存机制和替代本地化过程中的字体名称，并可在适当时候使用操作系统和FreeType2字体查找。

为了有效使用Scaleform字体系统，开发人员需要了解Flash Studio和Scaleform运行时间的字体特征。通过了解Flash的字体特征，开发人员可以开发出前后连贯的字体，从而可以有效渲染出期望质量的字体，并占用最少的内存。通过了解Scaleform运行时间的字体特征，开发人员可以为某一特定平台选择一种尽可能在质量、性能和内存使用方面均最佳的配置。

本文件旨在指导那些不是非常熟悉Flash和Scaleform的开发人员了解游戏用户界面的字体安装程序。引言的其余部分安排如下：

- “Flash 文本和基本字体”一节介绍 Flash 文本和字体使用的基本知识。前两部分描述了文本域和字符嵌入的知识，熟悉 Flash 的开发人员可以跳过这两部分。
- “游戏用户界面的字体决策”一节介绍开发人员在创建一个 Flash 游戏的用户界面时将需要作出的与基本字体有关的决策。建议每一个开发人员都阅读本章，因为它描述了本文件其余部分的结构特征。

1.1 Flash 文本和字体概要

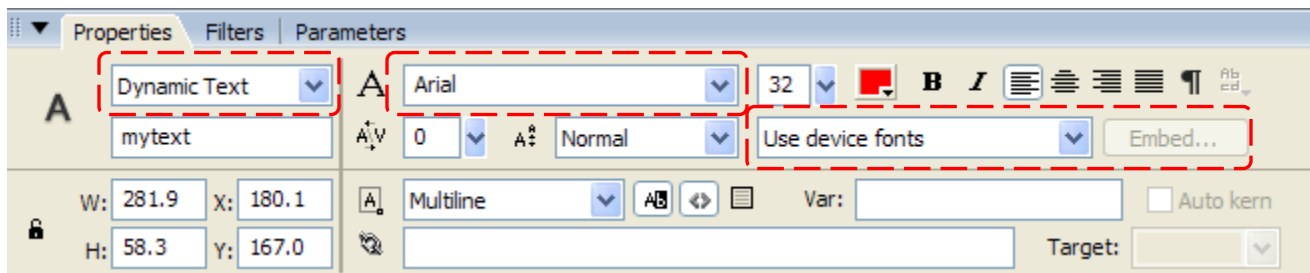
在 Flash Studio 中，开发人员首先用鼠标在平台上创建可视的文本域，然后输入相应的文本。应用到文本域中的字体是从安装在游戏开发人员的系统里的字体列表中按照名称选择的，并含有一个可用的嵌入选项，用于将内容放回不具备所需字体的系统中。

除了直接使用字体名称以外，开发人员还可以在字体库中创建字体符号，然后根据符号名称间接使用他们。通过与字符嵌入相结合，字体符号能够创建一个强大的概念框架，用于开发便携而连贯的游戏界面资源。本章的其余部分简要介绍了创建文本域和字符嵌入的基本知识。我们鼓励开发人员参考Flash Studio文件，以便得到更详细的解释。

Flash字体符号系统使用方便且易于安装，但它有若干限制，这使它很难共享字体以及进行国际化字体替代。而这些限制在Scaleform中得到了解决，详情见第1部分---创建游戏字体库。

1.1.1 文本域

要在Flash中创建文本域，开发人员首先应选择Text Tool，然后在平台上绘出相应的矩形区域。文本的各种属性，包括文本域类型、字体、大小和风格都设定在文本域属性面板中，文本域属性面板一般位于Flash Studio的底部。文本域属性面板如下所示。



虽然文本域可以有很多选项，但我们要讨论的关键属性就是上图中红线圈中的部分。一个文本域最重要的选项是位于左上角的文本类型，文本类型可设置为下列值之一：

- 静态文本
- 动态文本
- 导入文本

进行游戏开发时，动态文本属性是最常用的，因为它支持通过动作脚本修改内容，并支持通过使用用户安装的GFX::Translator 类别进行字体替代和国际化。静态文本不具备这些特点，它在功能上非常类似于矢量素材。导入文本可用来创建可编辑文本框。

上图红线圈中的其它两处分别是位于属性表最上面一行的字体名称，以及字体渲染方法。选择字体名称时，可以使用(a)开发人员系统中安装的任何一个字体的名称，或(b)在动画库中创建的任何一个字体符号的名称。在上面的例子中，“Arial”是选定的字体名称。此外，字体风格还可以通过粗体和斜体切换按钮进行设定。

对于国际化来说，字体渲染方法是最重要的设置，因为它控制着 SWF 文件的字符嵌入。在 Flash 8 中，它可以被设置为下列值之一：

- 使用设备字体
- 位图文本（无反锯齿）

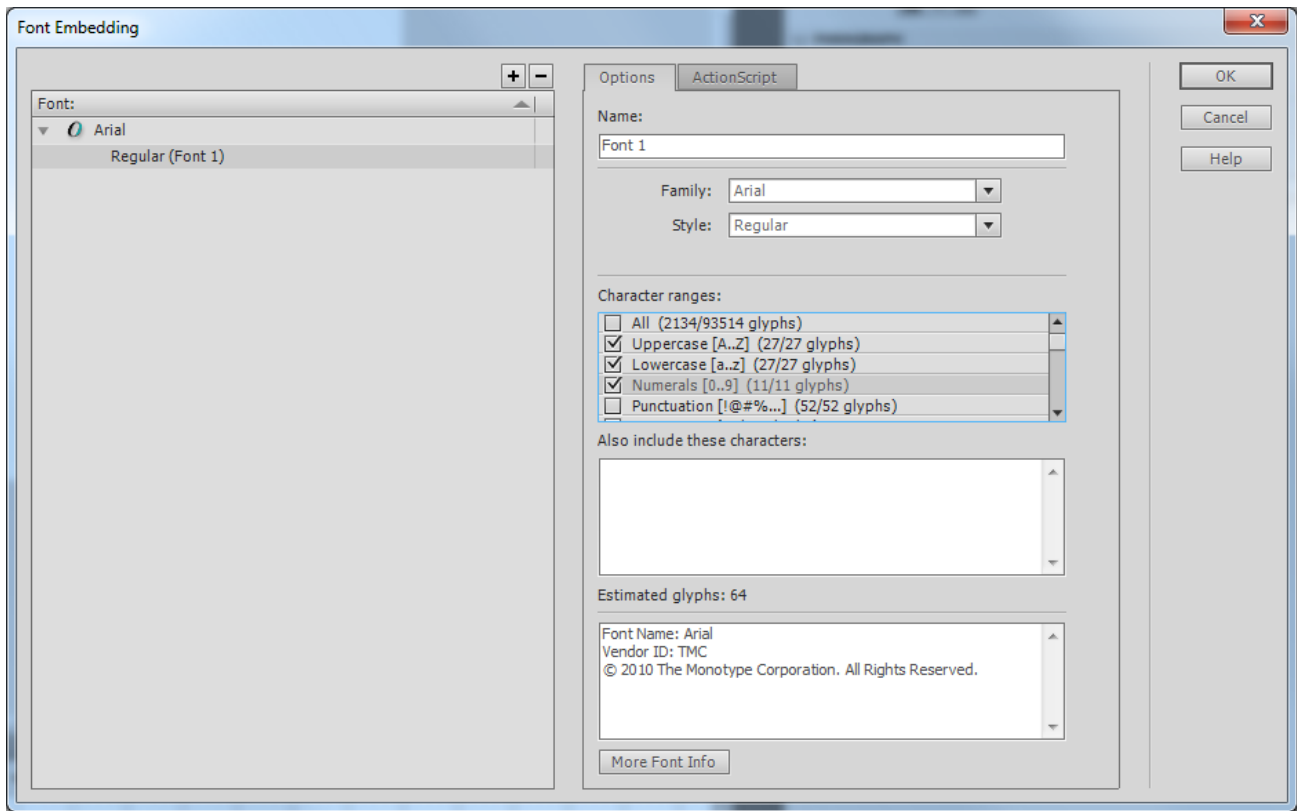
- 用于制作动画的反锯齿
- 用于提高可读性的反锯齿

如果您选择“**Use device fonts**”（“使用设备字体”），系统就会使用一个系统内置的字体渲染方法，这种方法的字体数据取自操作系统。使用设备字体的一个优势是，用这个方法创建的SWF文件较小，而且在播放时占用较少的系统内存。然而，如果当目标系统不具备所要求的字体时，**Flash**播放器将选择另一种字体代替，这时就会出现一些问题。这个替代字体可能与原来的字体看起来不一样，而且/或者不含有所有要求的字型。举例来说，如果一个游戏控制面板中没有操作系统字体库，它将显示不了文本，那些不可显示的字型在**Scaleform**中都将显示为小方块。

与使用系统字体不同的是，“用于制作动画的反锯齿”和“用于提高可读性的反锯齿”这两个设置依靠嵌入式字体字符，分别将字体渲染为动态性能，并且具有高品质的优化效果。只要这两个选项有一个被选中，嵌入按钮就会变成可用状态，此时用户可以选择嵌入字体的字符范围。

1.1.2 字符嵌入

为了使嵌入式字体渲染工作，必须针对文件中至少一个 **TextField**（文本字段）中的指定字体嵌入字符（假如导出字体，就不需要任何文本字段）。如果不嵌入必要的字符，系统就会采用设备字体，这样将带来上文所述的所有限制（除非在 **Scaleform** 中使用了 **Scaleform** 字体库，这一点稍后将作叙述）。如果要嵌入字体字符，用户需要按下 **Embed**（嵌入）按钮，此时将显示如下所示的字符嵌入对话框。



在这个嵌入对话框中，用户可以选择所需的嵌入字符范围，为文本域嵌入目前选定的字体和风格。使用相同字体风格的所有文本域将会共享相同的嵌入字符集，因此，通常只需设定嵌入其中一个文本域即可。

用户应该明白，从嵌入的观点看，字符的粗体和斜体属性是单独处理的（参阅上面屏幕截图上的“Style”（样式）组合框）。例如，如果同时使用普通的 **Arial** 和粗体的 **Arial**，则这两种字体风格要分别嵌入，这样就增加了文件的内存使用。不过，改变字体大小则不会不增加文件大小或内存占用。关于字体和字符嵌入的更多详情，请参阅 **Flash Studio** 文件中的“字体使用”章节。

嵌入字符是唯一真正便携地使用字体的方式。当字体的字符被嵌入时，其矢量表示即被保存到SWF/GFx文件，并且在后来使用时被加载到内存。

因为字型数据是文件的一部分，所以无论系统正在使用哪一种安装的字体，它都会永远准确地渲染。在 **Scaleform** 中，嵌入的字体在游戏控制面板（如Xbox 360, PS3, Wii）和台式电脑（如Windows、Mac、Linux）上的效果同样良好。使用嵌入式字体无可避免的副作用是，它会增加文件的大小和内存的使用。由于可能会大幅度地增加文件的大小，特别是使用亚洲语言的时候，开发人员将需要提前规划游戏的字体使用，并尽可能共享字体。

1.1.3 嵌入字体内存占用

要了解字符嵌入在 **Scaleform** 中所占用的内存，请参考下列表格。该表概述了当嵌入的字符数增加时，内

存被占用的情况。

嵌入的字符数	未被压缩的 SWF 文件大小	Scaleform 运行内存占用
1 个字符	1 KB	450 K
114 个字符-拉丁文 + 标点	12 KB	480 K
596 个字符-拉丁文和西里尔字母	70 KB	630 K
7,583 个字符-拉丁文和日语	2,089 KB	3,500 K
18,437 个字符-拉丁文和繁体中文	5,131 KB	8,000 K

这个示例表是根据嵌入“Arial Unicode MS”字型创建的。正如上文所述，加入欧洲字符集占用相对较少的内存，开发人员加入 500 个字符，大约增加 150K 的内存使用，这对于几种不同字体风格在本地分配来说是足够的。但是对于亚洲语言来说，由于它们字型数量庞大，它们占用的内存也会大大增加。

1.1.4 控制字体内存占用

因为嵌入大量的字符集会占用数兆的内存，开发人员将需要提前计划字体使用，以减少内存占用量。以下几种方法可以用来控制字体所占用的内存：

1. 在开发用户界面的素材资源之前，预先确定字体集内使用的字体和字体风格。粗体和斜体的字体风格在 **Flash** 中是作为单独的字符集分别嵌入的，因此应将他们视为完全独立的字体，并且只在必要时使用他们。不过，不同的字体大小不会占用任何额外的内存空间，所以使用不同的字体大小不会造成内存增加。将来，我们将提供一个假的粗体和斜体的选项，避免粗体和斜体的版本需要储存额外的字体字符。
2. 通过使用 **Scaleform** 字体库和/或导入字体使文档共享字体。在每一个单一的文件内嵌入相同的字符，可能会不合理地增加资源大小、内存占用（如果同时加载稍后的文件）和加载时间。如有可能，最好是将嵌入的字体存储入单独的 **SWF/GFx** 文件，这样的文件是共享文件，从而不需要重复占用内存或重新载入。**Scaleform** 字体库的使用将在本文件后面的部分加以讨论。
3. 在亚洲字符集中只嵌入所用的字符。在亚洲本地化游戏中，只嵌入在游戏文本中使用的字符，尽量避免在语言中嵌入所有字符。经过翻译，所有的游戏字串可以通过扫描而生成所需的特殊字符集，这些特殊字符集随后被嵌入到游戏中。只要游戏不要求通过 **IME** 导入任意动态文本，那么对字符集加以限制就可以大大节省内存空间。
4. 考虑使用 **GfxEport** 字体压缩（**-fc** 选项）。一般情况下，可使大小实际减少 10%-30%，而且不会明显降低质量（参阅第 5.3 节）。

为得到最佳效果的国际化游戏，用户可以选择结合所有这些技术，来限制所使用的字体，并通过 Scaleform 字体库共享这些字体。对于亚洲语言的游戏，如果只嵌入所用的或IME所要求的字符，以及只嵌入一套完整的字体，都将可以大大节省内存空间。

1.2 游戏用户界面的字体决策

在Scaleform内创建游戏的用户界面时，开发人员将需要针对有关字体的使用、配置和国际化方法等问题作出一些决策。本文件细分了四个独立的决策分类：

- **第1部分：创建游戏字体库**——帮助您在创建字体库时决定所要使用字体的数目和风格。
- **第2部分：选择国际化方法**——描述了创建国际化游戏资源的不同方法。
- **第3部分：设定字体来源**——描述字体查找的顺序以及如何在 **Scaleform** 内配置不同的可用字体来源。
- **第4部分：设定字体渲染**——描述 **Scaleform** 文本渲染的方式和其所拥有的不同配置选项。

创建游戏字体库是指选择游戏界面将使用的字体风格，这个步骤通常应该在开发游戏用户界面资源之前进行。在艺术字体方面，使用标准的字体库是非常重要的，这样可以确保在所有的用户界面文件中为同一目的而前后一致地使用不同的字体。在技术方面，控制嵌入式字体的数目十分重要，因为这样可以使它们符号应用程序的内存预算。

一旦决定了游戏的字体，下一步就应该决定如何使他们国际化。第2部分论述了三种可能的国际化模式：

- **导入替代字体**，由导入程序和独立的具体语言字体文件决定，这些文件被加载到播放器并通过 **Scaleform** 字体库共享。
- **设备字体仿真**，与上述模式类似，但这个模式利用的是设备字体而不是导入的字体，并且需要适当的系统字体的支持（目前的游戏控制面板上没有）。
- **自定义素材资源生成**，此种模式利用 **Flash** 创建的翻译系统为每个目标市场生成 **SWF/GFx** 文件的自定义版本。

开发人员可以选择一个模式，也可以将这些模式适当组合起来，然后按照所选定的模式进行游戏开发。

因为Scaleform中的字体可能来自多个字体来源，所以正确设定字体非常重要。为了帮助完成字体设定，第3部分“设定字体的来源”详细描述了Scaleform中的字体查找程序，包括一些介绍如何设立Scaleform字体库和字体提供程序的实例。字体提供程序是一个用户可以配置的类别，用来支持系统字体和FreeType 2 字体。它可以代替Scaleform字体库进行存取字体，而不需嵌入字体。

本文件的第四部分“设定字体渲染”描述了在 **Scaleform** 中采用的字体渲染方法，并着重描述了三个可用的初始化字体纹理选择，即在预处理时由 **gfxexport** 对纹理进行预渲染，将加载时间所产生的静态纹理进行打包，以及使用按照要求更新的动态纹理缓存。本文件讨论了每种方法的可用配置选项，以便开发人员可以为应用程序和目标平台选择正确的配置。

2 第 1 部分：创建游戏字体库

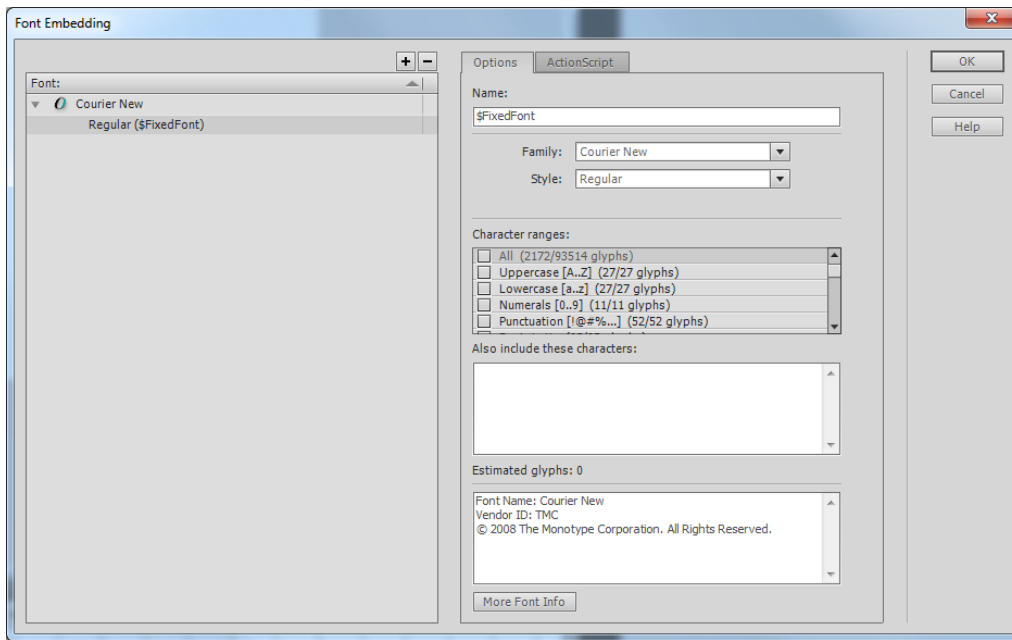
开发人员在开始进行游戏用户界面开发时，首先应确定一套将用于整个游戏的潜在的字体样式。例如，他们可以决定为所有的标题设立独特的字体类型，而在游戏其余的文本中使用另一种类型。确定之后，如果没有经过非常审慎的考虑，游戏的用户界面屏幕不应使用任何额外的字体类型。之所以这样限制，主要有以下三个原因：

1. 通过使用相同的字体，开发人员可以确保呈现给用户一致的游戏内容。如果不同的游戏画面使用不同的字体，将会产生混乱的用户界面，使用户难以阅读和理解。
2. 在国际化阶段，开发软件使用的字体往往需要被含有目标语言字符的字体所替代。如果有一套预先决定的固定的字体集，操作起来会更容易。
3. 如果有一套固定的字体集，则许多用户界面的画面可以共享这套字体集，这样字体的数据就可以在内存中被共享，从而大大降低内存的使用，减少画面加载时间。这是一项非常重要的技术措施，因为字体的数据需要占用大量的存储空间。

在 **Scaleform** 下，下一部分所述的导入字体替换方法，为某个游戏选择字体集的过程形式化为创建一个字体库，该库由一组文件构成，每种语言一个文件（例如：“**fonts_en.swf**”、“**fonts_kr.swf**”等）。创建了这个字体库文件后，库内的字体符号可以被导入到其他**Flash**文件并在整个开发阶段使用。下面的章节详细描述了如何创建字体符号，以及如何利用字体符号创建一个游戏字体库。

2.1 字体符号

在引言中，我们介绍了如何将字体应用于文本域，以及如何嵌入这些字体的字符用于便携式播放。在**Flash Studio**内，除了可以使用文本域属性的系统字体外，开发人员还可以定义新的字体符号，具体做法为：把鼠标移动到字体库位置，点右键然后选择“**New Font...**”项，此时显示下列对话框：



用这种方式创建的字体符号被添加到 FLA 文件的字体库中，并可以随后应用于类似常用系统字体的文本域中。例如，如果您将您的游戏字体命名为“\$FixedFont”，您就能可以选择这个名字作为您文本域的有效字体。

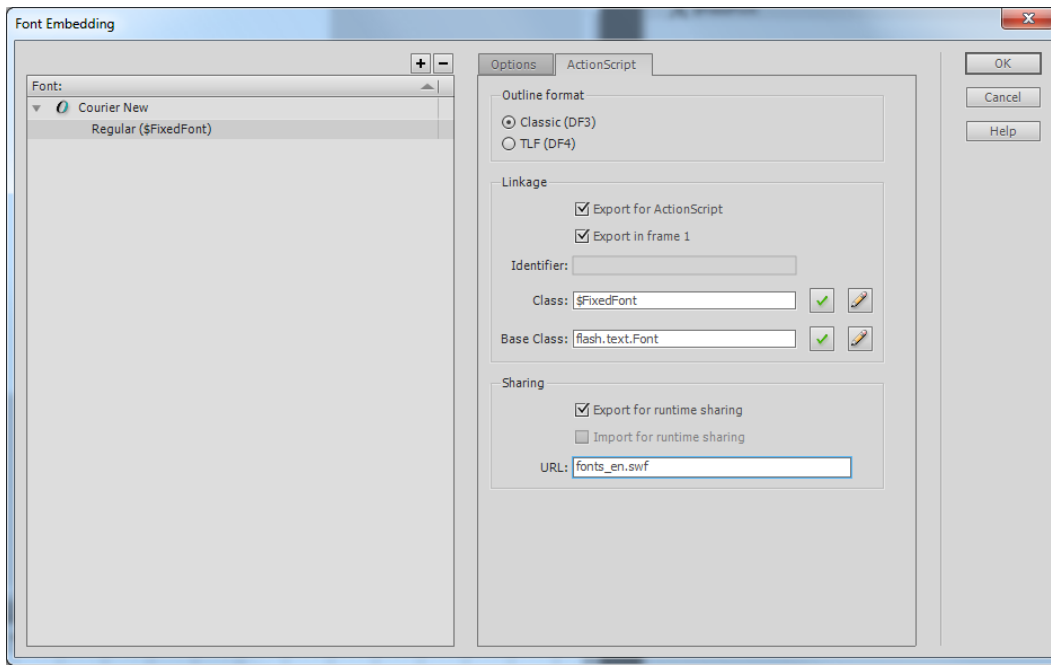
给字体库加上字体名称有利于发现以独特方式使用的字体。当字体库的字体在字体列表中显示出来时，在每个字体旁边都有一个星标，例如“\$ArialGame*”。如果开发人员只使用字体库中的字体名称，这个方法将有助于确保没有错误地引用用户界面文件中的其他字体。此外，如果所有字体符号都以美元符号字符“\$”开头来命名，它们将永远显示在文本域属性中字体列表的上方，给开发工作提供帮助。

一些网站推荐使用下划线字符“_”作为字体符号名称的前缀。虽然下划线看起来更整洁，但是使用下划线会导致文本域“Font rendering method”（“字体渲染方法”）这个选项框遭到破坏，因为Flash Studio会把任何一个以下划线开头的字体名称看作是一个内置字体。为了避免出现这个问题，我们选择使用美元符号字符。

2.1.1 导出和导入字体符号

与字体库其他条目相似，其他文件可以通过复制其内容或利用导入/导出机制，使用位于FLA文件的字体符号。要复制一个字体库的符号，先右击FLA字体库来源，然后选定“Copy”选项；做完这一步之后，切换到目标FLA文件字体库，右击这个目标FLA文件字体库并选定“Paste”选项。此时即成功复制出一个字体符号，同时这个符号的所有数据内容也在第二个文件中复制出来。

如果要避免复制数据，可使用Flash的导出/导入机制。要导出一个库符号，开发者可以右击该符号并选择“Properties”（属性），然后选择“ActionScript”选项卡，此选项卡将会显示下面的属性表：



通过按“OK”（确定）按钮，Flash 就会警告没有类定义。忽略此警告信息是安全的。

导出一个符号时，会为其指定一个导出标识符，并给其标上“Export for runtime sharing”（针对运行时共享导出）、“Export for ActionScript”（针对 ActionScript 导出）和“Export in first frame”（在第一个帧中导出）标志。建议设定与符号名称相同的导入标识符。URL 应给此 SWF 文件指定路径，因为导入此符号时会用到该路径；假如打算通过一个 Gfx::FontLib 对象替换此字体符号，就应将 URL 字段设为您的默认字体库。本文中我们将使用“fonts_en.swf”作为默认名称，不过，也可以使用任何其他名称。**请注意，必须在 Gfx 中设置默认字体库才能使用字体替换。**

例如：

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

您可以在 fonconfig.txt 中为 GfxPlayer 设置默认字体库。

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

第一个 fontlib 外观将用作默认库

一旦一个符号被导出，它就会在字体库的链接栏被标上“Export: \$identifier”标签。当对一个导入的字体库符号进行复制/粘贴或拖放操作时，该符号将不会被复制，而会在目标文件创建一个导入链接。这意味着该目标文件会更小，并且当被加载到内存中时，它将拖出其从源 SWF 导入的数据。

在 Scaleform 内，即使数据是从多个文件导入的，导入的 SWF 的数据也只会载入一次，这样就大量节省了系统内存。

字体符号的名称将不会被作为 TextField.htmlText 字符串的一部分或作为一个在 TextFormat 的字体名称而

返回；相反，系统字体的原始名称，例如“Arial”将被返回。同样地，除非字体符号能为一个导出的字体匹配导入名（不推荐），否则将不能通过动作脚本指定符号名称。

2.1.2 导出字体和字符嵌入

CS4 Flash Studio 以上版本才允许指定要针对导出的字体嵌入的字符。相反，要嵌入的字符集是通过系统区域设置来确定的。在 Windows 中，这一点由“控制面板\区域和语言选项\高级”内的“用于非 Unicode 程序的语言”设置控制。如果语言设置为“英语”，则每个字体将只有 243 个字型被导出。如果设置为朝鲜语，则将有 11920 个字型被导出。很明显，这不利于游戏开发，因此，创建了采用‘gfxfontlib.swf’的整套方法。

幸运的是，从 Flash CS5 起，可以指定要导出的字形（Glyph），因此不再需要‘gfxfontlib.swf’。开发者可以创建针对语言的字体库（如‘fonts_en.swf’、‘fonts_jp.swf’等），并直接指定应在这些字体库中嵌入的字形。

不过，假如仍然在用 CS4 Flash Studio（或更旧版本），就必须使用‘gfxfontlib.swf’方法（该方法仍然受到支持，不过不建议在当前版本中使用）。有关此方法的详细信息，请参阅针对 GFx 4.0 和更低版本的此文档的旧版本。

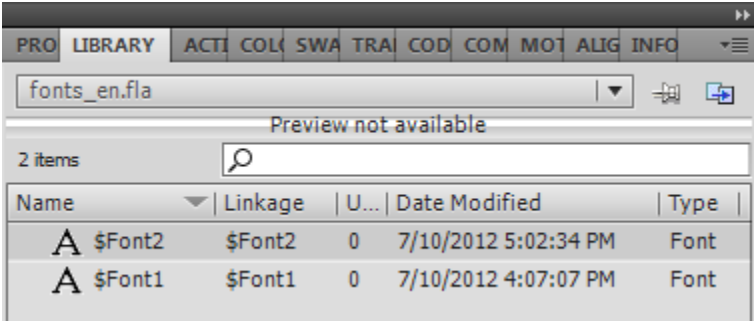
2.2 创建 gfxfontlib.swf 文件步骤

了解了字体符号后，我们现在就可以创建字体库文件了。要重申的是，创建这些文件是为了提供按语言的源字体。下一节将介绍如何映射和替换库字体的详细信息。

要创建针对某种语言的字体库文件，请在 Flash Studio 中创建一个新 FLA，并用字体符号填充其库。具体来说，你可以按照下列步骤进行操作。

1. 指定用于所有游戏画面的字体，并根据它们的目的给他们单独命名。对于游戏的标题和正常大小的字体来说，‘\$TitleFont’和‘\$NormalFont’都是不错的名称。
2. 为字体库创建一个新的 FLA 文件。
3. 为在第 1 步指定的每个字型创建一个新的字体符号。
4. 如果打算使用导入的字体替代，那么您应该为每个字体配置链接属性，以便使它在导出时携带与字体符号名称相同的标识符。
5. 指定应嵌入的字形。
6. 将产生的文件另存为 fonts_<lang>.swf，其中‘lang’表示语言（例如：‘en’– 英语、‘ru’– 俄语、‘jp’– 日语、‘cn’– 汉语、‘kr’– 韩语等）。此名称可以是任意名称，但以后在 FontMap 配置中会用到此名称。

除非出于文档编写目的想使用其他文本字段，否则不需要将任何文本字段添加到一个字体库阶段中。除了字体以外，不应将其它类型的符号添加到字体库文件中。**FLA**完成后，字体库显示出类似于下图的状态。



字体库完成后，其符号就可以在用户界面屏幕的其余部分使用了。为了能够在导入文件中使用字体库的字体，您可以将字体符号下拉到目标动画平台中，或使用前面介绍的复制/粘贴技巧。

3 第 2 部分：选择国际化方法

在用户界面国际化的过程中，两个主要条目，即文本域字符串及字体，需要被替代。替代文本域字符串是为了实现语言翻译，用目标语言中相应的词组取代开发文本。替代字体是为了为目标语言提供正确的字符集，因为原本的开发字体可能不包含所有需要的字符。

在本文件中，我们介绍了三种不同的方法，您可以利用任何一种对艺术字体资源进行国际化。这三种方法分别是：

1. 导入的字体替代。
2. 设备字体仿真。
3. 自定义素材资源生成。

导入替代字体是推荐的字体国际化方法，这个方法利用原本由 default ‘fonts_en.swf’ 字体库提供的、用于取代字型符号的 `GFX::FontLib` 和 `GFX::FontMap` 对象。

设备字体仿真与导入替代字体类似，不同的一点是这个方法利用字体映射而非其导入的符号，来提供实际的字体。这个方法容易设置，但有若干限制。

导入替代字体和设备字体仿真这两种方法都依赖于用户创建的用于翻译文本字符串的 `GFX::Translator` 对象。

与上述两种方法不同，自定义素材资源生成不利用字体映射或翻译对象，而利用 **Flash Studio** 的特点，为每个目标语言生成自定义的用户界面资源文件。对于含有少量静态的用户界面资源、内存极其有限的平台来说，选择这种做法比较适合。

3.1 导入替代字体

导入替代字体这个方法利用 **Flash** 的字体符号导入/导出机制来将文本域绑定到可替代字体。要使用此方法，开发者首先创建一个默认字体库文件，名称可以是，例如，第 1 部分所述的 “fonts_en.swf”，然后在所有游戏 UI 文件中使用从该文件导出的字体符号。当在 **Adobe Flash** 播放器中被测试时，它们会被从 “fonts_en.swf” 中导入，并用开发语言对其内容进行正确处理。然而，当在 **Scaleform** 中运行这些资源时，国际配置却可以被载入，使翻译和字体替代得以完成。

请注意，应该调用 `Loader::SetDefaultFontLibName (const char* filename) C++` 方法，并将一个文件名（仅仅是文件名，不包含路径！）作为一个 ‘filename’ 参数传递。例如，假如默认开发语言为 “韩语”，您需要进行如下调用：

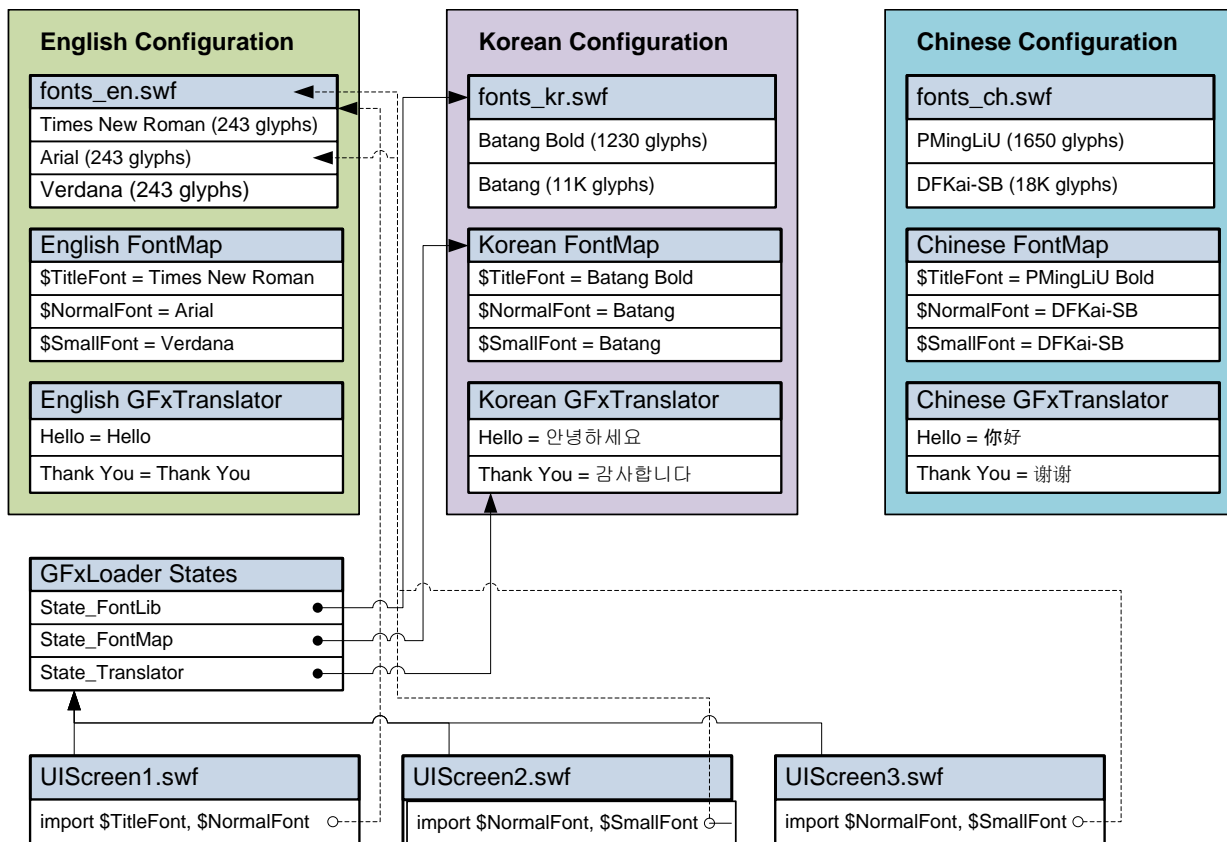
```
Loader.SetDefaultFontLibName("fonts_kr.swf");
```

应在针对主要 SWF 文件调用 `CreateInstance` 之前调用此方法。

从根本上说，游戏的国际配置由以下数据集组成：

1. 字符串翻译表，这个表将游戏中的每一个短语映射到其翻译版本。
2. 一套字体，它们为翻译中使用的字符提供字型。
3. 字体替代表格，它将如“\$NormalFont”这样的字体符号名称映射到配置中可用的字体。

在 `Scaleform` 中，这些组件表现为 `GFx::Translator`、`GFx::FontLib` 和 `GFx::FontMap` 三个状态，它们可以安装在 `GFx::Loader` 上。当一个动画文件被下载并含有调用指令 `GFx::Loader::CreateMovie` 时，装入程序根据字体配置状态中渲染文本的字体，将这个动画文件与字体配置状态绑定。要创建一个不同的字体绑定，用户需要在装入程序中设置一个新的绑定状态并再次调用 `CreateMovie` 指令。



上图表示的是一个潜在的国际字体设置。在这种设置内，三个名为“UIScreen1.swf”的用户界面文件通过“UIScreen3.swf”从“fonts_en.swf”字体库导入一套字体。导入文件链接由虚线箭头线表示。在下

述情况下此替换将会自动发生：(1) 导入文件与 `Loader::SetDefaultFontLibName()` 方法指定的那个导入文件相匹配，以及 (2) 加载 UI 屏之前在加载程序上设置了一个非空 `GFX::FontLib` 状态。

为了支持国际翻译,上图使用了三个配置：英语，韩语和中文。每个配置包括自己的字体文件和一个字体映射，如朝鲜语配置中含有“`fonts_kr.swf`”文件，而字体映射可以将导入的字型符号名称映射到嵌入到字体文件中的实际字体。在我们的例子中，载入程序状态设定为朝鲜语的配置，所以能创建朝鲜字体和翻译表以供使用。然而，载入程序的状态可以简单地被设定为中文或英文的配置，并将用户界面翻译成这两种语言。进行这样的翻译时不需要修改 `UIScreen` 文件。

3.1.1 国际文本的配置

当对含有导入替代字体的游戏进行国际化时，开发人员首先需要创建国际化艺术资源和所需的语言配置。通常艺术资源和字体资源会以SWF格式储存（调用前可转换为Scaleform格式），而字体映射和翻译表中的数据可以以具体游戏的数据格式存储，这种数据格式支持创建`GFX::FontMap`和`GFX::Translator`对象。在Scaleform样例中，我们利用“`fontconfig.txt`”配置文件提供这方面的资料；然而，游戏开发人员应该创建一个更先进的生产方案。

形式上，创建国际化配置的过程可细分为以下步骤：

1. 创建一个如第 1 部分所述的默认库文件（如“`fonts_en.swf`”），并用它来创建您的游戏素材。
2. 为每种语言创建一个翻译字符串表，这个翻译字符串表可以用来创建一个相应的 `GFX::Translator` 对象。
3. 决定每种语言的字体映射；将这个映射以可以用来创建 `GFX::FontMap` 的格式存储。
4. 为每个目标语言创建针对语言的字体文件，如“`fonts_kr.swf`”。

国际配置创建后，开发人员可以将国际化的用户界面屏幕加载到他们的游戏中。选定目标语言后，采取下列步骤：

1. 为这种语言创建一个翻译程序，并将它设置于 `GFX::Loader` 上。从 `GFX::Translator` 引出你自己的类别并覆盖翻译虚拟函数，即可进行翻译。开发人员可以通过覆盖这个类别，用他们选择的任何格式显示翻译数据。
2. 创建一个 `GFX::FontMap` 对象，并将它设置于 `GFX::Loader` 上。通过调用 `MapFont` 功能，给它添加必要的字体映射。第 3 部分提供了一个利用 `GFX::FontMap` 的例子。
3. 在加载器上设置游戏所必需的与任何其它字体相关的状态，例如，`GFX::FontProvider` 和/或 `GFX::FontPackParams`。如有必要，通过渲染线程上的 `GlyphCacheConfig` 来配置动态缓存。
4. 创建一个 `GFX::FontLib` 对象，并将它设置于 `GFX::Loader` 上。

5. 针对默认字体库（swf 内容中使用的那个字体库），调用 `SetDefaultFontLibName(filename)` 方法。

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

创建 `GFX::FontLib` 后，载入目标语言使用的字体源 SWF/GFx 文件，并通过调用指令 `GFX::FontLib::AddFromFile` 将其添加到字体库。通过这一调用可以使嵌入在参数文件中的字体被作为字体导入或设备字体使用。

6. 通过调用 `GFX::Loader::CreateMovie` 载入用户界面文件。字体映射、字体库和翻译状态将自动适用于这个用户界面。

3.1.2 Scaleform Player 中的国际化

为了演示国际化，Scaleform Player支持使用国际化用户参数文件，通常称为“fontconfig.txt”。当您拖动SWF/GFx文件到播放器时，如有可用，这个‘fontconfig.txt’配置将自动从本地文件目录载入；也可以通过命令行的/fc选项将其载入。载入国际化用户参数后，用户可以通过按Ctrl+N键切换它的语言配置。按F2键，当前的配置就会在播放器的HUD底部显示出来。

国际用户参数文件既可以保存为 8 字节的 ASCII，也可以保存为 UTF-16 格式，并通过列出带有各自属性的字体配置而形成线性结构。下列的国际用户参数文件可以用来说明前一节介绍的游戏设置。

```
[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" = "谢谢"
```

从这个例子可以看出，有三个单独的配置部分，每一个都以一个[FontConfig "name"]标题开始。当选中某个配置时，该配置的名称就在Scaleform Player的HUD处显示出来。在每一个配置内部，将使用适用于该配置的指令。可能使用的指令如下表。

配置语句	含义
fontlib "fontfile"	将指定的 SWF/GFx 字体文件载入 GFx::FontLib。第一个 fontlib 外观将用作默认字体库。
map "\$UIFont" = "PMingLiU"	将一个条目加入到GFx:FontMap，GFx:FontMap把在游戏用户界面屏幕上使用的一个字体映射到字体库提供的一个目标字体。导入替代字体后，\$UIFont应是最初“fonts_en.swf”导入到UI文件的字体符号的名称。
tr "Hello" = "你好"	将源字符串的一个翻译添加到它在目标语言中的对应字符串。开发人员应该使用更先进的解决办法来存储翻译表。

用户应该知道，配置文件的功能最初是作为一个例子，用于展示如何实现Scaleform的国际化。配置文件的结构今后可能会改变，将不另行通知；此外，随着Scaleform国际化功能的演进，我们计划将其过渡到XML格式。开发人员可以参照FontConfigParser.h/.cpp文件中的源代码，来了解如何配置字体状态。

Bin\Data\AS2\Samples\FontConfig and Bin\Data\AS3\Samples\FontConfig 目录中有一个更为完整的Scaleform 入替代字体的例子。该目录包括 “sample.swf” 文件和一个“fontconfig.txt” 文件，“sample.swf” 文件可以被放入 Scaleform Player，“fontconfig.txt” 文件可以将目录翻译成各种语言。我们鼓励开发人员对该目录的文件进行研究分析，以更好地理解国际化程序。

3.1.3 设备字体仿真

设备字体仿真与导入替代字体不同的是，它不利用导入的字体符号，而是依靠GFx::FontMap 替代系统字体名称。在创建用户界面的阶段，开发人员选择一套开发系统字体，例如“Arial”和“Verdana”，并将直接将他们作用用户界面所有资源的设备字体。当一个字体配置文件在Scaleform Player中运行时，它可以将系统字体的名称映射到它们在目标语言中的对应字体。举一个此类映射的例子，“Arial”在朝鲜语的用户界面可以被映射到“Batang”。反过来，“Batang”字体又可以通过使用GFx::FontLib从“font_kr.swf”文件被载入。

如果开发人员计划不依赖GFx::FontLib，而通过GFx::FontProviderWin32直接使用系统字体，或通过GFx::FontProviderFT2直接使用字体文件，设备字体仿真将是非常有帮助的。虽然这种做法看起来很容易设置，但由于设备字体的一些限制，导致使用设备字体不如利用导入的字体替代灵活：

- 设备字体在 Flash 不能被转换。如果设备字体应用了任何旋转的转换，Adobe Flash Player 将无法显示设备字体文本域。此外，播放器将不能准确调整设备字体文本，并会忽略应用于它的掩码。虽然所有这些功能在 Scaleform 中将正常运行，但由于 Flash 中支持有限，使得用户界面的艺术资源的测试更为困难。

- 设备字体不能配置有可视的“Anti-alias for animation”设置。除非 ActionScript 中的 TextField.antiAliasType 属性另有说明，否则 Scaleform Player 将一直为了提高设备字体文本可读性而使用反锯齿。
- 当使用设备字体时，开发人员必须重新编辑所有用户界面资源才能修改开发字体。当然，在 Scaleform 中可以使用字体映射，但当文件在 Adobe Flash 播放器中被测试时字体映射就不能使用了。相反，如果使用导入的字体替换，就可以通过简单地编辑和重新生成默认字体库文件 (“fonts_en.swf”) 来选择默认开发字体。
- 在处理 ActionScript 内的 TextFormat 时，Flash 只能辨认导入的字体符号的名称。这就是说，开发人员将需要使用直接的字体名称，例如“Arial”，而不应该使用符号的名称，如“\$TitleFont”。

设备字体仿真中，虽然在开发过程中“fonts_en.swf”文件作为一种字体的符号贮存库仍然是有用的，但不一定必须使用该文件。如果开发人员选择使用“fonts_en.swf”文件，那么就不应导出其字体符号。相反，可以将这些符号复制到目标用户界面的文件中，在那里他们可以作为他们所代表的映射设备字体的别名。

3.1.4 创建字体库的分步指南

下面我们介绍创建一个简单 SWF 的步骤，该 SWF 使用采用两种语言的字体库，并使用 fontconfig.txt。

1. 创建一个新的 FLA，并选择 ActionScript 2.0 或 3.0。将其称为‘main-app.flaswf’。这将是一个使用字体库的主要应用程序 SWF。
2. 创建另一个 FLA，选择您在上一步所选择的同一 ActionScript。将其称为‘fonts_en.swf’。这将是我们的默认字体库。
3. 转到‘Library’（库）窗口，单击鼠标右键，选择‘New Font...’（新建字体...）。将名称指定为 \$Font1，选择“Family”（家族）为“Arial”，“Style”（样式）为“Regular”。
4. 在“Character ranges”（字符范围）中，选择要嵌入的字符。对于 English（英语），选择‘Basic Latin’就足够了。
5. 切换到‘ActionScript’选项卡。复选‘Export for ActionScript’（针对 ActionScript 导出）、‘Export for frame 1’（针对帧 1 导出）、‘Export for run-time sharing’（针对运行时共享导出）。在‘URL’中，输入字段类型‘fonts_en.swf’。单击‘OK’（确定）按钮。对于 ActionScript 3.0，Flash Studio 将给您显示一条警告‘A definition for this class could not be found...’（找不到此类的定义...）– 忽略此警告，并再次单击‘OK’（确定）。
6. 为另一个字体重复步骤 3 – 5，将其命名为‘\$Font2’，并选择不同的‘Family’（家族）和/或‘Style’（样式）。在两个字体中，‘Family’（家族）和‘Style’（样式）是相同的，然后 Flash Studio 可能会由于与第一个字体重复而丢弃第二个字体。比如说，我们这次选择‘Times New Roman’。
7. 再创建一个 FLA，同样的 ActionScript 设置。将其称为‘fonts_kr.flaswf’或‘fonts_ru.flaswf’，或者任何其他名称，具体取决于您要使用的语言。假如是 Korean（韩语），那么这个名称就是‘fonts_kr.flaswf’。

8. 转到 ‘Library’ (库) 窗口, 单击鼠标右键, 选择 ‘New Font...’ (新建字体...)。将名称指定为 \$Font1, 但选择 “Family” (家族) 为某种韩语字体 (例如作为 “바탕”), “Style” (样式) 为 “Regular”。
9. 在 “Character ranges” (字符范围) 中, 选择要嵌入的字符。对于韩语, 它可能是 ‘Korean Hangul (All)’ (韩文 (全部))。
10. 切换到 ‘ActionScript’ 选项卡。复选 ‘Export for ActionScript’ (针对 ActionScript 导出)、‘Export for frame 1’ (针对帧 1 导出)、‘Export for run-time sharing’ (针对运行时共享导出)。在 ‘URL’ 输入字段中, 键入 ‘fonts_en.swf’。单击 ‘OK’ (确定) 按钮。对于 ActionScript 3.0, Flash Studio 将给您显示一条警告 ‘A definition for this class could not be found...’ (找不到此类的定义...) – 忽略此警告, 并再次单击 ‘OK’ (确定)。
11. 为另一个字体重复步骤 8 – 10, 将其命名为 ‘\$Font2’, 并选择 ‘Family’ (家族) 为, 比如说, ‘바탕체’。
12. 同时发布 ‘fonts_en.swf’ 和 ‘fonts_kr.swf’。
13. 现在就可以回到我们的 main-app.flw 上来了。但首先转到默认字体库文件 (即 ‘fonts_en.swf’), 转到 Library (库), 在那里选择 ‘\$Font1’ 和 ‘\$Font2’, 并将其复制到一个剪贴板 (Ctrl-C 或右击 -> ‘Copy’ (复制))。
14. 切换到 ‘main-app.flw’, 转到 Library (库), 粘贴字体符号 (Ctrl-V 或右击 -> ‘Paste’ (粘贴))。您就会看到您的字体, 其中包含 ‘Import:’ (导入:) 前缀。
15. 在 ‘main-app.flw’ 的阶段上创建一个文本字段。使其成为 ‘Classic Text’ (经典文本)、‘Dynamic Text’ (动态文本)。从下拉列表 ‘Family’ (家族) 中选择 ‘\$Font1*’。键入 ‘\$TEXT1’, 作为该文本字段的上下文 (翻译时, 这将用作 ID)。
16. 再创建一个文本字段, 选择 ‘\$Font2*’, 在其中键入 ‘\$TEXT2’。
17. 发布 ‘main-app.swf’, 我们在这里的工作就完成了。
18. 接下来, 需要在与存储所有 swf 相同的目录中创建一个 fontconfig.txt。打开 ‘Notepad’ (记事本) 或支持 Unicode 或 UTF-8 的任何其他文本编辑器。键入以下代码:

```
[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：尽管我们使用了字体的韩语名称（“바탕”和“바탕체”），但Flash可以在SWF中使用英语名称（“Batang”和“BatangChe”）。这就是我们在fontconfig.txt中使用英语字体名称的原因。要确保在SWF中使用哪些字体名称，您可以使用`gfxexport -fntlst <swfname>`命令并分析输出*.lst文件。

19. 保存该文件（不要忘记将其另存为Unicode或UTF-8！）。大功告成！在GFXPlayer中打开‘main-app.swf’，您就会看到默认文本‘This is’（这是）和‘English!’（英语！）。使用Ctrl-N切换到Korean（韩语），您就会发现如何使用fonts_kr.swf中的字体（由于我们仅在fonts_kr.swf中嵌入了韩语字形）用韩语替换英语文本。

3.2 自定义资源生成

顾名思义，自定义资源生成就是为每一个目标语言发行创建自定义的SWF/GFx文件。例如，如果一个游戏使用的是“UIScreen.fla”文件，每一个游戏发行就会手动生成一个SWF的不同版本，即朝鲜语的“UIScreen.swf”版本和英文的“UIScreen.swf”不同版本。这个文件的不同版本可以放置到不同的文件系统目录，或者只放在目标市场而不作发行。

使用自定义资源的主要优点是，可以在每个文件中嵌入最小数目的字型。虽然一般我们不建议这种做法，但当内存极为有限（如供用户界面使用的内存少于600K），并且艺术资源符合下列条件时，可以考虑使用自定义资源生成：

- 用户界面资源文件相对较小，并且其文本内容很简单。
- 很少有多用户界面文件一起下载（如果它们是一起下载的，其他方式所提供的字体分享功能将非常有用）。
- 动态文本域更新有限，并且嵌入字符的使用数量很少。
- 用户界面不需要亚洲语言IME支持。

如果开发人员选择采用自定义资源生成一个游戏名称，建议阅读Flash Studio文件中的“利用字符串面板编辑多语种文本”主题，并考虑使用字符串面板（按Ctrl + F11键）编辑国际化文本。开发人员需要将“Replace strings”设定为“manually using stage language”，以便使Flash替代文本可以在当前Scaleform版本中正常工作。

4 第 3 部分：设定字体资源

Flash的每个文本域都带有一个字体名称，这个字体名称或直接储存，或编码于HTML标记内。当显示文本域时，通过搜索以下字体资源可以获得所用的渲染字体：

1. 本地嵌入字体。
2. 从单独的 SWF / GfX 文件导入的字体符号。
3. 通过 GfX::FontLib 安装的、通过字体名称或导入替代字体搜索到的 SWF/GfX 文件。
4. 系统字体提供程序，如 GfX::FontProviderWin32。前提是用户安装了这样的程序。

嵌入和导入字体的使用在本文件的开始部分已经做了描述。在大多数情况下，嵌入字体的运作与Flash相同，因此不需要自定义配置。在查找字体的目的方面，导入字体符号的运作与嵌入字体类似。

用于设定非嵌入字体查找的三种可安装状态分别是GfX::FontLib、GfX::FontMap和GfX::FontProvider。正如本文件前面介绍的那样，GfX::FontLib和GfX::FontMap用于为导入替代字体或设备字体仿真查找SWF/GfX加载的字体。下面将详细介绍如何使用系统字体提供程序。

系统字体提供程序与调用指令GfX::Loader::SetFontProvider一起安装，它允许字体数据来自另一种非SWF文件源。字体提供程序只能与动态缓存一起使用，并且只有当字体数据不是嵌入在SWF或字体库中的时候才被搜索。目前，有两种字体提供程序含有Scaleform：

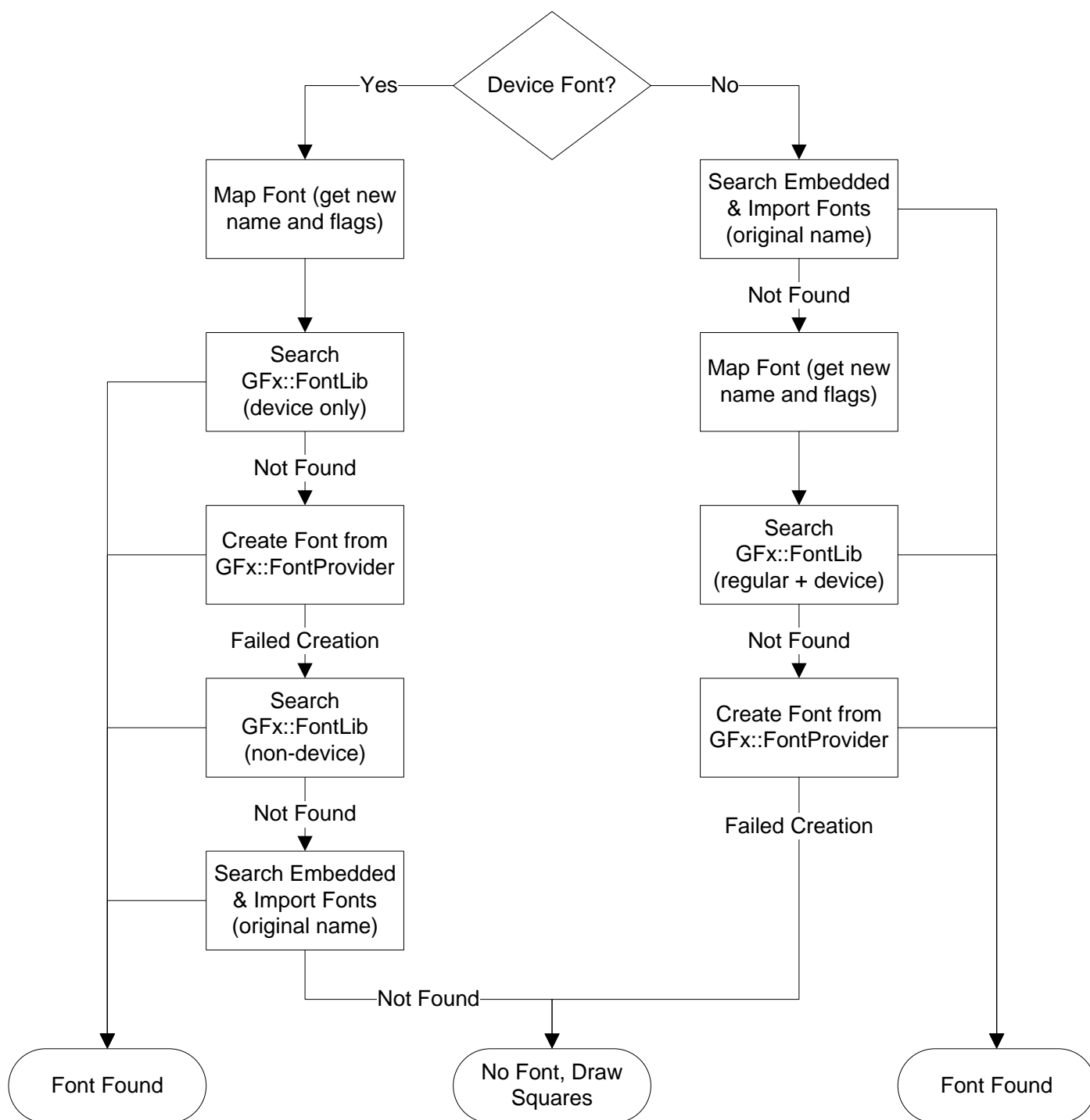
- GfX::FontProviderWin32 – 依靠 Win32 APIs 来获取字体字型数据。
- GfX::FontProviderFT2 – 使用 David Turner 开发的 FreeType-2 字体库来读取和解读独立的字体文件。

在这部分我们详细描述了字体查找的顺序，并提供了代码实例，用来说明如何配置不同的字体资源。

4.1 字体查找的顺序

下一页的流程图解释说明了在Scaleform中查找字体的顺序。从中可以看出，通过设备字体标记可以修改查找，在文本域属性中选定“Use device fonts”渲染方法就可以设置设备字体标记。

在 Adobe Flash 中，设置设备字体标记即可从系统中获得相应的字体，且先于命名相同的嵌入字体（在这种情况下，后者被作为一种备选方案）。Scaleform 重复这个做法，但它是先搜索其安装的字体库，然后再搜索系统字体提供程序。这种设置并不会引起冲突，因为大多数控制面板便携式游戏利用共享字体库，而选择使用系统字体提供程序的游戏会使字体库处于初始化（空）状态。



从上图可以看出，如果一个文本域不使用设备字体，就会首先搜索嵌入字体，反之，则最后搜索嵌入字体。此外，搜索嵌入式字体时总是以文本域中使用的原有字体名称为基础，而字体映射则可用于替代从 `GFX::FontLib` 和 `GFX::FontProvider` 中查找的字体名称。

4.2 Gfx::FontMap

Gfx::FontMap是一个用来替代字体名称的状态。如果在国际化过程中开发字体不具备所需的字符，Gfx::FontMap允许使用替代的字体。导入替代字体和设备字体仿真都使用字体映射。当字体映射用于导入替代字体时，它将字体符号标识符转换为字体名称。当字体映射用于设备字体仿真时，它将原来的字体名称映射到翻译的字体中。

在第3部分，通过使用下列字体配置文件中的命令行，来创建字体映射：

```
[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，使用映射命令将字体导入的标识符，例如“\$TitleFont”，映射到实际的字体名称，实际的字体名称又被嵌入在字体库文件中。

使用下列的C++命令可以得到同等的字体映射设置。

```
#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);
```

在这种情况下，所有这三个字体都被映射到相同的字体名称;特别是“\$NormalFont”“\$SmallFont”也有着相同的字体风格，这样就节省了字体库文件所占用的内存。对MapFont设定第三个参数，就可以形成不同的字体风格，这样就迫使映射使用特定的嵌入。如果不指定参数，MFF_Original值就会被使用，这意味着字体查找应保留文本域被指定的原有风格。字体大小可以由scaleFactor参数设置来改变。默认情况下，scaleFactor设置为1.0f。这个参数在字体显示不清楚时需要适当增强效果时将被用到。

开发人员应该注意，Gfx::FontMap 是一个绑定状态，这意味着利用 Gfx::FontMap 创建的动画即使后来加载时被改变，也仍将使用这个状态。如果设置不同的字型映射，Gfx::Loader::CreateMovie 调用指令就会给它返回一个相同文件名的不同的 Gfx::MovieDef。这也适用于其他所有的字体配置状态。

4.3 Gfx::FontLib

如第 3 部分所述，Gfx::FontLib 状态代表一个可安装的字体库，该字体库 (1) 为从默认 “fonts_en.swf” 中导入的字体提供替代选择，并 (2) 提供用于设备字体仿真的字体。先在字体库中搜索，然后在系统字体提供程序中搜索。下面的例子清楚的解释了如何使用 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);
```

这个原理是像正常情况一样创建并载入动画，但不同的是，此处的动画是用来作为字体存储，而不是用于播放。它可以加载尽可能多的所需动画。如果不同的动画界定了相同的字体，则只使用第一个。

AddFontsFrom 的第一个参数是动画的清晰度，这个清晰度将作为字体的一个资源。第二个参数是插件标记，如果装入程序要给内存中的动画添加AddRef，则应设置此参数。只有当用户不在加载动画上使用智能指针（例子中的M1和M2）的时候，才有必要使用此标记。如果插件标记处于非活跃状态，那么字体绑定数据，如导出或打包的纹理就可能被尽早释放，使用字体时就不得不重新载入/再生这些绑定数据。

与字体映射类似，Gfx::FontLib也是一个绑定状态，并为所创建的动画提供参照，直至消失。

4.4 Gfx::FontProviderWin32

Gfx::FontProviderWin32字体提供程序可以在Win32 API中使用，并依靠 GetGlyphOutline函数来检索矢量数据。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的构造函数将Windows Display Context的处理程序作为一个参数。在大多数情况下，可以使用屏幕DC (::GetDC(0))。

如有需要，字体将被创建。

4.4.1 使用自动提示文本

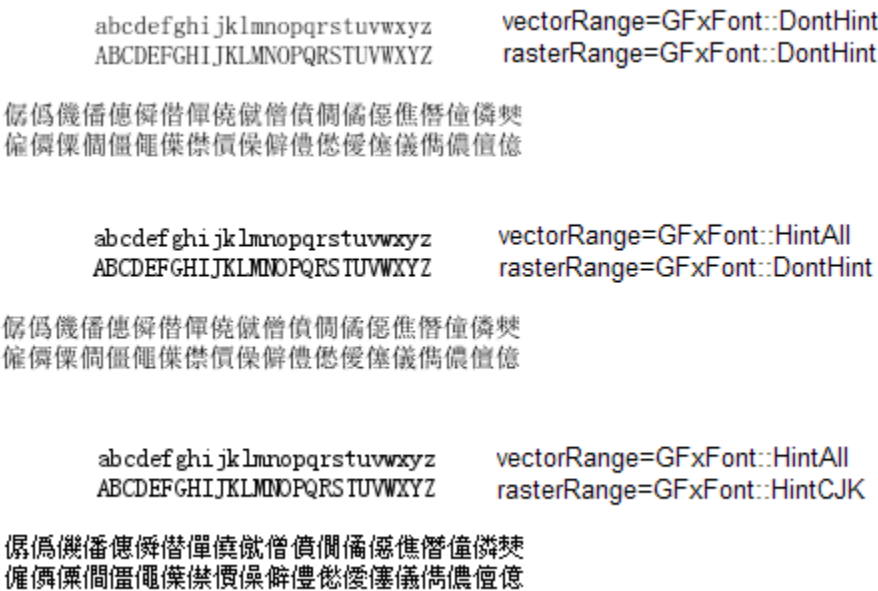
一般来讲，字体提示是一个非同寻常的难题。**Scaleform**提供了一个自动提示的机制，但是在中文，日文和韩文（中日韩）字符上这个自动提示的机制运行不佳。此外，大多数设计良好的中日韩字体中某些尺寸的字型含有栅格图像，因为中日韩的提示遇到小尺寸的字型就会变得非常复杂。复制含有栅格图像的向量字型是一个很好的而且切实可行的解决办法。以**font APIs (Gfx::FontProviderWin32 and Gfx::FontProviderFT2)**为基础的系统字体提供程序能产生矢量和光栅形式的字型。字体提供程序有一个界面，用来控制自动提示。各种界面略有不同，但有相同的原则。有四个参数：

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

参数**VectorRange**和**RasterRange**控制使用自动提示的字符的范围。有以下各值：

```
Font::DontHint - 不使用自动提示，  
Font::HintCJK - 中文，日文和韩文字符使用自动提示，  
Font::HintAll - 所有字符都使用自动提示。
```

RasterRange 优先于 **VectorRange**。参数 **maxVectorHintedSize** 和 **maxRasterHintedSize** 界定使用自动提示时像素中的最大字体尺寸。下图包括 **SimSun** 字体的例子，且含有不同的选项。



可以看出，自动矢量提示对这个字型没有多大用处，而使用栅格图像则可以明显地更改文本。然而，对于某些字体来说，矢量提示是有意义的，例如“**Arial Unicode MS**”字体。

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
𐀀𐀁𐀂𐀃𐀄𐀅𐀆𐀇𐀈𐀉𐀊𐀋𐀌𐀍𐀎𐀏𐀐𐀑𐀒𐀓𐀔𐀕𐀖𐀗𐀘𐀙𐀚
𐀛𐀜𐀝𐀞𐀟𐀠𐀡𐀢𐀣𐀤𐀥𐀦𐀧𐀨𐀩𐀪𐀫𐀬𐀭𐀮𐀯𐀰𐀱𐀲𐀳𐀴𐀵𐀶𐀷𐀸𐀹𐀺

请注意，与Adobe Flash不同的是，Scaleform支持设备字体的任意仿射变换。旋转和倾斜当然会使字型模糊，但任意仿射变换仍然相当适合动画文本。

默认情况下，Win32和FreeType两个提供程序都使用以下值：

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

设定字体提示的特别接口将在下面进行介绍。

4.4.2 设定自动提示

请注意，如果设置了SetHinting()之后再调用SetHintingAllFonts()函数，那么这个具体字体的提示操作不会更改。确切的函数原型是：

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

或

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

请注意，如果在设置了提示 SetHinting() 后调用 SetHintingAllFonts() 函数，那么这个具体字体的提示操作不会更改。确切的函数原型是：

```
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);
```

参数Name可以使用UTF-8 编码。

4.5 Gfx::FontProviderFT2

这个字体提供程序使用David Turner开发的FreeType-2 library字体库。它的使用方法与Gfx::FontProviderWin32类似，不同的是使用Gfx::FontProviderFT2需要将字体名称和属性映射到实际的字体文件。

首先，我们建议开发人员阅读FreeType手册，以便能恰当地配置和创建字体库。开发人员需要决定使用静态还是动态链接，以及适当的运行时间配置设置，例如字体驱动器、内存配置器、外部文件流等。

使用静态链接的Windows MSVC编译程序应该使用下列字体库：

freetype<ver>.lib – 用于释放多流 DLL 代码生成，
freetype<ver>_D.lib – 用于调试多流 DLL，
freetype<ver>MT.lib – 用于释放多流 DLL(静态 CRT)，
freetype<ver>MT_D.lib – 用于调试多流 DLL (静态 CRT)。

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

此外，要确保freetype2/include和freetype2/objs这两个目录分别可以在额外的引用功能和字体库两个路径上使用。

Gfx::FontProviderFT2对象的创建过程如下：

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

构造函数含有一个参数：

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

这是FreeType字体库处理。如果这个值为零（默认值），供应程序将在内部初始化FreeType字体库。当应用程序已经使用了FreeType字体库，并欲在Scaleform和其它系统上共享这一处理时，能够指定一个现有的初始化外部处理程序。注意，当使用外部处理程序的时候，开发人员应注意妥善释放字体库（调用指令：FT_Done_FreeType）。此外，应用程序必须保证处理的寿命要长于Gfx::Loader的寿命。使用外部处理程序也是因为有时需要配置FreeType字体库（内存配置器等）的运行时间回调。请注意，此配置也可以被内部初始化。fontProvider->GetFT_Library()这个函数将使用的FT_Library 处理程序返回，并确保在实际使用字体之前供应程序不会调用FreeType字体库。这样就有机会在创建Gfx::FontProviderFT2后重新配置FreeType字体库运行时间。

4.5.1 将 FreeType 字体映射入文件

与Win32 API不同的是，FreeType字体不会将映射字体的名称（和属性）提供给实际的字体文件。因此，这个映射必须从外部提供。Gfx::FontProviderFT2 有一个简单的映射机制，可以将字体映射到文件和内存。

```
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);
```

参数“fontName”指定字体字型的名称，例如，“Times New Roman”。参数“filename”指定字体文件的路径，例如，“C:\\WINDOWS\\Fonts\\times.ttf”。“fontFlags”的值可能为“Font::FF_Bold”，“Font::FF_Italic”，或“Font::FF_BoldItalic”。“Font::FF_BoldItalic”这个值实际上等于“Font::FF_Bold | Font::FF_Italic”。“faceIndex”是被传递到 FT_New_Face 的。在大多数情况下，这个参数是 0，但并非总是如此，这取决于字体文件的类型和内容。请注意，这个函数实际上并不打开字体文件;而只是构成映射表。只有真正被要求时字体文件才被打开。与 Win32 字体提供程序不同的是，FreeType 字体只是使用函数参数来配置自动提示。

4.5.2 将字体映射到内存

FreeType允许将字体映射到内存。例如，如果一个单独的字体文件包含了很多字型，或如果字体文件已经被应用程序加载，这个配置可能是有意义的。

```
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);
```

下面是一个将字体映射到内存的简单例子。

```
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);
}
```

在这个例子中，内存没有被释放（而且最终会导致内存泄漏）。这是因为映射表只在字体数据保持一个裸指标常数。所以开发人员应该负责妥善释放内存。应用程序必须保证这个内存块的寿命要**长于** **Gfx::Loader**的寿命。它旨在为字体映射机制提供尽可能多的自由。例如，应用程序可能已经使用了含有预先载入内存的字体的**FreeType**，其中字体的分配和销毁都由**Scaleform** 在外部处理。

在Windows中使用**FreeType**字体映射如下例所示。

```
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);
```

请注意，这只是一个例子，在一个真正的应用程序中指定绝对硬编码文件路径的做法是不可取的。通常它应该来自一个配置文件，或通过自动扫描字体的字型得到。指定明确的字体名称看起来好像小题大做，因为字体通常包含这些名字，但这样做避免了会占用很大内存空间的文件解析操作。函数 **MapFontToFile()** 只存储这些资料，但并不打开文件，除非它被要求打开文件。这样，可以指定大量的映射字体，而只有少数被真正使用。在这种情况下不用执行任何额外的文件操作。

5 第4部分：配置字体渲染

Scaleform可以用两种方式渲染文本字符。第一种，将字体字型光栅化为纹理，然后使用成批的三角形纹理（每字符两个）对它进行渲染。或者还可以将字体字型镶嵌成三角形网格，并将其渲染为矢量图形。

对于程序中的大多数文本来说，操作时应始终使用有纹理的三角形。字型的纹理通过动态或静态的缓存生成。动态字型缓存更加灵活，并能产生更高质量的图像，而静态缓存的优势是可以进行脱机计算。

开发人员可以根据目标平台和标题的需要，从下列项目中选择字体渲染选项：

1. 使用动态缓存，进行快速动画运行及高品质的导出。动态缓存会使用一个固定数额的纹理内存，并会减少加载时间，这是因为不是所有字型都需要被光栅化和/或载入。
2. 使用静态缓存，从磁盘载入预先生成的雾化贴图纹理。开发人员可以使用 `gfxexport` 工具来提前生成打包的字型纹理，剔除字体的矢量数据，这样它就不需要被加载。
3. 使用静态缓存，在加载时间使用 `GFx::FontPackParams` 程序自动生成纹理。这点类似以前的方法，不同的是纹理不需要从磁盘加载。
4. 使用矢量图形直接渲染文本字型。

建议在高端系统，如PC、Xbox 360、PS3中，以及很多情况下Nintendo Wii中使用动态缓存。动态缓存能确保某一分辨率使用最少的加载时间和最高质量的字体渲染。如果开发人员计划通过 `GFx::FontProviderWin32`或`GFx::FontProviderFT2`程序使用系统或外部字体支持，也需要使用动态缓存。

对于CPU和内存有限的平台，如PSP和PS2来说，使用静态缓存并由`gfxexport`工具导出纹理是一个很好的选择，因为在CPU和内存有限的平台中，对字型运行时光栅化可能会占用过大的存储空间。如果开发人员需要在执行渲染时避免动态纹理更新，也可以使用静态缓存。然而，由于我们优化了矢量数据存储占用，即使在较低等级的操作系统中，动态缓存也可能成为一个好的选择。我们建议开发人员对他们的游戏数据进行实验，以确定最佳的解决办法。

虽然Scaleform可以被这样配置，矢量图形很少单独用于文本渲染。相反，字型镶嵌通常只用于大型字型，而小型文本通常结合字型镶嵌和以纹理为基础的方法，以达到高效率的渲染。开发人员可以参考“矢量控制”章节以获取更多关于如何控制或禁用此选项的详细介绍。

5.1 配置字形缓存

Scaleform 中的字体渲染是通过 `Render::GlyphCacheConfig` 或 `GFX::FontPackParams` 状态对象配置的。默认情况下，字形缓存是由渲染线程上的 `Renderer2D` 对象自动创建的，并为动态字形进行了初始化。字型填充只用于静态纹理初始化；其参数默认为零。在这样的设置状态下，所有创建的动画除非来自被预处理为静态纹理的 **Scaleform** 文件，否则都会自动使用动态字体缓存。

GFX 4.0 升级说明

Scaleform 3.x 和 4.0 版本之间的动态字形缓存配置变化很大。尽管 **GFX 3.3** 依赖于 `GFXLoader` 维护并在主线程上配置的 `GFXFontCacheManager` 对象，但 **GFX 4.0** 还是用 `Renderer2D` 维护且在渲染线程上配置的 `GlyphCacheConfig` 接口所取代。有关在 **Scaleform GFX 4.0** 中配置缓存的更多详情，请参阅第 5.2 节。

当从一个纹理渲染光栅字形时，始终使用字形缓存系统。。从程序内部来说，缓存管理器负责维护文本批次顶点阵列，当静态和动态纹理缓存同时被使用时就会产生顶点阵列。当启用动态缓存时，缓存管理器负责分配缓存纹理，使用光栅字符对它们进行更新，并使纹理与批次顶点数据保持同步。当只有静态缓存被使用时，缓存管理仍需要保持文本顶点阵列，但它并不需要分配或更新动态纹理。

静态缓存呈现出预先光栅化点阵图纹理，含有密集的填充字型。静态纹理的光栅化和填充可以在加载时间通过 `GFX::FontPackParams` 进行，也可以脱机时用 `'gfxexport'` 工具进行。在这两种情况下，字体都有静态的纹理。嵌入字体的字型有或没有一套静态的纹理，依字体加载方式而定。如果字体有静态的纹理，他们将始终被用来渲染字体的字型；否则，就会启动动态缓存以渲染字体的字型。

渲染方法除了要根据使用的字体纹理类型而定以外，还要根据目标字型的像素大小而定。更正式地说，要使用以下逻辑进行：

```
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，如果纹理缓存不可用，或者如果字型过大不能够被纹理渲染，就使用矢量渲染。缓存方法的选择要根据字体填充字型纹理的可用性而定。以下各节将详细介绍如何设置这两种方法。

5.2 利用动态字体缓存

当字体和字符集被限制为例如基本拉丁语、希腊语、斯拉夫语等时，静态缓存是很有效的。然而，对于大多数亚洲语言来说，静态缓存可能会消耗太多的系统和视频内存，从而导致加载缓慢。此外，如果使用太多不同的字体也可能发生这种情况；换句话说，也就是当嵌入式字型总数过大（比如10000个或更多）的时候。在这些情况下，应该使用动态缓存机制。此外，动态缓存可以提供更多功能，如优化可读性，大大提高字体的质量。默认情况下，动态缓存已启用，并分配其缓冲区；要禁用动态缓存，您可以调用：

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

在上面调用中，`renderer` 为一个 `Render::Renderer2D` 对象，并在渲染线程上进行维护。此调用将把字形缓存使用的动态纹理数量设置为零，从而有效地将其禁用。要避免默认临时缓冲区分配，应在初始化渲染 HAL 之前执行此调用。

当渲染文本时，动态缓存将字型光栅化，并根据要求更新各个纹理。它使用一个简单的LRU（近来最少使用的）缓存方案，但却含有一个智能适应字型即时填充纹理。您可以按照如下方式设定纹理参数：

```
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);
```

上述值是默认使用的。

`TextureWidth`, `TextureHeight` - 缓存纹理的大小。这两个值在 2 幂运算时都四舍五入。

`MaxNumTextures` - 用于缓存的纹理的最大数目。

MaxSlotHeight - 字型的最大高度。像素字型的实际高度不能超过此值。更大的字型被渲染为矢量图行。


SlotPadding - 用来防止字型削波和重叠的盈余值。在大多数实际情况第二个值就很适用。

TexUpdWidth, TexUpdWidth - 用来更新纹理的图像大小，在实际情况通常大小为 256x512 (128K 的系统内存)的图像就很合适。也可以将大小降至 256x256 或甚至 128x128，但如果这样做的话，需要更频繁地进行纹理更新操作。

动态字形缓存 用来压缩字体轮廓，动态运行，采用“最近使用最小”缓存策略。要了解字体字形缓存如何工作，请看一个非常简单的内存分配器，它可以在一个给定的 1MB 的空间中仅分配 4 KB 内存块。分配器只能对这些 4KB 存储块进行分配或删除。显然，分配器同时最多可以分配 256 个数据块。但是在大多数情况下，请求的数据块远远小于 4K。可能为 16 字节、100 字节、256 字节等，但是不能超过 4K 字节。在这种情况下，就需要设计一种机制来处理 4K 空间里的小数据存储块以获得 1MB 存储空间中获得更大的存储容量。但是支持的最小存储单元是相同的 - 同时可分配 256 个存储区域。动态字形缓存中有一个类似的运行机制，但为二维的文本空间。也就是说，动态字形缓存可以存储 $\text{MaxSlotHeight} + 2 * \text{SlotPadding}$ 个单元，但是对字体压缩度更高，特别是在小的存储空间。平均情况下这可以增加动态字体缓存容量 2-5 倍（压缩字体轮廓，增加存储空间需要 10 秒时间）。通常，60-80% 纹理空间被用来有效载荷，不依赖于字体大小。

总缓存能力（即，同时储存在缓存内存的若干不同字型的最大数量）要根据平均字型大小而定。对于典型的游戏用户界面，我们粗略估计上述参数缓存能力在 500 至 2000 个不同字型之间。在典型的情况下，约 70-75% 的界面是有效运用的。缓存字型的最大数目必须足够多，以便能够处理任何一个单行文本域的可见部分。在这种情况下，如果一个单行文本域可见部分的不同字型的数量超过缓存内存的能力，其余的字型就被渲染为矢量图行。

前面提到，动态缓存允许额外的能力并支持“提高可读性的反锯齿”和“提高动画效果的反锯齿”选项。这些选项是文本域的属性，Flash 设计者可以在文本对话框面板中选择它们。优化了可读性的文本看起来更为锐化和更具可读性。虽然它也可以被制成动画，但这个动画耗费较大，因为它会造成更频密的纹理更新活动。此外，字型是自动添加像素网格以减少像素自动拟合运作的模糊强度的，这就意味着文本行也被添加了像素。在动画时，尤其是缩放时，视觉上有一个抖动的效果。下图显示了这些选项之间的差别。

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

当使用“提高可读性的反锯齿”选项时，字型光栅器执行自动拟合程序（也称为自动提示）。Scaleform

不使用来自Flash文件或任何其他字体来源的字型提示。它渲染文本所需要的就是字型概述。它提供字体源以外的外观相同的文本。Scaleform自动提示程序需要字体的某些资料，即平顶的拉丁字母的高度。必须准确管理超调量字型(例如O、G、C、Q、o、g、e等)。通常情况下，字体不提供这些资料，因此，必须在某种程度上将这些资料推导出来。为此Scaleform使用平顶的拉丁字母，他们是：

- 大写字母有：H, E, F, T, U, V, W, X, Z,
- 小写字母有：z, x, v, w, y。

为了使用自动适应程序，字体必须包含至少一个上述的大写字母和至少一个上述小写字母。如果字体不包含上述字母，Scaleform将产生一个日志警告：

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

“警告：字体'Arial':没有提示字符('HEFTUVWXZ' and 'zxvwy'中的任何一个)。自动提示为禁用状态。”

如果显示了这个警告，Flash设计者应该立即嵌入这些字母。通常情况下，只要在“嵌入字符”对话框添加“Zz”或嵌入“Basic Latin”就足够了。

5.3 使用字体压缩器 – gfxexport

命令行 gfxexport 工具的目的是预处理 SWF 文件，生成分发并加载到游戏中的 GFX 文件。预处理过程中，该工具可将图像从 SWF 文件中剥离，并将它们提取到外部文件中。可以用许多有用的格式存储外部文件，例如，DDS 和 TGA。选定 -fc 选项时，gfxexport 也将压缩字体矢量数据。这是有损压缩，因此您可能需要用参数进行试验，以便于找到内存消耗与字体质量之间的最佳折衷方案。

命令行选项	行为
-fc	启用字体压缩器。
-fcl <size>	设置名义字形大小。小的名义大小将会导致较小的数据大小，但字形较不精确。默认值为 256。多数情况下，名义大小 256 可节约大约 25% 的内存（与一般在 Flash 中使用的 1024 的名义大小相比），而且质量不会明显降低。不过，对于确实大的字形，您可能需要增加名义大小。
-fcm	合并已压缩字体的边缘。一个 Boolean 标志告知 FontCompactor 是否应该合并相同的轮廓和字形。合并时，数据会变得更加密实，并可节约 10-70% 的内存 – 具体取决于字体。但是，如果字体中包含太多字形，哈希 (Hash) 表可能会消耗额外的内存，每个唯一路径 12（32 位）个或 16（64 位）个字节，加上每个唯一字形 12（32 位）个或 16（64 个）个字节。

5.4 预处理字体纹理 - gfxexport

当 `-fonts` 这个选项被指定时，`gfxexport` 也可以光栅化和导出填充字体纹理，并将他们保存在用户指定的一个格式里。当在 `Scaleform Player` 中加载 `Scaleform` 文件时，外部的纹理将被自动加载，并在文本渲染时被用作一个静态缓存。我们不建议将导出的纹理用于典型 `GFx` 应用中。不过，这可能对低端移动平台有益，以及在某些特殊情况下有益。

利用 `gfxexport` 来生成字体纹理具有以下优点：

- 保存外部的纹理，这样即可去除字型的矢量数据，节省了内存。相反，字型数据被加载的静态纹理所取代。
- 加载纹理文件可能比在加载时间仅在 `CPU` 系统生成纹理文件的速度快。
- 纹理文件可以转化为紧凑的具体游戏格式，并可以通过覆盖 `GFx::ImageCreator` 而被加载。在您执行直接支持这些格式的 `Render::Renderer` 时，会发现这一点很有帮助。

使用预先生成的纹理确实也有缺点，如文本渲染质量较低，而且与使用动态缓存相比，载入时间可能更长。静态文本使用雾化贴图，导致不能提示，这意味着它不支持“提高可读性的反锯齿”的设置。

下面的命令行将把 `test.swf` 文件预先处理为 `test.gfx` 文件，为所有的嵌入式字体创建额外的纹理文件。

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

选项 `-strip_font_shapes` 将把嵌入字体的矢量数据从产生的 `Scaleform` 文件中去除。虽然这样做会节省内存，却使这些数据不可能被调回用于大型字符的矢量渲染，使字体纹理字型超过标称大小时质量下降。

下表列出了 `gfxexport` 与字体相关的选项。这些选项用于控制纹理的大小、字型的标称大小和目标文件格式。大部分选项与字型填充参数相符，因为 `gfxexport` 在生成字体纹理时要使用字型填充程序，字型填充参数在下一节中将进行描述。

命令行选项	操作
<code>-fonts</code>	导出字体纹理。如果未指定，将不会产生字体纹理（可以在动态缓存或加载时间填充时进行）。
<code>-fns <size></code>	纹理字型像素的标称大小；如果没有指定，则默认值为 48。如果纹理字型为最大尺寸，采用标称大小。使用 <code>tri-linear mip-map</code> 滤器对较小的字符进行运行时间渲染。
<code>-fpp <n></code>	在单个字型图像周围留有的像素空间，默认值为 3。
<code>-fts</code> <code><WxH></code>	字型被填充的纹理的尺寸。默认大小为 256x256。在指定方形纹理时，只

命令行选项	操作
	可指定一个大小，例如：'-fts 128'指的是 128x128'。'-fts 512x128' 指定的是矩形的纹理。
-fs	为每个字体添加单独的纹理。默认情况下，字体共享纹理。
-strip_font_shapes	不要将字型形状数据写入生成的 GfX 文件。
-fi <format>	当<format>为 TGA8(灰度化)，TGA24 (灰度化)，TGA32 或 DDS8 时，为每个字体纹理指定导出格式。默认情况下，如果图像格式(-i option)为 TGA，那么 TGA8 就被用于字体的纹理，否则就用 DDS A8。

警告：如果您计划在您的游戏中仅使用打包的静态字体或字形打包器，您应如 5.2 节所述禁用动态字形缓存纹理。如果不执行此操作，就会仍然分配默认纹理，而且默认纹理仍然保持未使用状态。

5.5 设定字体字型填充程序

载入文件时，字体字型填充程序将嵌入字型光栅化并填充。此外，字体也可能被GfXExport预光栅化。前面讲过，字体缓存管理器可以同时使用动态和静态两种机制。举例来说，字体字型填充程序可以被设置为使大型字符集的字体使用动态缓存，而使只有Basic Latin字符的字体预先光栅化并使用静态缓存。

前面讲过，一般的策略是，如果静态缓存可用，就使用预光栅化静态纹理。否则如果动态缓存被启用，就使用动态缓存。

默认状态下字体字型填充程序是禁用状态，这意味着Scaleform将对所有的嵌入式字体将使用动态缓存，除非程序明确创建并设定了填充参数。按照下列命令可以做到这一点：

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

然而，当使用GfXExport（以及各个Scaleform文件）时，字型可预光栅化。上述指令只是表示“加载时不要填充任何字型”；如果字型被GfXExport预填充，它们将被用来作为静态缓存。

如果应该使用静态缓存（嵌入式字型数量较少），可以按照以下方式进行配置。

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


`SetUseSeparateTextures()` 控制填充。如果它被采用，填充程序将对每一个字体使用单独的纹理。

否则，程序会把所有字型填充的尽可能紧密。使用单独的纹理可能减少纹理之间切换的次数，因此也会减少基本渲染的次数。但它通常会增加系统的已用内存和影像内存。其默认值为：无。

`SetGlyphCountLimit()` 这个参数控制填充字型的最大数目。默认值为0时，意味着数目没有限制。如果字体的嵌入字型总数超过此限制，字体就不会被填充。当亚洲语言与基于拉丁文字的语言或其它语言一起使用时，应该设定这个参数。如果您将这个限制设置到500，大部分的亚洲字体将被动态缓存（如果动态缓存被启用），而对字型数目较少的字体使用静态的纹理。

`SetTextureConfig()` 控制所有的纹理参数，它们是：

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

上述值是默认的。

NominalSize - 存储在纹理中反锯齿字型的标称尺寸（像素）。此参数控制纹理中最大字型的尺寸；大多数字型比这个参数要小得多。这个参数也控制纹理内存的使用和大型文本的锐化之间的权衡。注意它被称作“**NominalSize**”。与动态缓存不同的是，静态缓存使用实际包围盒填充不同大小的字型，而在动态缓存中字型是被拉到字型插槽的。**NominalSize** 这个值的大小与文本的像素高度是完全一样的值。这也意味着动态缓存的“分辨率的能力”比静态缓存稍微好一些。

PadPixels - 在单个字型图像周围留有空间的大小。这个值至少应为1。这个值越大，被缩小的文本的边缘将越平滑，但同时也浪费了更多的纹理空间。

TextureWidth, TextureHeight - 字型将要被填充的纹理的尺寸，这些值将使用2幂运算。

几种使用的情况及其意义列举如下：

- 1) 一切都采用默认设置。动态缓存被启用;字体字型填充程序未被使用。除了从预处理加载的预光栅化字型的纹理以外，其它部分都使用动态缓存。

- 2)

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

字体字型填充程序一直被使用。动态缓存被禁用。

- 3)

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

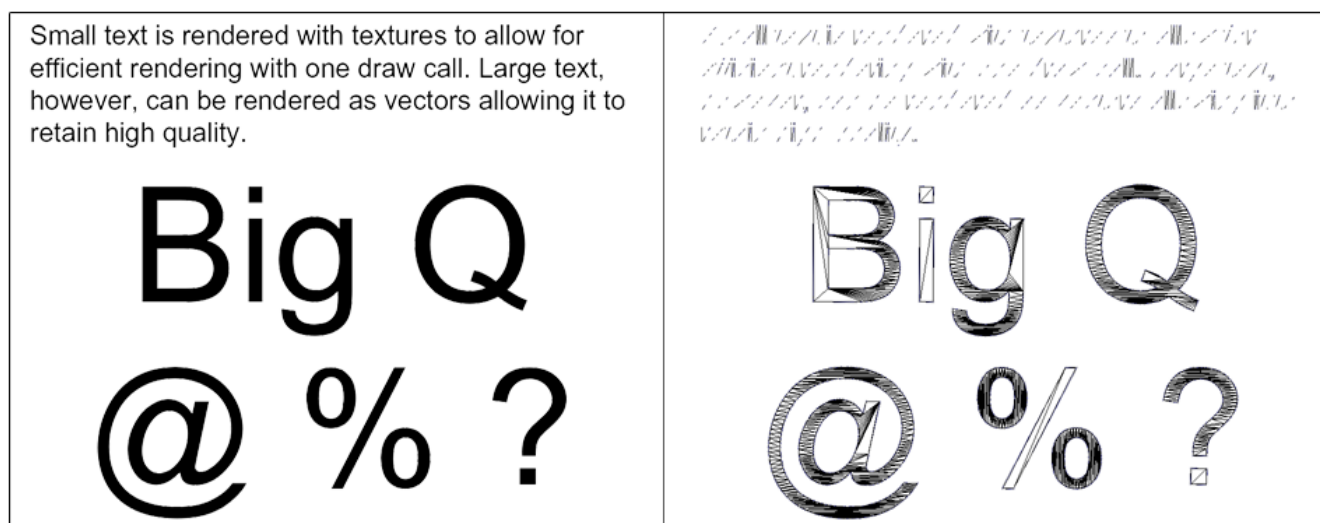
字体字型填充程序和动态缓存都被使用。含有500个或更少嵌入式字型的字体使用预光栅化静态的纹理，其他的字体使用动态缓存。

同时有必要提到的是，这些功能中的一些仅适用于动态缓存。举例来说，优化可读性的文本含有自动像素网格拟合功能（所谓的自动提示），只适用于动态缓存。任何效果，如模糊，阴影和辉光等也只适用于动态缓存。一般来讲，字型填充程序和静态缓存适合于性能欠佳的低预算系统。

5.6 矢量控制

本文件前面已经介绍过，镶嵌矢量图形在**Scaleform**中用作渲染大型字型时的备选，因为大型字型不适合纹理缓存。因为在**Flash**中对个体文本字符的大小没有限制，对它们进行纹理渲染所需的内存量在一个特定点之外就被禁止。解决这一问题的一个方法是限制字型位图的大小，并从那个特定点开始使用双线性过滤。但是，这样做将导致渲染质量迅速下降。

为了能够渲染大型的高品质文本，**Scaleform**能够将字型渲染转换为三角形图形，然后利用边缘反走样技术。在大多数情况下，用户不会注意到这个转换，因为当文本字符尺寸增大时，字型图形的边缘将保持平滑。下图表明了纹理和三角形网格渲染的文本之间的差异。在 **Scaleform Player** 中，可执行按键 **Ctrl+W** 可对线帧模式进行切换，用户可以看到渲染是如何完成的。



虽然三角形渲染的字型看起来不错，但它们在渲染时需要更多的处理时间，因为三角形和原始渲染计数增加了。当使用动态缓存时，纹理字型的最高高度是由缓存内存的 **MaxSlotHeight** 值决定的，这个值可以通过 **Render::GlyphCacheConfig::SetParams** 指令进行变更。如果将要显示的字型放到纹理插槽的内部，这个字型会被纹理渲染，否则就使用矢量渲染。由于要使用的渲染技术是根据每个字型确定的，所以当其像素的高度接近缓存内存插槽的最高高度时，某一单行文本可能同时包含位图和矢量标志。由于有亚像素精度，进行不同渲染的符号的类型几乎难以区分彼此。

静态缓存的运作与此不同。静态缓存时，所有字型都是根据字型的指定标称尺寸预先光栅化的，并且使用三线过滤器进行了调整。由于字型的标称尺寸是固定的，文本渲染的方法也是根据字型的标称尺寸进行选择的，而不是根据个体字型的高度而选择的。

对于动态缓存和静态缓存，开发者可以通过修改 `GlyphCacheParams::MaxRasterScale` 值来控制纹理到矢量渲染开关发生的点。

`SetMaxRasterScale`的参数指定了纹理槽的最大尺寸（像素）的乘数，随后就会切换到向量。

`MaxRasterScale`的默认值为1。**1.25**这个值意味着除非屏幕上的字型尺寸大于存储在纹理里的字型的标称尺寸的1.25倍，否则播放器将使用纹理渲染文本。因为默认的标称尺寸值为48像素，矢量渲染将被用于在屏幕上大于60像素的字型。

如果 `MaxRasterScale` 的值设置得足够高，则矢量渲染将永远不会被使用。对于通过启用 `-strip_font_shapes` 选项执行 `gfxexport` 工具而产生的 `GFx` 文件，矢量渲染也是禁用。在后一种情况下，字体字型的数据不再保存在文件中，所以矢量化也是不可能的。

6 第 5 部分：文本过滤效果和动作脚本扩展

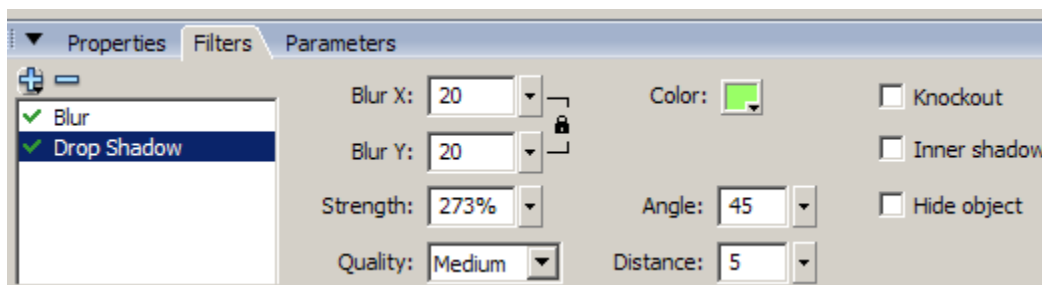
当动态字型缓存被启用时，Scaleform 2.x支持将模糊、下拉阴影和辉光滤镜这些效果应用到文本。支持文本过滤器需要动态缓存；当使用字型填充程序或静态字体纹理时，系统不支持过滤效果。在播放器的当前版本，过滤器只有被直接用于文本域时才能运作，当被应用到动画剪辑时，将不会有任何效果。

虽然Scaleform文本过滤器的运作类似于Flash，两者之间还是有一些不同。在Adobe Flash中,过滤器持续适用于整个被光栅化的文本域中。这意味着模糊、下拉阴影、以及辉光过滤器可能会被相继使用。这种方法具有很好的灵活性，但是计算量太大。与Flash不同的是，Scaleform内对过滤器的支持是有限的，但运行非常

快。与动态适配的字型缓存配合使用，过滤器的运作几乎与普通的文本一样快。虽然对过滤器的支持有限，Scaleform却可以提供很好地制作柔化阴影和辉光效果的能力。

6.1 过滤器类型，可用选项和限制

在Flash Studio内，过滤器中可一次添加一个文本域。



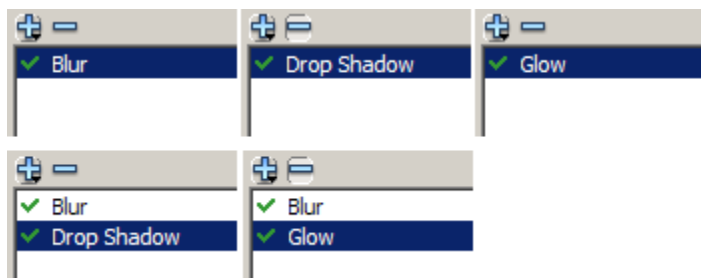
Flash和Scaleform过滤器之间的主要差别是，Scaleform在缓存中创建并存储模糊化字型的点阵图副本，而Flash将过滤器应用到所产生的文本域中。

Scaleform 只支持模糊和下拉阴影过滤器。辉光过滤器实际上是下拉阴影过滤器的一个子集。与 Flash 不同的

是，过滤器不是被持续使用的，相反，他们作为两个不同的层独立运行。在 Scaleform 内，程序只考虑一个下拉阴影或辉光过滤器，即两者中位于过滤器列表最后的那个。另外，一个可选的模糊过滤器也可用于文本本身。一般算法如下。

- 通过过滤器列表迭代；
- 如果列表最后的是“辉光”或“下拉阴影”，则存储阴影滤波参数；
- 每个“辉光”或“下拉阴影”覆盖列表中的前一个“辉光”或“下拉阴影”；
- 如果列表最后的是“模糊”，则存储模糊滤波参数；
- 每个“模糊”覆盖列表中的前一个“模糊”。

最后组合成的有效的过滤器为：

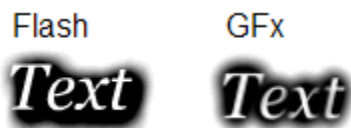


如果您在这个列表内添加下拉阴影和辉光过滤器，程序只考虑后者。

在Scaleform内的渲染也与Flash不同。先是“阴影”或“辉光”层被渲染，然后是文本本身被渲染上阴影，并有一个可选的“模糊”过滤器适用于原来的字型。如果“阴影”或“辉光”过滤器上显示“Knockout”（脱模）或“Hide Object”（隐藏对象）标记，文本（第二层的）将不会被渲染。此时Scaleform不支持“Inner Shadow”（内部阴影）和“Inner Glow”（内部辉光）。

由于这个算法的性质上的问题，在Scaleform内的过滤器有一定的限制。X和Y上的最大模糊值都被限制为15.9像素。最大的阴影或辉光强度也被限制为1590%。

“Knockout”选项的运作也不相同。在Flash中，原始图像是被整体脱去阴影的，同时保持阴影偏移量。而在Scaleform中，这个操作是逐个字型进行的，并忽略阴影偏移。此外，如果阴影或辉光半径较大，字型图像可能重叠和互相模糊：









因此，从基本上讲，只有对于半径较小的辉光过滤器（或对于阴影偏移为零的下拉阴影过滤器）来讲，“Knockout”这个选项才是有用的。

6.2 过滤品质

Flash使用一个简单的方块滤波来模糊图像。“品质”控制着过滤器运作的次数，而过滤器运作的次数严重影响生成图像的效果。举例来说，3x3的低品质模糊过滤器只能计算3x3像素区域的平均值。中等品质意味着相同的过滤器被应用两次；高品质意味着相同的过滤器被应用三次。这意味着品质对过滤的视觉半径影响很大。因为方法不同，视觉的结果也不同，但是很类似。事实上，在一个简单的、只应用一次的方块滤波生成的结果质量很差。

与Flash不同的是，Scaleform使用的是一个很好的高斯模糊滤波器和智能的递归算法，其速度并不依赖于过滤半径。在Scaleform内只有两个品质层：低品质和高品质，并且它们能生产非常类似的视觉效果

。只有当过滤器面板上设定为“Quality: Low”（品质：低）时，程序才会应用低品质。否则，Scaleform 使用的是高品质的过滤器。两个品质的区别在于，高品质的过滤器可以以分数的半径值 (sigma) 进行操作，而低品质的过滤器使用的是整数半径。在大多数情况下 Scaleform 模拟分数半径并适当调整字型。但如果文本域被反锯齿化以优化阅读，低质量的阴影可能看起来略有不准确。一般建议小型文本使用高品质的过滤器，以提高可读性。

	Low quality	Medium quality	High quality
Flash			
GFx			

从上图可以看到，在Flash中三种品质的差异是非常明显的;而在Scaleform中则差异很小。在Scaleform中只有低品质与其它两者不同，中品质和高品质产生的效果是相同的。

低品质过滤器运作更快，但字型缓存系统不存在这个差异。还有，低性能系统最好使用低品质的过滤器，尤其是当没有可用的硬件浮点运算的时候。

高品质过滤器需要浮点计算，并使用一种递归的执行程序，在此链接中有详细说明：

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

6.3 动态过滤

通过使用 Flash 的时间表和 Scaleform 的动作脚本扩展，可以生成动态的阴影效果。不过，这里有必要了解一下这种动态操作要付出的耗费。当改变半径、强度、或品质时，Scaleform 不得不重新生成字型图像并将其存储在缓存中。这将导致更频繁地更新缓存。事实上，更改前面提到的值相当于增加了字母的复合度。每个版本都得被储存在缓存中。相反，改变颜色（包括透明度）、角度、距离，并不影响性能。举例来

说，如果你想要实现阴影或辉光的淡入或淡出效果，最好使颜色的半透明度（透明度）动态化，而不是去改变半径和/或强度。

6.4 使用 ActionScript 中的过滤器

Scaleform 支持标准的过滤器类（如 DropShadowFilter、BlurFilter、ColorMatrixFilter、BevelFilter），这些过滤器类用于 AS2 和 AS3 的动态/输入文本字段。有关更多详细信息，请参阅 Flash 说明文档。